Python

Pedro Luis Kantek Garcia Navarro
UFPR

23 de agosto de 2018

Sumário

O autor, pelo autor

Meu nome é Pedro Luis Kantek Garcia Navarro, conhecido como Kantek, ou Pedro Kantek. Nasci em Curitiba há mais de 60 anos. Sou portanto brasileiro, curitibano e coxa-branca com muito orgulho, mas sendo filho de espanhóis (meus 7 irmãos nasceram lá), tenho também a nacionalidade espanhola. Aprendi a falar em *castellano* era o que se falava na minha casa, o português é meu segundo idioma, só visto na escola. Estudei no Colégio Bom Jesus e quando chegou a hora de escolher a profissão, lá por 1972, fui para a engenharia civil, mas sem muita convicção. As alternativas: medicina (arghhh!) ou direito (arghhh! arghhh!).

Durante a copa do mundo de futebol de 1974 na Alemanha, ao folhear a Gazeta do Povo, achei um pequeno anúncio sobre um estágio na área de processamento de dados (os nomes informática e computação ainda não existiam). Lá fui eu para um estágio na CELEPAR, que acabou mais de 35 anos depois. Na CELEPAR fui de tudo: programador, analista, suporte a BD (banco de dados), suporte a TP (teleprocessamento), coordenador de auto-serviço, coordenador de atendimento, ... Minha carreira lá terminou na área de governo eletrônico. Desde cedo encasquetei que uma boa maneira de me obrigar a continuar estudando a vida toda era virar professor. Comecei essa desafiante carreira em 1976, dando aula num lugar chamado UUTT, que não existe mais. Passei por FAE, PUC e cheguei às Faculdades Positivo em 1988. Na década de 80, virei instrutor itinerante de uma empresa chamada CTIS de Brasília, e dei um monte de cursos por este Brasil afora (Manaus, Recife, Brasília, Rio, São Paulo, Fortaleza, Floripa, ...). Em 90, resolvi voltar a estudar e fui fazer o mestrado em informática industrial no CEFET. Ainda peguei a última leva dos professores franceses que iniciaram o curso. Em 93 virei mestre, e a minha dissertação foi publicada em livro pela editora Campus (Downsizing de sistemas de Informação. Rio de Janeiro: Campus, 1994. 240p, ISBN:85-7001-926-2). O primeiro cheque dos direitos autorais me manteve um mês em Nova Iorque, estudando inglês. Aliás, foi o quarto livro de minha carreira de escritor, antes já havia 3 outros (MS WORD - Guia do Usuário Brasileiro. Rio de Janeiro: Campus, 1987. 250p, ISBN:85-7001-507-0, Centro de Informações. Rio de Janeiro: Campus, 1985. 103p, ISBN:85-7001-383-3 e APL - Uma linguagem de programação. Curitiba. CELEPAR, 1982. 222p). Depois vieram outros. Terminando o mestrado, rapidamente para não perder o fôlego, engatei o doutorado em engenharia elétrica. Ele se iniciou em 1994 na UFSC em Florianópolis. Só terminou em 2000, foram 6 anos inesquecíveis, até porque nesse meio tive que aprender o francês - mais um mês em Paris aprendendo-o. Finalmente virei engenheiro, 25 anos depois de ter iniciado a engenharia civil. Esqueci de dizer que no meio do curso de Civil desistí (cá pra nós o assunto era meio chato...) em favor de algo muito mais emocionante: matemática. Nessa época ainda não havia cursos superiores de informática. Formei-me em matemática na PUC/Pr em 1981. Em 2003, habilitei-me a avaliador de cursos para o MEC. Para minha surpresa, fui selecionado e virei delegado do INEP (Instituto Nacional de Pesquisas Educacionais) do Governo Brasileiro. De novo, visitei lugares deste Brasilzão que sequer imaginava existirem (por exemplo, Rondonópolis, Luiziânia, Rio Grande, entre outros), sempre avaliando os cursos na área de informática: sistemas de informação, engenharia e ciência da computação. Atualmente estou licenciado da PUC. Já fiz um bocado de coisas na vida, mas acho que um dos meus sonhos é um dia ser professor de matemática para crianças: tentar despertá-las para este mundo fantástico, do qual – lastimavelmente – boa parte delas nunca chega sequer perto ao longo de sua vida. Em 2018, comecei a dar aulas na UFPR.

O autor, há muitos anos, quando ainda havia cabelo...

Capítulo 1

O século XXI e a computação

1.1 Representação do conhecimento

Se olharmos a história deste nosso pequeno planeta e abstrairmos a hipótese Teológica (já que cientificamente ela não tem comprovação) veremos uma história mais ou menos assim:

1. Há coisa de 13.8 bilhões de anos começa a existir o nosso universo. O que havia antes ? Ninguém sabe, se é que existia algo. Como se sabe isso ? Porque se filmarmos o universo agora e voltarmos o filme, tudo converge para um ponto único no espaço passados 13.8 bilhões de anos.
2. 6 bilhões de anos atrás aparece o nosso planeta, ainda como uma bola incandescente de matéria ígnea. A partir disso, ela lentamente começa a esfriar de fora para dentro.
3. Há uns 3.000.000.000 de anos, surgem os primeiros seres vivos, capazes de – entre outras coisas – se reproduzirem por cópia de si mesmos. Aparece aqui o DNA (ácido desoxi-ribonucleico) o elemento comunicador da herança. Foi o primeiro e até hoje bem importante local onde se armazenou o conhecimento.
4. Mais tarde, surge o sexo. Ele tem enorme importância nesta história ao permitir que os filhos herdem um pouco do DNA do pai e um pouco da mãe, acelerando a evolução.
5. Um bocado de tempo depois, surgem os animais superiores e nestes o cérebro passa a ser responsável pelo local onde o conhecimento é produzido e armazenado. Denomina-se esta fase à do conhecimento extragenético.
6. Enquanto limitado ao conhecimento localmente produzido, o cérebro pode armazenar pouco. O próximo estágio de ampliação é quando surge a fala. Agora o conteúdo de um cérebro pode ser repassado a outro, com resultados melhores e crescentes. O nome é conhecimento extrassomático.
7. Com a fala, os dois sujeitos tem que estar tête-à-tête (cabeça a cabeça). A seguir, surge a escrita. Agora a comunicação é possível na modalidade um-muitos (antes era um-um) e mesmo um cérebro morto pode se comunicar pela escrita.
8. Enquanto a escrita é manuscrita, está naturalmente limitada. A próxima novidade é a imprensa que ao automatizar a produção de textos, amplia significativamente o acesso e a disseminação de informações. Outro pontos que não é muito comentado é o barateamento dos livros. Antes uma Bíblia era tão cara, que só paróquias ricas é que podiam comprá-la.
9. No século XX, surge o rádio, o cinema e a televisão. Mais conhecimento se difunde (se é que dá para chamar alguma coisa como o Faustão de conhecimento...)
10. Finalmente, o último estágio da ampliação do conhecimento reside no software. Hoje, quase 30 milhões de brasileiros sabem declarar seu IR, sem precisar fazer um exaustivo aprendizado sobre as regras da declaração, graças a um... software. Quando você vai fazer uma tomografia, o software interpreta os sinais de raio X e cria uma bela imagem do seu interior. O médico que faz o exame não tem a menor idéia de como a máquina opera. Os exemplos são quase inumeráveis. O nome disso é inteligência extrassomática.

1.2 A indústria da informação

Durante o século XX a maior indústria foi a do petróleo. Depois, a do automóvel e finalmente no final do século, a indústria de computadores passou a ser a maior. A tendência para o futuro é de que ela suplante todas as demais. Hoje o homem mais rico do mundo construiu a sua fortuna vendendo um software. Outros dois muito ricos fizeram fortuna inventando um produto novo (o Google).

O computador nasce na cabeça de um cientista inglês (Alan Matheson Turing em 1937) e é construído para a guerra pelos ingleses em 1942. A sequência de escopo do computador é

guerra a idéia aqui é usar o computador como otimizador dos recursos bélicos, criptografia, pesquisa operacional, tabelas balísticas...→

aritmética o computador se transforma em super-calculadora. A matemática passa a ser a linguagem da informática, por excelência. O FORTRAN é dessa época... →

negócios em plena sociedade capitalista, os negócios passam a ser otimizados pelo computador. Dessa época é o COBOL a linguagem representante... →

aldeia global Negócios, governos, universidades e organizações sociais passam a compartilhar o ciberespaço. Surge e se robustece a Internet... →

lazer o computador vira um eletrodoméstico. As casas de classe média passam a contar com um (ou mais de um) computador. Jogos, música, filmes, e-mule, mp3, ... →

... →

realidade virtual Não se sabe direito o que virá por aqui. As técnicas de simulação levadas ao paroxismo permitirão criar realidades virtuais, com toda a contradição exposta no título. Talvez retornemos à dúvida de Platão (*o universo existe ou é apenas uma sensação que eu tenho?*).

O computador se diferencia de qualquer outra máquina pela sua generalidade. No texto original de Turing isto fica claro pela conceituação de uma máquina universal. Assim, diferentemente de um liquidificador (que só serve para picar comida e tem apenas um botão de liga-desliga), um computador precisa – antes de qualquer coisa – ser programado para fazer algo. De fato, se comprar um computador vazio e ligá-lo, ele não vai fazer nada.

1.3 Componentes: hardware e software

Tecnicamente, chama-se a parte física do computador de **hardware** significando ferragem, duro (hard), componente físico, etc. Já a programação recebe o nome de **software** significando mole (soft em oposição a hard), componente lógico, não físico, programa de computador. Às vezes surge também o termo *peopleware*.

A fabricação de hardware é engenharia. Sua produção pode ser planejada e otimizada e isso de fato é feito. A fabricação de software, a despeito de ser chamada de engenharia, cá entre nós, não o é muito. Está mais para arte do que para engenharia.

Talvez, como seres humanos, devamos dar graças aos céus por esta situação. Pois enquanto a produção de hardware é (ou pode ser) em grande parte robotizada, a produção de software ainda exige um bom cérebro cinzento por trás.

As restrições que ainda existem sobre o software e que talvez nunca deixem de existir, impedem a programação dos computadores em linguagem natural. Hoje ainda não é possível dialogar com um computador como se fala com uma pessoa medianamente inteligente. Os computadores ainda não conseguem tal proeza.

Para programar um computador necessitamos uma linguagem. O hardware só entende uma linguagem de pulsos elétricos, chamada de linguagem de máquina. Para facilitar a nossa vida, já que é muito difícil para o homem programar nela, criaram-se programas tradutores que recebem comandos e dados em uma linguagem artificial e a convertem em linguagem de máquina. A maioria desses tradutores recebeu o mesmo nome da linguagem que eles traduzem. Assim, temos o Java, o Cobol, Pascal, Apl, Clipper, Dbase, Basic, Natural, Fortran, C, Php, Lisp, Prolog, Modula, entre outras. Veja no sítio www.tiobe.com o índice TPC que mensalmente lista as 100 linguagens mais usadas no mundo.

Quando os primeiros computadores foram construidos, a única linguagem por eles entendida era a binária. Sequências intermináveis de uns e zeros, algo assim como 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 1 0 0 1 0 1 0 1 0 0 1 0 0 0 1 0 1 0 1. Era um autêntico samba do crioulo doido. Chamou-se a isto, mais tarde, linguagem de primeira geração.

A seguir, o primeiro melhoramento: o assembler (ou montador), programa que era capaz de entender instruções escritas em uma linguagem um pouco mais humana, e traduzir cada uma destas instruções em sua expressão binária equivalente. Junto com este programa criou-se a primeira linguagem digna deste nome: o assembler, até hoje usado, se bem que muito raramente.

Programar em assembler ainda exige muito talento e paciência. A máquina precisa ser conhecida em inúmeros detalhes de construção, os programas são longos e difíceis –principalmente sua manutenção. Entretanto esta é a linguagem mais eficiente do ponto de vista dos consumo dos recursos da máquina, isto é, ela gera programas velozes e pequenos.

Depois, vieram as linguagens de terceira geração: a primeira foi o FORTRAN, depois COBOL, PL/I, PASCAL, C, C++, Java, LISP, Prolog, Python e outras menos votadas: BASIC, ADA, APL. Há um distanciamento do programador em relação à máquina, e uma aproximação do mesmo em relação ao problema a resolver. Estas linguagens são mais fáceis de aprender e embora gerem programas maiores e mais lentos, aumentam em muito a produtividade humana de escrever código de programação.

Um exemplo diferente para mostrar todos estes conceitos de linguagens poderia ser o seguinte: Suponha um engenheiro brasileiro, que só conhece o idioma português. Ele precisa transmitir certas ordens de fabricação para um colega soviético, que só conhece o idioma russo, com escrita cirílica. Os dois podem tentar conversar quanto quiserem, mas provavelmente não haver nenhum resultado prático útil. É necessária a presença de um tradutor, alguém que ouvindo o português

consiga transformar as ordens em suas equivalentes russas. Finalmente, o engenheiro brasileiro verá se as ordens foram corretamente traduzidas e executadas analisando o resultado final do trabalho.

Neste exemplo, o engenheiro brasileiro é o programador. O colega soviético é a máquina, que ao invés de russo, só entende a linguagem elétrica. O tradutor é a linguagem de programação.

Podemos ainda estabelecer dois níveis para linguagens de programação: baixo e alto. Linguagem de baixo nível é aquela que requer a especificação completa do problema nos seus mínimos detalhes. No nosso exemplo, equivaleria ao engenheiro brasileiro descrevendo tin-tin por tin-tin todos os passos para montar uma engenhoca qualquer. Já as linguagens de alto nível, pressupõe uma tradução "inteligente", o que libera o engenheiro brasileiro de informar todas as etapas. Ele diz o que deve ser feito, sem precisar estabelecer como isto vai ser realizado.

1.4 Arquitetura de Von Neumann

Todas as linguagens compartilham uma estrutura parecida, já que rodam todas no mesmo computador que é construído usando a chamada Arquitetura de Von Neumann. Bem resumido, essa arquitetura prevê a existência da memória, de uma unidade aritmético lógica, canais e o elemento ativo: a unidade de controle. Essa estrutura comporta os seguintes tipos de comandos ou ordens: entrada-saída de dados, aritmética, movimentação em memória, condicional e desvio.

Com apenas essas 5 ordens, são construídos todos os programas que existem ou que existirão em nosso mundo usando este tipo de computador.

Uma pergunta que se pode fazer aqui, é se é melhor aprender tudo de uma linguagem e se tornar um especialista nela, ou se é melhor conhecer várias, ainda que não chegando a muita profundidade em nenhuma.

Não há resposta fácil, mas apostar todas as suas fichas em uma única pode ser perigoso. A tecnologia tem substituído a linguagem "da vez" mais ou menos a cada 8, 10 anos. Pior, ao se fixar em uma linguagem, corre-se o risco de misturar questões do problema (do algoritmo) com questões do programa (da linguagem).

Como na vida real, parece ser melhor buscar ser poliglota, ainda que obviamente haja uma linguagem preferida. Neste caso, todo o esforço deve ser no núcleo mais ou menos comum a todas as linguagens e deixar em segundo plano o aprendizado das idiossincrasias de cada uma.

Fica claro que programar um computador é "emprestar inteligência ao computador". Ou seja, para programar a solução de um problema, há que saber resolver esse problema. E, não apenas resolver em tese, mas descrever detalhadamente todas as etapas da solução.

De fato, a programação é detalhista ao extremo.

Como se disse acima, o computador surge em 1937, descrito em um artigo denominado "on a computable numbers". Este artigo era tão inédito que quase não é publicado: faltavam revisores que o atestassem. O conceito matemático de computador está lá, quase 10 anos de existir algo físico que lembrasse o conceito.

Ao mesmo tempo, o exército nazista alemão começa a usar um esquema criptográfico muito inteligente através de uma máquina denominada ENIGMA, a qual, tinham eles certeza, era inexpugnável. Mal sabiam que do outro lado do canal da Mancha estava Turing. Ele criou um computador, batizado COLOSSUS, que quebrava o código ENIGMA em poucas horas. Foi o primeiro computador construído pelo homem. O ENIAC de que os americanos se gabam tanto, veio depois.

No meio da guerra, Turing foi enviado aos Estados Unidos e lá teve contacto com Von Neumann. Este é o autor da assim chamada arquitetura de Von Neumann, que é a que usamos até hoje em qualquer computador.

1.5 Algoritmo

> *Algorithms are the most important, durable, and original part of computer science because they can be studied in a language – and machine – independent way. This means that we need techniques that enable us to compare algorithms without implementing them.*
>
> Skiena, Steven - The Algorithm Design Manual.

Outra citação, esta do autor Ahmed Shamsul Arefin, diz:

> *A good algorithm is like a sharp knife - it does exactly what it is supposed to do with a minimum amount of applied effort. Using the wrong algorithm to solve a problem is trying to cut a steak with a screwdriver: you may eventually get a digestible result, but you will expend considerable more effort than necessary, and the result is unlikely to be aesthetically pleasing.*

Segundo mestre Aurélio, algoritmo é "Processo de cálculo, ou de resolução de um grupo de problemas semelhantes, em que se estipula, com generalidade e sem restrições, regras formais para a obtenção do resultado ou da solução do problema".

Algoritmo é a idéia que está por trás de um programa de computador. É a "coisa" que permanece igual ainda que esteja em um programa Java para rodar num PC em Curitiba, ou esteja em um programa PASCAL rodando em um Cray em Valladolid na Espanha.

Do ponto de vista da informática, algoritmo é a regra de solução de um problema, isto é, surgida uma necessidade buscar-se-á uma solução, ou construir-se-á um algoritmo capaz de solucionar o problema.

Já um programa de computador, (segundo Wirth) "é uma formulação concreta de algoritmos abstratos, baseados em representações e estruturas específicas de dados"e devidamente traduzido em uma linguagem de programação.

Outra definição de algoritmo, esta dos autores Angelo Guimarães e Newton Lages:

Algoritmo é a descrição de um padrão de comportamento, expressado em termos de um repertório bem definido e finito de ações "primitivas", das quais damos por certo que elas podem ser executadas". Para encerrar, e tomando alguma liberdade, podemos considerar um algoritmo como sendo uma receita de bolo.

O conceito de algoritmo deve ser entendido, para sua correta compreensão, em seus dois aspectos, a quem chamaremos estático e temporal. Na sua visão estática, um algoritmo é um conjunto de ordens, condições, testes e verificações. No seu aspecto temporal, o algoritmo passa a ser algo vivo, pois atua sobre um conjunto de dados de entrada, para gerar os correspondentes dados de saída.

Tais características não podem ser separadas, elas estão instrinsicamente ligadas. A dificuldade em fazer bons algoritmos, é ter em mente, enquanto se escreve o algoritmo (aspecto estático) o que ele vai fazer com seus dados (aspecto temporal).

Vamos exemplificar esta discussão, comentando sobre o algoritmo de solução de uma equação do segundo grau.

Dada a equação $Ax^2 + Bx + C = 0$, para encontrar as duas raizes, segundo a inesquecível fórmula de Bhaskara, procedemos da seguinte forma:

- Calculamos DELTA, que é B ao quadrado, menos o produto de 4,A e C.
- Se DELTA < 0, as raizes não existem no campo dos números reais
- Se DELTA $= 0$, as raizes são iguais
- A primeira raiz é (-B) + $\sqrt{DELTA}$ dividido por 2A
- A segunda raiz é (-B) - $\sqrt{DELTA}$ dividido por 2A.

No seu aspecto dinâmico, teríamos que gerar uma série de dados de entrada (triplas A,B,C), seguir o algoritmo, e verificar se os resultados que ele gera são aqueles esperados de antemão.

Dado um problema, para o qual temos que escrever um algoritmo, usualmente não há solução única. Quanto mais complexo o problema a resolver, maior a quantidade de possíveis algoritmos corretos. Cada um certamente terá as suas qualidades e os seus defeitos. Alguns são muito bons, e outros muito ruins, mas entre um e outro existem inúmeras variações, que nem sempre são facilmente reconhecidas.

Pode-se dizer que um algoritmo é um programa de computador do qual se faz a abstração da linguagem de programação. Neste sentido o algoritmo independe de qual será sua implementação posterior. Quando pensamos em algoritmos, não nos preocupamos com a linguagem. Estamos querendo resolver o aspecto "conteúdo". Mais tarde, quando programarmos, a linguagem passa a ser importante, e agora a preocupação é com a "forma".

1.6 Qualidades de um bom algoritmo

> "Se um programa é fácil de ler, ele é provavelmente um bom programa; se ele é difícil de ler, provavelmente ele não é bom."(Kernighan e Plauger)

Clareza O algoritmo é uma ferramenta de entendimento e solução de um problema. Deve portanto, ser o mais claro possível. Ao fazer o algoritmo seu autor deve se preocupar constantemente em se o que está pensando está visível no que está escrevendo.

> "Nunca sacrificar clareza por eficiência; nunca sacrificar clareza pela oportunidade de revelar sua inteligência."Jean Paul Tremblay.

Impessoalidade Nem sempre quem vai examinar, depurar, corrigir, alterar etc , um algoritmo é seu autor. Portanto, nada de usar macetes, regras pessoais, nomes que só tem sentido para o autor etc. Neste ponto é de muita ajuda a existência de um bem organizado conjunto de normas de codificação e de elaboração de algoritmos para a organização. As melhores e mais bem organizadas empresas de informática dedicam parcela significativa de seus esforços a esta atividade.

Simplicidade Poucos programadores resistem à tentação de elaborar uma genial saída para um problema. Nada contra isso, desde que essa saída não fira os princípios de simplicidade que todo algoritmo deve procurar satisfazer. Nem sempre é fácil fazer algoritmos descomplicados, principalmente porque em geral os problemas que enfrentamos não são simples. Transformar algo complicado em soluções singelas, parece ser a diferença entre programadores comuns e grandes programadores.

Reaproveitamento Existe inúmera bibliografia sobre algoritmos. Grandes instalações de informática costumam criar o conceito de "biblioteca de algoritmos". Muitos softwares também estão partindo para esta saída. O programador não deve se esquecer nunca de que um algoritmo novo, custa muito caro para ser feito, e custa muito mais caro ainda para ser depurado. Se já existir um pronto e testado, tanto melhor.

Capricho Tenha sempre em mente que a principal função de um algoritmo é transmitir a solução de um problema a outrem. Desta maneira, um mínimo de capricho é indispensável a fim de não assustar o leitor. Letra clara, identação correta, nomes bem atribuídos, tudo isto em papel limpo e sem borrões. Como o leitor já pode estar imaginando, esta característica é fundamental em provas, avaliações e trabalhos. A primeira consequência prática desta regra é: faça seus algoritmos a lápis.

Documentação Nem sempre o texto que descreve o algoritmo é muito claro. Além do que, ele informa o que é feito, mas não porque é feito, ou quando é feito. Assim, o programador pode e deve lançar mão de comentários sempre que sentir necessidade de clarificar um ponto.

> "Bons comentários não podem melhorar uma má codificação, mas maus comentários podem comprometer seriamente uma boa codificação". Jean Paul Tremblay.

Correção (ou integridade ou ainda robustez) Um algorítmo íntegro é aquele onde os cálculos estão satisfatoriamente colocados, e atuam corretamente para todos os dados possíveis de entrada. Não adianta um algoritmo sofisticado e cheio de estruturas de controle, se suas operações aritméticas elementares não estiverem corretas. É mais comum (e perigoso) o algoritmo que funciona em 95% dos casos, mas falha em 5% deles.

Eficiência Esta característica tem a ver com economia de processamento, de memória, de comandos, de variáveis etc. Infelizmente, quase sempre é uma característica oposta à da clareza e da simplicidade, sendo estas mais importantes do que a eficiência. Entretanto, em alguns casos, (principalmente na programação real - dentro das empresas), a eficiência pode ser fundamental. Neste caso, perde-se facilidade, para se ter economia. O bom programador é aquele que alia a simplicidade ao máximo de eficiência possível.

Generalidade Dentro do possível, um algoritmo deve ser o mais genérico que puder. Isto significa que vale a pena um esforço no sentido de deixar nossos algoritmos capazes de resolver não apenas o problema proposto, mas toda a classe de problemas semelhantes que aparecerem. Usualmente o esforço gasto para deixar um algoritmo genérico é pequeno em relação aos benefícios que se obtém desta preocupação.

Modularidade Algoritmos grandes dificilmente ficam bons se usarmos uma abordagem linear. É praticamente impossível garantir clareza, impessoalidade, documentação etc, quando se resolve um algoritmo de cima a baixo. Uma conveniente divisão em módulos (isto é, em sub-algoritmos), para que cada um possa ser resolvido a seu tempo. Não podemos esquecer que a regra áurea da programação estruturada é "dividir para conquistar".

O que um computador entende

Baseado na arquitetura de Von Neumann, (um processador, uma unidade de controle, uma ULA, memória e registradores), todo computador construído neste paradigma é capaz de obedecer, lá no nível mais baixo, a um pequeno conjunto de ordens. Note-se como ele é restrito.

Entrada/Saída Este tipo de ordem, é para orientar o computador no sentido de adquirir um dado externo (entrada), ou para divulgar alguma coisa que ele calculou (saída).

Aritmética Estas ordens são similares aquelas que aprendemos no primeiro grau, e destinam-se a informar ao computador como ele deve efetuar algum cálculo. Estas ordens são: adição, subtração, multiplicação, divisão, potenciação, funções trigonométricas etc. Não se esqueça que a linguagem universal dos computadores é a matemática. Escrever um programa de computador correto é como demonstrar um teorema.

Alternativa É talvez a ordem mais importante no universo da computação. É ela que dá ao computador condições de "ser inteligente" e decidir qual de dois caminhos deve se seguido. Sempre tem a forma de uma pergunta, seguida por duas ordens distintas. Se a pergunta for respondida afirmativamente, a primeira ordem será executada, e se a pergunta for negada, a segunda ordem será executada.

Desvio Esta ordem é para dar ao computador habilidade de saltar algumas ordens e executar outras. Um programa de computador nem sempre é obrigado a seguir as instruções sequencialmente começando no começo e terminando no fim. O desvio é potencialmente perigoso quando usado sem critério, e por isso veremos em seguida que uma das regras da programação estruturada é o estabelecimento de rígidos controles sobre os desvios.

Movimentação na memória Embora pareça estranho, esta é ordem mais comum em um programa. Considerando o tamanho das memórias de computador e a quantidade de coisas que são lá colocadas, é imprescindível que os programas tenham a habilidade de manusear todas estas coisas. Fazendo uma figura de linguagem, suponha que a memória do computador é um grande supermercado. Se os artigos não puderem se mexer de lugar, bem poucas coisas acontecerão.

Outra observação, é que o verbo "mover", quando usado em manipulações dentro da memória é usado por razões históricas, e certamente contém um equívoco. O correto seria usar o verbo "copiar", já que quando um conteúdo é levado de um lado a outro, sua instância anterior não desaparece. Usa-se aqui uma regra jocosa, mas que sempre é verdadeira. **computador não usa apagador**.

ciclo de vida

Qual a história de vida de um programa de computador ? Isto é como nasce, cresce e morre um programa ? Grosso modo, pode se dizer que as principais etapas são:

- estudo do problema: Espera-se que haja algum problema ou dificuldade esperando ser resolvido pelo programa de computador. É pré-requisito antes de tentar a solução de qualquer questão, estudá-la, compreender o entorno, as limitações, os balizadores, os recursos disponíveis. Uma boa maneira de estudar e compreender um problema é fazer hipóteses, estudar comportamentos, projetar simulações. Só se pode passar adiante se boa parte do entorno do problema for entendida. Note-se que nem sempre é possível aguardar até ter 100% do conhecimento antes de seguir adiante. Às vezes não dá tempo, mas o percalço principal nem é esse: é que muitas vezes o conhecimento não existe e só vai aparecer quando se cutucar o problema. Como se diz em espanhol *los caminos se hacen al caminar*. Quando isto ocorrer, e ocorre muitas vezes, não tem jeito: anda-se para frente e para trás, e a cada passada o conhecimento é maior.

- bolar ou procurar o algoritmo. Agora é hora de converter a solução encontrada em algo próximo a um algoritmo. Isto é, algo formal, com condições iniciais, resultado final, e manipulações entre o que entra e o que sai. Usam-se os comandos permitidos em algoritmos e só estes. Usualmente, há aqui o que se chama de aprofundamento interativo. Começa-se com uma declaração genérica que vai sendo detalhada e especificada até chegar ao nível básico, no qual os comandos algorítmicos (entrada/saída, aritméticos, alternativos, movimentação e de repetição) podem ser usados. Note que pode-se copiar ou aproveitar um algoritmo já existente. Entretanto este re-uso dificilmente é imediato. Sempre há que estudar e eventualmente modificar um algoritmo já existente, a menos que quando este foi feito seu autor já tinha a preocupação com o reuso.

- transcrever em LP. É hora de escolher um computador, um ambiente operacional e sobretudo uma linguagem de programação que esteja disponível e seja conhecida. Dessas 3 opções (computador, ambiente e linguagem) surge a plataforma definitiva: agora é possível escrever, testar, depurar e implementar o programa de computador.

- depurar. Usualmente há uma sequência de testes:

  - compilação correta: nenhum erro apresentado pelo compilador
  - testes imediatos sem erro: testes *ad hoc* para este programa (e só para ele) dão os resultados esperados.
  - teste alfa: um esforço completo de testes, inclusive integrando este programa aos seus antecessores e sucessores. Usualmente o teste é feito pelo(s) autor(es) do programa.
  - teste beta: um teste completo feito por terceiros. Há uma regra aqui de que quem faz (programador) não deve ser quem testa (testador).

- implementar e rodar. Com o programa testado e liberado é hora de rodá-lo em produção. Nesta hora ocorre o que tem sido chamado de *deploy* (dizem as más línguas que sempre associado a seu irmão gêmeo, o *delay*). Dependendo do porte da instalação e do sistema, às vezes esta passagem é complexa e trabalhosa.

- manutenção. Seria muito bom se a história acabasse na etapa anterior. Não é o que ocorre, já que o mundo muda e os programas precisam mudar com ele. Se a manutenção é pequena, usualmente pode-se fazê-la sem muito transtorno, mas caso não seja, este ciclo recomeça. Note-se que após algumas manutenções grandes o programa começa a dar mostrar de instabilidade –a popular colcha de retalhos – e é hora de começar tudo de novo, na nova versão do programa.

Como se aprende algoritmos?

Não é fácil aprender algoritmos. No que têm de engenharia, é mais fácil. Há fórmulas, procedimentos, práticas consagradas prontas a serem usadas. Mas, no que são arte, tais receitas desaparecem. É mais ou menos como tentar ensinar alguém a desenhar (outra arte) ou a compor música (outra ainda).

Olhando como se aprende a desenhar, é possível ter alguns *insights*. Aprendizes de desenho copiam desenhos e partes de desenho extensivamente. Pode-se tentar algo parecido aqui.

Em resumo, estas seriam as indicações

- Estudando algoritmos prontos. Pegue bons livros de algoritmos, e estude as implementações que lá aparecem. Transcreva tais algoritmos em um simulador (tipo visualg) ou mesmo em uma linguagem de programação. Execute os algoritmos. Busque entender o que cada parte do mesmo faz. Sugira hipóteses sobre o funcionamento e provoque pequenas mudanças no código para confirmar (ou rejeitar) suas hipóteses.

- Tenha intimidade com algum ambiente de compilação e testes. Esta proposta é importante para poder testar algoritmos. Se ficar embatucado diante de uma simples implementação, você terá grandes dificuldades de escrever algoritmos.

- Escreva pequenos algoritmos. Tente fazer os exercícios (começando sempre pelos mais simples) de qualquer bom livro de algoritmos.

- Treine a interpretação dos enunciados. Enquanto não entender o que deve ser feito, não adianta olhar para o algoritmo, pois o motivador das ações que ele toma estará ausente. Nunca se deve começar a estudar ou a construir um algoritmo se não houver um claro entendimento do que o algoritmo deve fazer.

1.7 Programação Estruturada

O termo programação estruturada veio à luz pela primeira vez, nos fins da década de 60, quando Edsger Dijkstra (Lê-se como *Dikster*) escreveu um artigo, publicado em *Communications of the ACM*, cujo título é "O comando GO TO é prejudicial". Vivíamos a época de ouro do COBOL, e o comando GO TO é a instrução de desvio desta linguagem. Dijkstra observou que a qualidade dos programadores decai em função do número de GO TOs usados em seus programas. Segundo ele, "Comandos GO TO tendem a criar caminhos de lógica confusos e programas pouco claros". A sua recomendação foi de que o comando em questão fosse banido das linguagens de alto nível.

Nessa época também (1966) dois professores italianos, Giuseppi Jacopini e Corrado Bohm, provaram matematicamente que qualquer lógica de programação poderia ser derivada de três tipos básicos de procedimentos, a saber: sequência, alternativa e repetição.

Chegou-se então ao âmago da programação estruturada, que é trocar a instrução de desvio pelas instruções de repetição. Ao fazer isto, todos os componentes de um programa passam a ter uma entrada e saída únicas. Atente-se que ao usar uma instrução de desvio, essa regra de uma entrada e uma saída é violada: o conjunto passa a ter no mínimo duas saídas.

Sequência simples Trata-se de um conjunto de comandos simples, que seguem um ao outro e que são executados na ordem em que aparecem. Exemplo

```
A ← 10
B ← A + 5
X ← 1 + 2
```

Dentro de uma sequência simples de comandos a entrada sempre é pelo primeiro comando e a saída sempre é pelo último. Graças ao princípio da programação estruturada (uma entrada e uma saída), uma sequência simples pode ser substituída por um bloco com entrada e saída únicas.

Alternativa ou comando condicional É um comando que permite dois caminhos alternativos, daí o nome, a depender de alguma condição. Na matemática convencional isto já existe, por exemplo na fórmula de Bhaskhara. No momento de calcular a raiz quadrada de ∇, há que se tomar uma decisão. Se $\nabla < 0$ os cálculos cessam e a resposta de raízes imaginárias deve ser dada. Se $\nabla \geq 0$, a raiz pode ser extraída e um ou dois valores de raízes emergem.

O ponto importante aqui é que em algum lugar os dois ramos se juntam novamente, e a regra de ouro da programação estruturada (entrada única e saída única também) continua verdadeira.

Tal como na sequência, um comando alternativo, por mais complexo que seja, pode ser substituído por única caixa.

Repetição O terceiro bloco da programação estruturada é a resposta à necessidade de usar o comando de desvio. Perceba-se que aqui existe um desvio implícito, já que ao se terminar o bloco, há que se desviar para o início do mesmo, que é o que caracteriza uma repetição. A restrição a obedecer não é a ausência de desvio – de resto, impossível de obedecer – mas sim a regra da entrada e saída únicas. Em outras palavras não é proibido usar o desvio, desde que ele esteja contido em limites bem determinados.

Tal como na sequência e na alternativa, um comando de repetição, por maior ou mais complexo que seja, pode ser substituído por única caixa.

A importância desta descoberta para o software teve tanto impacto quanto a de que qualquer forma lógica de hardware pode ser construida pelas combinações de portas AND, OR e NOT.

A grande vantagem dos três elementos da programação estruturada é que possuem a característica de pacote ou caixa preta, uma vez que todos tem uma única entrada e uma única saída. Isto significa que partes de programas construidos

com estas estruturas podem ser também considerados pacotes, sempre com entradas e saidas únicas. Esta é a melhor qualidade da programação estruturada, e é por esta característica que o GOTO deve ser desprezado na construção de bons algoritmos. Tanto isto é verdadeiro, que as linguagens mais modernas, não tem mais a instrução de desvio, banida por perniciosa e desnecessária.

1.8 A máquina de Turing

Embora a humanidade use o conceito de algoritmo há milênios (O de MDC é devido a Euclides), e o próprio nome algoritmo derive do autor árabe de um tratado de álgebra escrito por volta de 800 dC, a definição precisa do que seja um algoritmo geral é apenas de 1930. Ela é devida a Turing, através do seu conceito de Máquina de Turing. Esta não é uma máquina real, sendo apenas uma abstração matemática, daí o seu poder de encanto.

Em 1900, o matemático alemão Hilbert apresentou em um congresso uma lista de problemas que segundo ele *estariam ocupando as mentes matemáticas no século que ora se iniciava.* O décimo problema de Hilbert, dizia: **...haverá algum procedimento mecânico geral que possa em princípio resolver todos os problemas da matemática ?** [1]

Em 1937, Turing respondeu a esta pergunta através do conceito de Máquina de Turing. Ela é composta por um número finito de estados, ainda que possa ser muito grande. Deve tratar um input infinito. Tem um espaço exterior de armazenagem também infinito. Turing imaginou o espaço exterior para dados e armazenamento como sendo uma fita, que pode mover-se para frente e para trás. Além disso a máquina pode ler e escrever nesta fita. A fita está composta por quadrados (ou seja, o nosso ambiente é discreto). Apenas 2 símbolos podem ser escritos nos quadrados da fita, digamos 0 significando quadrado em branco e 1 (quadrado preenchido), ou vice-versa.

Isto posto, lembremos que:

1. os estados internos do aparelho são finitos em número

2. o comportamento da máquina é totalmente determinado pelo seu estado interior e pelo input.

Ou seja, dado o estado inicial e um símbolo de input, o aparelho deve ser determinístico [2]. Logo ele,

- muda seu estado interno para outro, e/ou
- Muda o simbolo que foi lido pelo mesmo ou por outro, e/ou
- movimenta-se para a direita ou para a esquerda, 1 quadrado, e/ou
- Decide se continua ou interrompe e para.

Definição de uma máquina de Turing Para especificarmos nossa máquina de Turing, teríamos que escrever algo do tipo

Estado atual	Sinal lido	faz o que	Vai para estado	Escreve	Anda para onde
0	0	→	0	0	D
0	1	→	13	1	E
1	0	→	65	1	D
1	1	→	1	0	D
2	0	→	0	1	D.PARE
2	1	→	66	1	E

Esta definição acima, é adequada para a tarefa de numeração das Máquinas de Turing, fenômeno esse que é central na teoria. Entretanto, para operar com esta máquina, talvez seja mais adequado escrever uma tabela como

estado	chega	escreve?	anda	vai para
0	0		D	0
0	1		E	13
1	0	1	D	65
1	1	0	D	1
2	0	1	PARE	
2	1		E	66

[1] Eis o problema original, como proposto por Hilbert: Determination of the solvability of a diophantine equation: Given a diophantine equation with any number of unkown quantities and with rational numerical coefficients: to devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers

[2] um fenômeno é deterministico quando gera sempre o mesmo resultado a partir das mesmas condições iniciais. Por exemplo, soltar um corpo em queda livre. Ao contrário um fenômeno probabilístico (ou randômico ou estocástico) é o fenômeno no qual dadas as mesmas condições iniciais o resultado é imprevisível. Por exemplo, lançar um dado

As diversas máquinas de Turing Para ver isso funcionando, precisa-se numerar as máquinas de Turing. Necessita-se um pouco de esperteza aqui. Primeiro, converte-se D, E, PARE, $\rightarrow$ e vírgula como os números 2, 3, 4, 5 e 6. Ou usando um sistema unário, 110, 1110, 11110, 111110 e 1111110. Zero será 0, e 1 será 10, como já se viu. As colunas 4 e 5 nas tabelas não precisam separação, já que o conteúdo da coluna 5 é apenas 1 ou zero e sempre ocupa 1 caracter.

Pode-se dar ao luxo de não codificar nenhuma seta, nem nada do que as antecede (colunas 1, 2 e 3) desde que se organize os comandos em rigorosa ordem crescente (e tomando cuidado para preencher comandos que no modo tradicional não seriam escritas por nunca ocorrerem na máquina).

Fazendo tudo isso, e escrevendo toda a programação da máquina que soma 1 a um número unário, ela fica:

```
10101101101001011010100111010010110101111010000111010010101 1
10100010111010100011010010110110101010101101010101101010100.
```

Convertendo este número binário para decimal chegamos a 450813704461563958982113775643437908 O que significa que a máquina que soma 1 em unário é a 450.813.704.461.563.958.982.113.775.643.437.908^a máquina de Turing. Naturalmente, mudando-se (otimizando-se) alguma coisa na programação, esta máquina muda de número.

A máquina que soma 1 em unário é a 177.642^a máquina. A que multiplica 2 em binário expandido é 10.389.728.107^a máquina e a que multiplica por 2 um número unário é a 1.492.923.420.919.872.026.917.547.669 ^{a} máquina.

E assim vai. A T_7, é não corretamente especificada, já que existem uma sequência de 5 uns, e ela emperra ao processar tal sequência.

Máquina Universal de Turing Teremos uma máquina U, que antes de mais nada lerá as informações na fita de entrada para que ela passe a se comportar como a máquina T. Depois disso vem os dados de input que seriam processados pela máquina T, mas que serão pela U, operando como se fosse a T.

Dizendo que quando a n-ésima máquina de Turing atua sobre o número m produz o número p, podemos escrever $T_n(m) = p$. Podemos pensar nesta relação como sendo uma função de 2 parâmetros (n e m) que leva a p. Esta função, pelo que vimos é totalmente algoritmica. Logo, ela pode ser realizada por uma máquina de Turing, a quem chamaremos de U. Então, fornecendo o par (n, m) a U, obtemos como resposta p.

Podemos escrever $U(n, m) = T_n(m)$

A máquina U quando alimentada primeiro com o número n, passa a se comportar como a T_n

Como U é uma máquina de Turing, ela é uma $T_{alguma-coisa}$. Quanto é essa alguma coisa ?

É mais ou menos 7.24×10^{1688} , ou seja o número 724 seguido de 1686 zeros.

Todos os computadores modernos, desde um humilde Z80 até um Cray multi-vetorial são máquinas de Turing.

Acabamos de ser apresentados a um moderno computador: em $U(n, m) = p$, U é o hardware, n é o software (ou programa), m são os dados de entrada e finalmente p é o resultado esperado.

Para efeito de entender a MT, pode-se desenhar algo como segue: Usar como modelo a MT que multiplica por 2 em unário

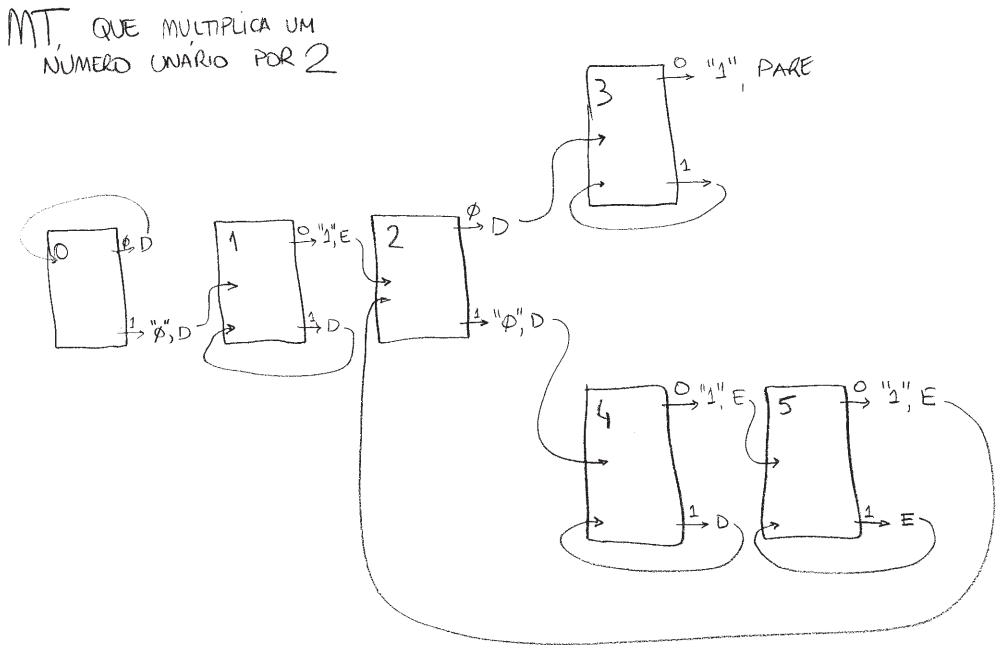


Figura 1.1: Kurt Göedel

1.9 Teorema da Incompletude de Göedel

Gödel nasceu em 18 de abril de 1906 em Brno atual Rep. Tcheca. Durante toda a sua vida escolar somente uma única vez Gödel deixou de ganhar a nota máxima (e logo em matemática). Foi batizado de criança como *der Herr Warum* (O senhor Por que ?). Apresentou sua tese de doutorado em Viena em 1929. Morreu em 14/01/78 de fome.

Em fins do século XIX, no Congresso de Matemática de Paris, em 1900, Hilbert, renomado professor em Göttingen apresentou 23 problemas, que segundo ele estariam ocupando os matemáticos do século XX. Seu segundo problema perguntava se é possível provar que os axiomas da aritmética são consistentes, isto é, dado um número finito de passos lógicos corretos, nunca se chegará a uma contradição. Na esteira desse problema, Bertran Russell, um filósofo matemático – ou será matemático filósofo ? – em 1910, começou uma série de livros denominados *Principia Mathematica*, na qual, baseado nos axiomas de Peano, desenvolvia todo um programa de formalização da matemática. A tentação era saborosa: provar que lógica e matemática eram a mesma coisa. Mas, o segundo problema de Hilbert continuava em aberto. Estudos posteriores de um jovem matemático austríaco, Kurt Gödel, emigrado para os EUA, que se consubstanciaram no teorema que leva o seu nome, deram o tiro de misericórdia na pretensão. Gödel provou que num sistema lógico formal existem assertivas verdadeiras que não podem ser provadas. Foi mais um sopetão na pretensão homocêntrica do Universo, assim como fizeram Copérnico, Darwin e Freud, só para ficar nos mais evidentes. Antes de passar ao Teorema de Gödel, diga-se apenas que dos 23 problemas, aproximadamente a metade ainda não tem solução (terá algum dia ?), e pior, ou melhor, a matemática se desenvolveu em centenas de direções nem sonhadas por Hilbert. Ainda bem que o trabalho final dele, terminava com a citação: *Enquanto uma ciência oferece abundância de problemas, ela está viva.*

Aliás, buscando inspiração no Congresso de 1900, na virada do século XX para o XXI, um americano rico, deixou 1 milhão de dólares para quem resolver por primeiro 7 problemas ainda pendentes na matemática: O problema P versus NP, além da hipótese de Riemann, a teoria de Yang-Mills (hipótese de lacuna de massa), as equações de Navier-Strokes, a conjectura de Poincaré, a conjectura de Birch, Swinnerton e Dyer e a conjectura Hodge) [3]

Demonstração do Teorema de Incompletude

Seu teorema se baseia em um processo de 3 etapas:

1. Estabelecer as regras pelas quais axiomas (e teoremas) podem gerar novos teoremas

2. Estabelecer os axiomas da aritmética em linguagem de lógica de predicados

3. Estabelecer uma forma de numeração para os teoremas assim gerados.

Eis os axiomas de Peano para os números naturais. Note-se que o símbolo s denota sucessor, um conceito primitivo. O sucessor de 0 é 1, de 1 é 2, ... de n é n+1:

[3] vide em www.claymath.org as regras do prêmio e a descrição formal deste e de mais 6 problemas:

$\forall x \sim (0 = sx)$	Não existe x tal que 0 seja seu sucessor
$\forall x, y(sx = sy) \Rightarrow (x = y)$	Se 2 números tem o mesmo sucessor, são iguais
$\forall x, x + 0 = x$	0 é o elemento neutro na adição
$\forall xy, x + sy = s(x + y)$	x somado com o sucessor de y é igual ao sucessor de x+y
$\forall x, yx \times sy = xy + x$	x=2, y=5 leva a $2 \times 6 = 2 \times 5 + 2 = 12$
$\forall x, x \times 0 = 0$	0 é o elemento absorvente na multiplicação
$\forall x, x = x$	identidade
$\forall xyz, (x = y) \Rightarrow ((x = z) \Rightarrow (y = z))$	propriedade transitiva
$\forall xy, (x = y) \Rightarrow (A(x, x) \Rightarrow A(x, y))$	A é qualquer fórmula com 2 variáveis livres
$(P(0) \wedge \forall xP(x) \Rightarrow P(sx))) \Rightarrow \forall xP(x)$	Regra fundamental da indução matemática

Com estas informações, Gödel, passou à etapa 3 do teorema, usando a tabela a seguir:

Símbolo	Código	Símbolo	Código	Símbolo	Código
0	1	(	6	$\sim$	11
s	2	)	7	$\wedge$	12
+	3	,	8	$\exists$	13
$\times$	4	x	9	$\forall$	14
=	5	1	10	$\Rightarrow$	15

Note-se que só existe a variável x. Logo quando só aparece x, ele será identificado como x_1. Quando aparecerem x e y, eles serão x_1 e x_{11}, e assim por diante.

Agora para cada cláusula, Gödel bolou um número, que mais tarde foi chamado "Número de Gödel", e que é construído da seguinte maneira. Tomam-se os primos a partir do 2. São eles: 2, 3, 5, 7, 11, 13, 17, 23, 29... Cada axioma terá seu número de Gödel calculado por um sistema de numeração onde as bases são os primos e os expoentes são o numero da tabela acima. Para clarear, vejamos o número de Gödel do axioma 4 de Peano:

$\forall x, y; x + sy = s(x + y)$ [Escrito em linguagem matemática]

$x_1 + sx_{11} = s(x_1 + x_{11})$ [Escrito na forma adequada para o teorema de Gödel],

e o número é:

$2^9.3^{10}.5^3.7^2.11^9.13^{10}.17^{10}.19^5.23^2.29^6.31^9.37^{10}.41^3.43^9.47^{10}.53^{10}.59^7$

Note-se que o numero é enorme, mas isso é de propósito. É para que, dado um número, SE ELE FOR UM NÚMERO DE GÖDEL, possa-se recuperar qual o axioma ou teorema que lhe deu origem. Basta fatorá-lo em seus componentes primos, e verificar qual o expoente de cada um dos primos. Ou seja, a relação axioma - número de Gödel é bi-unívoca. Estamos chegando perto da prova: Seja o predicado PROVA(x,y,z), lido como x é o número de Gödel da prova da fórmula Y (que tem número de Gödel y), o qual teve o inteiro z inserido dentro dela. Note-se que PROVA(x,y,z) é um jeito fácil de escrever uma imensa e longa expressão onde aparecem x1, x11 e x111, Essa expressão inclui muitos procedimentos, entre eles:

- dado um inteiro, fatore-o em seus fatores primos
- ache a expressão que lhe deu origem
- verifique se a expressão verifique se ela é uma fórmula
- verifique se ela esta provada

Claramente são procedimentos irrealizáveis na prática, mas em princípio, computáveis. Vamos considerar um caso especial do predicado acima: Suponha que a fórmula Y é alimentada com seu próprio número de Gödel, e vamos tentar negar a existência de tal prova. Escreve-se $\sim \exists\, xPROVA(x, y, y)$ Em palavras: x é o número de Gödel da prova obtida na fórmula Y pelo número y (Número de Gödel de Y). Vamos chamar ao número de Gödel da PROVA(x,y,y) de g. Finalmente: o TEOREMA DE GÖDEL $\sim \exists\, xPROVA(x, g, g)$ é verdadeiro, mas não é provável (no sentido de poder ser provado) no sistema aritmético formal.

A demonstração ocupa poucas linhas: Suponha que $\sim \exists\, xPROVA(x, g, g)$ é provável (no mesmo sentido) e seja p o número de Gödel dessa prova P. Então, nós temos que PROVA(p,g,g) é verdadeiro, desde que P é a prova de G, na qual foram substituídas as variáveis livres. MAS, a veracidade de PROVA(p,g,g) contradiz $\sim \exists\, xPROVA(x, g, g)$, e daqui temos que não existe essa prova. Então, se tudo foi construído corretamente, a afirmação $\sim \exists\, xPROVA(x, g, g)$ é verdadeira e portanto não existe a $PROVA(x, g, g)$ dentro do sistema. Em resumo: Dentro de um sistema de lógica formal existem teoremas que são indecidíveis.

O teorema de Goedel decide as duas primeiras questões de Hilbert: a matemática não é completa e e nem pode ser provada consistente, mas não resolve a terceira questão. Esta foi respondida negativamente de forma independente por dois cientistas: o primeiro denominado Alonso Churchs em abril de 1936 na universidade de Princeton ano nos Estados Unidos. Outro foi Alan Matheson Turing, em agosto de 1936 no Kings College em Londres. Um terceiro matemático, o polonês naturalizado americano Emil Post, professor do City College de New York submeteu em outubro de 1936 um artigo ao Jornal de Lógica Simbólica no qual expõe uma idéia similar à de Turing, embora menos ambiciosa.

Do trabalho de Church derivou-se a notação λ para definir funções, que está por trás da construção do interpretador LISP. O trabalho do Post deu origem ao PROLOG. O trabalho de Turing é simplesmente o projeto de um moderno computador digital.

Capítulo 2

Linguagens de programação

Humanos pensam em abstrato. Máquinas obedecem a comandos binários (bits ligados e desligados). A questão é: "como passar de um a outro ?". Nos primórdios da computação não havia jeito: era preciso converter do abstrato para o binário e quem tinha que fazer isso eram os humanos.

Era tarefa difícil: os códigos binários são enfadonhos e complexos. Além do que os níveis de detalhe são exasperantes. Claramente havia um gargalo aqui: semanas ou meses para escrever, depurar e implementar algum código mais sofisticado.

Cedo, nessa história, alguém se perguntou: porque não pôr os computadores para fazer essa delicada tradução (de idioma abstrato e humano para os códigos elétricos e binários exigidos pelo computador) ?

Nascia o conceito de linguagem de programação. Quando se fala em LP, está-se falando de 2 coisas interligadas: de um lado uma linguagem: suas regras de escrita, sua síntaxe e até alguma semântica. De outro lado há também um programa de computador (certamente escrito em OUTRA linguagem) que lê comandos escritos naquela síntaxe e semântica acima descritas e converte tais comandos em ordens que a máquina consiga obedecer. As vantagens de usar uma LP:

- É muito mais fácil aprender uma linguagem de programação do que aprender os códigos binários do computador. A diferença de tempo de aprendizado é de horas (LP) para meses (linguagem de máquina).

- Diferentes computadores (diferentes arquiteturas) podem usar o mesmo programa escrito em uma LP, desde que cada um tenha seu tradutor. É só olhar a lista de ambientes onde roda Python (mais de 10).

- Detalhes enfadonhos e repetitivos podem ser assumidos pelo programa tradutor liberando o humano desses detalhes. Por exemplo, a leitura de um registro em disco exige centenas de detalhes, mas que são sempre os mesmos. Assim, usando uma LP o programador pode comandar "leia o disco" e o tradutor se encarrega dos detalhes.

- É tentador escrever linguagens especializadas em algum domínio científico e/ou tecnológico. Mais e mais detalhes podem ser escondidos. É o que se faz em Matlab (engenharia), Tensorflow (redes neurais), Lisp (processamento simbólico), só para citar alguns.

- Usando este mecanismo, a humanidade passa a contar com uma ferramenta fantástica precisando investir pouco esforço. Pode-se fazer uma analogia com dirigir um automóvel. Ninguém precisa ser mecânico ou engenheiro para dirigir um carro. Qualquer um pode ser motorista.

Ao usar uma linguagem de programação, o seguinte ciclo precisa ser seguido

1. O problema é estudado, seus algoritmos e fórmulas traduzidos e um programa é escrito. A este programa, dá-se o nome de programa **fonte**.

2. O programa tradutor dessa linguagem, chamado **compilador** ou **interpretador** é carregado no computador e realiza a tradução.

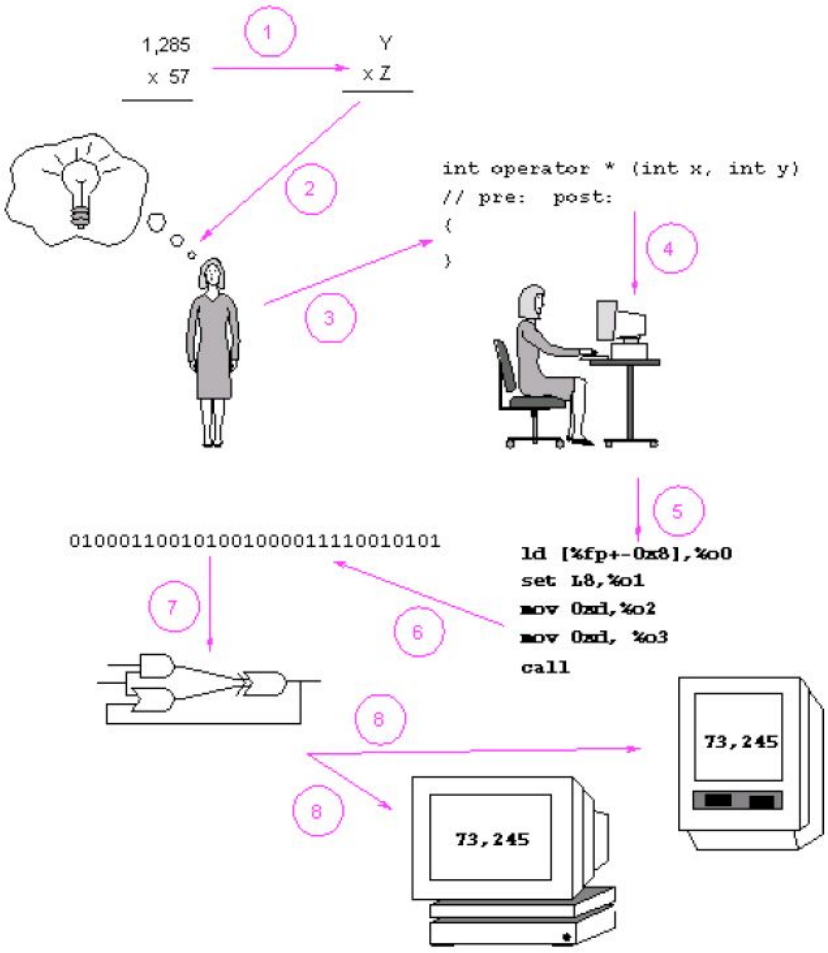
3. O resultado é chamada programa **objeto** e ele agora é versão executável do programa fonte.

4. O programador agora executa o programa objeto obtendo o resultado da computação.

5. Havendo erros ou divergências (sempre as há), corrige-se o programa fonte e o ciclo recomeça.

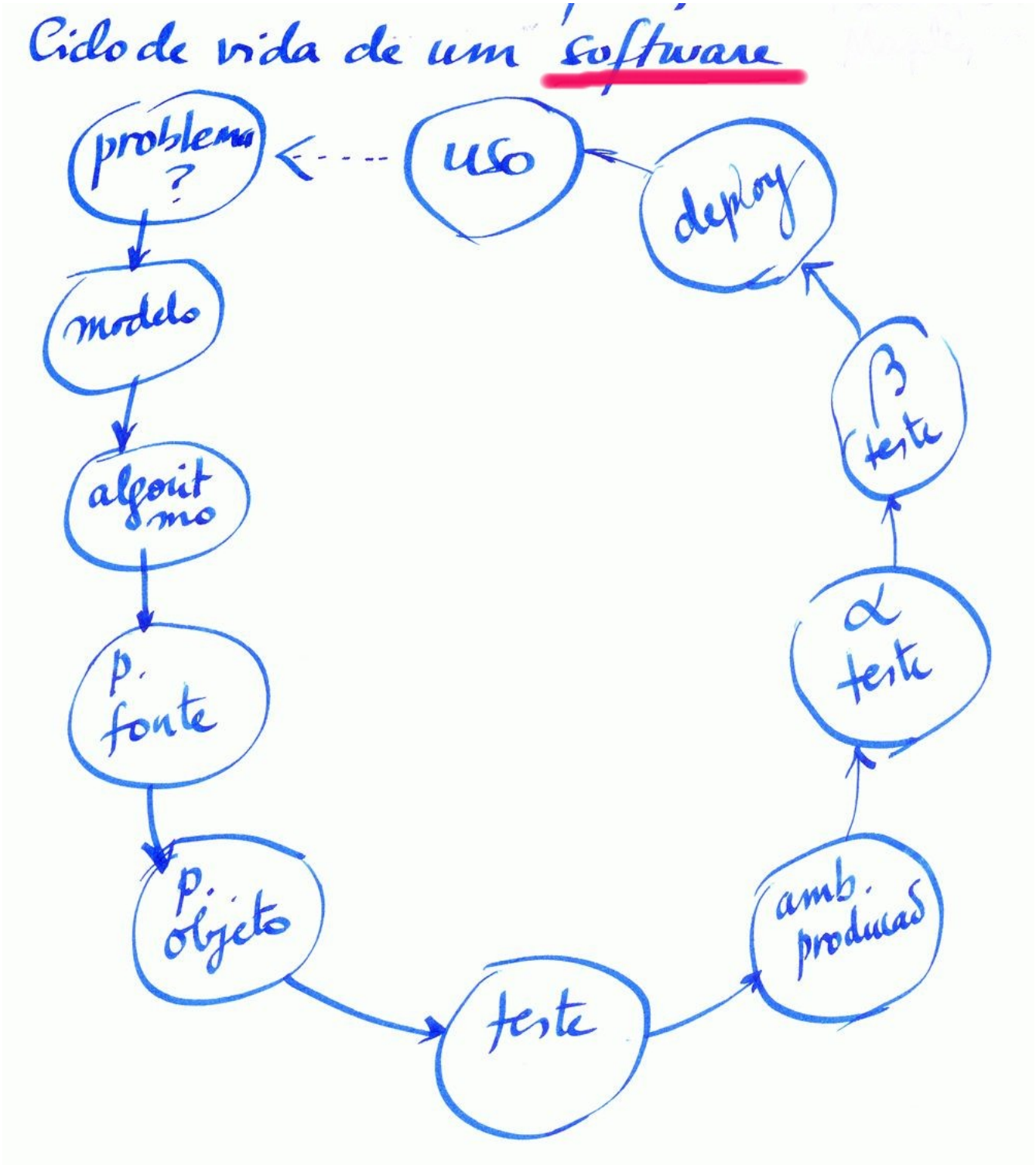
6. Quando o resultado obtido é certificado e julgado satisfatório o programa sofre um processo chamado *deployment* que significa sua entrada em regime normal de produção.

7. Quando as condições de entorno ou o problema se modificam, o ciclo recomeça.



Neste desenho uma tentativa de explicação do processo acima descrito.

Outra possível explicação gráfica para o ciclo de vida do software



Existem mais de 10.000 linguagens de programação, a grande maioria não são amplamente usadas. Só eu já criei 3 linguagens, é um exercício mental fantástico, você se sente como um criador de mundos. A seguir algumas classificações.

As linguagens de programação podem ser classificadas de acordo com diversas categorias, eis a seguir uma tentativa:

2.1 Categorias de LP

2.1.1 Quanto à geração

As gerações de software grosso modo acompanharam a idade da tecnologia. As linguagens de primeira geração nasceram junto com o computador e foram sendo desenvolvidas a partir dos desenvolvimentos teóricos na engenharia do software.

1^a geração	linguagem de máquina. No máximo, um montadores	ASSEMBLER
2^a geração	linguagens de especificação de problemas.	COBOL
3^a geração	linguagens com jargão do problema. Mais fáceis de aprender e usar.	ARENA

2.1.2 Quanto ao nivel

Indica quão próximo do hardware (e afastado do peopleware) está uma linguagem

baixo nivel	Específico de um hardware. Muito eficiente	ASSEMBLER
médio nivel	Mais fácil de aprender e usar. Menos eficiente	FORTRAN
alto nivel	Abstração da máquina. Muito fácil de usar	APL

2.1.3 Quanto à vocação

Aproveita uma certa maneira de pensar e resolver um certo tipo de problema e busca se adaptar a isso

matemática	Jargão da matemática	FORTRAN, MATHEMATICA
negócios	Boa documentação	COBOL
aprendizado	Ênfase na facilidade, rigor e documentação	PASCAL
inteligência artificial	processamento simbólico	LISP ou PROLOG
banco de dados	Implementa álgebra relacional	SQL
software básico	Ênfase na facilidade JUNTO com eficiência	C

2.1.4 Quanto à modalidade de uso

Trata a maneira como a linguagem é chamada e como são executados os programas escritos na linguagem:

compilada	Programas FONTE são traduzidos em linguagem OBJETO	diversas
interpretada	Programas são traduzidos e executados linha a linha	diversas
gerador de código	Programas que geram programas	diversos

2.1.5 Quanto ao paradigma

Implementam uma visão de como tratar a complexidade crescente

procedural	Lista de todas as ações que o programa deve fazer	diversas
funcional	Programas são funções (matemáticas)	LISP, APL, Haskell
lógicas	Programas são bases de inferência	PROLOG

2.1.6 Quanto a liberdade de escrita

Sugere regras e auto-controles para impedir que o programador cometa erros

liberdade total	O programador faz o que quer	diversas
Estruturada	Implementam-se o controle restrito do fluxo	diversas
Orientado a objetos	Encapsulamento e niveis crescentes de abstração	diversos

2.1.7 Quanto à idade

Depende do ano em que a linguagem viu a luz: há desde as antigas (Fortran, de 1954, COBOL de 1958, LISP de 1955) às mais modernas (Python de 1995, Lua meados dos anos 90). A diferença aqui podem ser sistematizada nos seguintes pontos:

- As antigas tem mais idiossincrasias, já que nascem em ambientes ainda pequenos sem massa crítica e com pouca experiência. Escrever linguagens era adentrar a um mundo novo.
- As antigas tem uma obsessão com a economia de recursos. Em 1974 aprendi a programar em um computador IBM1130 que tinha 8KB de memória. De qualquer maneira continua sendo boa idéia economizar, mas não há mais porque gastar horas para economizar 3 bytes, por exemplo.
- As modernas agregam inúmeros recursos advindos (copiados) de ambientes bem sucedidos. Esta tendência levada ao exagero gera linguagens que parecem Franksteins, mas há quem goste.
- Preocupação com usabilidade: esta é uma questão recente na ciência da computação, mas é bem importante. Descobriu-se que além de se adaptar bem ao computador, uma LP deve se adaptar bem também ao cinzento.

2.2 Assembler

O assembler não é uma linguagem de alto nivel. É apenas um tradutor de códigos e portanto está muito próximo da linguagem de máquina. Utilizá-lo é um autêntico calvário, já que todas as operações que o computador precisa realizar ficam por conta do programador. A produtividade é muito baixa, e a única vantagem que se consegue é a eficiência: os programas voam.

Por outro lado, tamanha aderência às entranhas da máquina cobra seu preço: na mudança de plataforma tecnológica um monte de coisas precisa ser reestudado e reciclado.

Não é uma boa idéia programar profissionalmente em assembler. Mas, quem domina esta técnica, é bastante valorizado no mercado. Mesmo que trabalhando em outras linguagens, quem conhece o assembler, conhece o interior da máquina. Sem dúvida alguma, isto tem o seu valor. A seguir um trecho pequeno de um programa em assembler para plataforma Intel.

```
 L100: MOV AH,2AH ;pega a data
INT 21H
PUSH AX ;DEPOIS VOU DAR POR EXTENSO
MOV BX,10
MOV DI,OFFSET DIA
MOV AH,0
MOV AL,DL
DIV BL
OR AX,3030H
STOSW
MOV AL," "
STOSB
MOV AH,0
MOV AL,DH
PUSH AX ;DEPOIS VOU DAR POR EXTENSO
DIV BL
OR AX,3030H
STOSW
MOV AL," "
STOSB
```

2.3 Fortran

Fortran é uma linguagem muito antiga, originalmente sugerida pela IBM na década de 50. Seu texto se assemelha a uma descrição matemática e de fato, FORTRAN significa FORmula TRANslation. É, ou pelo menos era, uma linguagem franciscana de tão pobre em recursos. Foi a única linguagem ensinada em escolas de engenharia até a década de 80. O primeiro compilador de FORTRAN foi desenvolvido para o IBM 704 em 1954-57 por uma equipe da IBM chefiada por John W. Backus.

A inclusão de um tipo de dados de número complexo na linguagem tornou a linguagem Fortran particularmente apta para a computação científica.

As principais revisões são a Fortran 66, 77, 90 e 95. A mais famosa e usada é a Fortran 90. Sua principal vantagem é a eficiência em calculo numérico pesado.

A seguir, um exemplo de FORTRAN note-se que já é um Fortran revisto (e moderno): basta olhar o formato do comando IF (antes era muito mais idiossincrático e obscuro).

```
C        1         2         3         4         5         6
C234567890123456789012345678901234567890123456789012345678901234567890123456789012345
      PROGRAM BASKHARA
C
      REAL  A,B,C, DELTA, X1,X2, RE, IM
C
      PRINT *, "Este programa resolve uma equação de 2o.grau"
      PRINT *, "do tipo: a*x**2 + b*x + c = 0"
C
      PRINT  10, "Digite a,b,c: "
 10   FORMAT(A,1X,$)
 20   READ(*,*,ERR=20) A,B,C
C
      DELTA=B*B-4.*A*C
C
```



```
      IF (DELTA.GT.0) THEN      ! (DUAS RAIZES REAIS)
         X1=(-B-SQRT(DELTA))/(2.*A)
         X2=(-B+SQRT(DELTA))/(2.*A)
         PRINT *, "RAIZES:  X1=",X1
         PRINT *, "         X2=",X2
      ELSE IF (DELTA.EQ.0) THEN ! (DUAS RAIZES REAIS IGUAIS)
         X1=-B/(2.*A)
         X2=X1
         PRINT *, "RAIZES: X1=X2=",X1
      ELSE                      ! (DUAS RAIZES COMPLEXAS)
         RE=-B/(2.*A)
         IM=SQRT(-DELTA)/(2.*A)
         PRINT *, "RAIZES COMPLEXAS: X1=",RE," -",IM,"i"
         PRINT *, "                  X2=",RE," +",IM,"i"
      ENDIF
C
      END
```

2.4 Lisp

LISP é a segunda linguagem de alto nivel mais antiga. O LISP nasce como filho da comunidade de Inteligência Artificial. Na origem, 4 grupos de pessoas se inseriram na comunidade de IA: os linguistas (na busca de um tradutor universal), os psicólogos (na tentativa de entender e estudar processos cerebrais humanos), matemáticos (a mecanização de aspectos da matemática, por exemplo a demonstração de teoremas) e os informáticos (que buscavam expandir os limites da ciência).

O marco inicial é o encontro no Dartmouth College em 1956, que trouxe duas consequências importantes: a atribuição do nome IA e a possibilidade dos diferentes pesquisadores se conhecerem e terem suas pesquisas beneficiadas por uma interfecundação intelectual.

A primeira constatação foi a de que linguagens existentes privilegiavam dados numéricos na forma de arrays. Assim, a busca foi a criação de ambientes que processassem símbolos na forma de listas. A primeira proposta foi IPL (Information Processing Language I, por Newel e Simon em 1956). Era um assemblão com suporte a listas. A IBM engajou-se no esforço, mas tendo gasto muito no projeto FORTRAN decidiu agregar-lhe capacidade de listas, ao invés de criar nova linguagem. Nasceu a FLPL (Fortran List Processing Language).

Em 1958, McCarthy começou a pesquisar requisitos para uma linguagem simbólica. Foram eles: recursão, expressões condicionais, alocação dinâmica de listas, desalocação automática. O FORTRAN não tinha a menor possibilidade de atendê-las e assim McCarthy junto com M. Minsky desenvolveu o LISP. Buscava-se uma função Lisp Universal (como uma máquina de Turing Universal). A primeira versão (chamada LISP pura) era completamente funcional. Mais tarde, LISP foi sofrendo modificações e melhoramentos visando eliminar ineficiências e dificuldades de uso. Acabou por se tornar uma linguagem 100% adequada a qualquer tipo de resolução de problema. Por exemplo, o editor EMACS que é um padrão no mundo UNIX está feito em LISP.

Inúmeros dialetos LISP apareceram, cada um queria apresentar a sua versão como sendo a melhor. Por exemplo, FranzLisp, ZetaLisp, LeLisp, MacLisp, Interlisp, Scheme, T, Nil, Xlisp, Autolisp etc. Finalmente, como maneira de facilitar a comunicação entre todos estes (e outros) ambientes, trabalhou-se na criação de um padrão que foi denominado Common Lisp. Como resultado o Common Lisp ficou grande e um tanto quanto heterogêneo, mas ele herda as melhores práticas de cada um de seus antecessores.

Seja agora um exemplo de uma função lisp:

```
> (defun li (L1 L2)   ; pergunta se duas listas são iguais
    (cond
       ((null L1) (null L2))
       ((null L2) nil)
       ((not (eql (car L1) (car L2))) nil)
       (t (li (cdr L1) (cdr L2)))))
LI
```

2.5 Prolog

O nome Prolog para a linguagem concreta foi escolhido por Philippe Roussel como uma abreviação de *PROgrammation LOGique*. Foi criada em meados de 1972 por Alain Colmerauer e Philippe Roussel, baseados no conceito de Robert Kowalski da interpretação procedimental das cláusulas de Horn. A motivação para isso veio em parte da vontade de reconciliar o uso da lógica como uma linguagem declarativa de representação do conhecimento com a representação procedimental do conhecimento.

A seguir um histórico de Prolog:

- Precursores: Newell, Shaw e Simon, com sua Logic Theory Machine, que buscava a prova automática de teoremas, em 1956.
- Robinson, 65, no artigo *A machine oriented logic based on the resolution principle*. Propunha o uso da fórmula clausal, do uso de resolução e principalmente do algoritmo de unificação.
- Green em 1969, escreveu o *Question Answer System*, que respondia perguntas sobre um dado domínio. Na mesma época, Winograd escreveu o software Planner.
- Todos estes tinham problemas de eficiência, por causa, entre outras coisas, da explosão combinatória.
- Em 1970, em Edinbourg, Kowalski bolou o seguinte esquema:
  - Limitou a expressividade às cláusulas de Horn
  - Criou uma estratégia de resolução, não completa, mas muito eficiente
- Em 72, em Edinbourg e Marselha, surgiu a linguagem PROLOG.
- Em 77, Waren escreve o primeiro compilador de PROLOG. Começa o sucesso de PROLOG na Europa. Nos EUA, sempre deu-se maior atenção ao LISP.
- Foi a base para o computador de 5^a geração do Japão.

A seguir, um pequeno trecho de uma sessão PROLOG:

```
parent(maria,jorge):-true.
parent(joao,jorge):-true.
parent(jorge,ana):-true.
...
parent(cris,paulo):-true.
```

Daí: podemos perguntar:

```
? parent(joao,jorge)
YES
? parent(celia,cris)
NO
? parent(cris,carlos)
NO
? parent(X,jorge)
X=maria
X=joao
no (significando acabou)
? parent(X,Y)
X=maria
Y=jorge
X=joao
Y=jorge
...
? parent(X,Y),parent(Y,paulo)  [ler esta vírgula como OU]
X=jorge
Y=cris
```

2.6 Cobol

O nome vem da sigla de COmmon Business Oriented Language (Linguagem Comum Orientada aos Negócios), que define seu objetivo principal em sistemas comerciais, financeiros e administrativos para empresas e governos.

O COBOL foi criado em 1959 pelo Comitê de Curto Prazo, um dos três comitês propostos numa reunião no Pentágono em Maio de 1959, organizado por Charles Phillips do Departamento de Defesa dos Estados Unidos. O Comitê de Curto Prazo foi formado para recomendar as diretrizes de uma linguagem para negócios. Foi constituído por membros representantes de seis fabricantes de computadores e três órgãos governamentais, a saber: Burroughs Corporation, IBM, Minneapolis-Honeywell (Honeywell Labs), RCA, Sperry Rand, e Sylvania Electric Products, e a Força Aérea dos Estados Unidos, o David Taylor Model Basin e a Agência Nacional de Padrões (National Bureau of Standards ou NBS). Este comitê foi presidido por um membro do NBS. Um comitê de Médio Prazo e outro de Longo Prazo foram também propostos na reunião do Pentágono. Entretanto, embora tenha sido formado, o Comitê de Médio Prazo nunca chegou a funcionar;

e o Comitê de Longo Prazo nem chegou a ser formado. Por fim, um subcomitê do Comitê de Curto Prazo desenvolveu as especificações da linguagem COBOL.

As especificações foram aprovadas pelo Comitê de Curto Prazo. A partir daí foram aprovadas pelo Comitê Executivo em Janeiro de 1960, e enviadas à gráfica do governo, que as editou e imprimiu com o nome de Cobol 60. O COBOL foi desenvolvido num período de seis meses, e continua ainda em uso depois de mais de 40 anos.

O COBOL foi definido na especificação original, possuia excelentes capacidades de autodocumentação, bons métodos de manuseio de arquivos, e excepcional modelagem de dados para a época, graças ao uso da cláusula PICTURE para especificações detalhadas de campos. Entretanto, segundo os padrões modernos de definição de linguagens de programação, tinha sérias deficiências, notadamente sintaxe prolixa e falta de suporte da variáveis locais, recorrência, alocação dinâmica de memória e programação estruturada. A falta de suporte à linguagem orientada a objeto é compreensível, já que o conceito era desconhecido naquela época. Segue o esqueleto de um programa fonte COBOL.

```
 IDENTIFICATION DIVISION.
         PROGRAM-ID. HELLO-WORLD.
       *
        ENVIRONMENT DIVISION.
       *
        DATA DIVISION.
       *
        PROCEDURE DIVISION.
        PARA-1.
            DISPLAY "Alô, Mundo.".
       *
            STOP RUN.
```

2.7 APL

Ela nasceu do trabalho de um professor de matemática canadense de nome Keneth Iverson. Sua proposta original era a de produzir uma nova notação matemática, menos sujeita às ambiguidades da notação convencional.

Na década de 60, trabalhando na IBM em conjunto com Adin Falcoff, ambos produziram a primeira versão de APL, quando um interpretador da linguagem ficou disponível.

A principal característica de APL é o uso de um conjunto especial de caracteres que incluem algumas letras gregas (rho, iota...), símbolos matemáticos convencionais (o sinal de vezes, o de dividido...) e alguns símbolos especialmente inventados.

Este fato sempre limitou a disseminação da linguagem. Até o advento das interfaces gráficas, como o windows, por exemplo, exigia-se um hardware especial para poder programar em APL.

Programas em APL em geral sempre são muito pequenos, embora poderosos. A linguagem está preparada para tratar arranjos de grandes dimensões. Por exemplo, quando em APL se escreve A+B, se A e B forem escalares (isto é um número único), a resposta também o será. Se A e B são vetores de 100 números, a resposta também o será. Idem para matrizes e até arrays-nd. Em algumas versões de APL este n chega a 256 dimensões.

Este é um comando EDIT (troca algo, de... para...)

```
      ∇edit[⎕]∇
    ∇
[0]   r←depara edit x
[1]   r←,x
[2]   r[(r=1↑depara)/⍳⍴r]←1↓depara
[3]   r←(⍴x)⍴r
    ∇ 2018-06-24 10.03.49 (GMT-4)
       x←3 3⍴?9⍴4
       x
2 1 3
1 3 3
2 2 2
       2 99 edit x
99  1  3
 1  3  3
99 99 99
```

2.8 Basic

A linguagem BASIC (acrônimo para Beginners All-purpose Symbolic Instruction Code), foi criada, com fins didáticos, pelos professores John G. Kemeny e T. Kurtz em 1963 no Dartmouth College.

BASIC também é o nome genérico dado a uma grande família de linguagens de programação derivadas do Basic original. Provavelmente existem mais variações de Basic do que de qualquer outra linguagem de programação.

Basic é uma linguagem imperativa de alto nível, pertencente à terceira geração, que é normalmente interpretada e, originalmente, não estruturada, por ter sido fortemente baseada em FORTRAN II.

Um programa em Basic tradicional tem suas linhas numeradas, sendo que é quase que padrão usar números de 10 em 10 (o que facilita a colocação de linhas intermediárias). Os comandos são poucos, simples e facilmente compreensíveis na língua inglesa (LET, IF, ...). Um programa em Basic, que calcula a fórmula de Bhaskara ficaria como:

```
10  REM RESOLVE EQUACAO DO SEGUNDO GRAU
20  READ A,B,C
25  IF A=0 THEN GOTO 410
30  LET  D=B*B-4*A*C
40  IF D<0 THEN GOTO 430
50  PRINT "SOLUCAO"
60  IF D=0 THEN GOTO 100
70  PRINT "PRIMEIRA SOLUCAO",(-B+SQR(D))/(2*A)
80  PRINT "SEGUNDA SOLUCAO",(-B-SQR(D))/(2*A)
90  GOTO 20
100 PRINT "SOLUCAO UNICA",(-B)/(2*A)
200 GOTO 20
410 PRINT "A DEVE SER DIFERENTE DE ZERO"
420 GOTO 20
430 PRINT "NAO HA SOLUCOES REAIS"
440 GOTO 20
490 DATA 10,20,1241,123,22,-1
500 END
```

O programa demonstra a falta de estruturação da linguagem original, pois o IF funciona como um GOTO condicional, o que favorece o código espaguete.

2.9 Clipper

Tudo começou com o dbase, que foi o precursor de bancos de dados para micros. Por volta de 83, 84 surgiu o dbase 2 (nunca houve o 1) para micros de 8 bits com memórias típicas de 64Kb e um ou dois disquetes de 8 polegadas e cerca de 512 KB de capacidade em cada um.

O dbase já era um gerenciador de dados, dentro da visão relacional, com mínima redundância de dados e com todas as garantias de acesso e uma linguagem estruturada, no melhor padrão, de uso geral, capaz de garantir um aumento na produtividade na programação.

A história do dBASE começa em 74, no Jet Propulsion Laboratory, conhecido como JPL, e instalado em Pasadena, Califórnia.

Nesta instalação da NASA trabalhava Jeb Long, pesquisador que lidava com dados obtidos nas pesquisas espaciais através de naves não tripuladas. Estes, por serem em grande número, começaram a trazer problemas e dissabores a Jeb Long, que desenvolveu, então, um programa de computador capaz de gerenciar tais informações. Este programa com nome de JPLDIS, foi concluído e alcançou razoável sucesso tendo sido bastante comentado pelo pessoal técnico. Foi o que bastou para que um analista de sistemas (Wayne Ratliff) começasse a desenvolver um programa gerenciador de dados, genérico, mas tendo como modelo o JPLDIS.

Rapidamente o novo trabalho tornou-se melhor do que o original, e Ratliff desenvolveu-o cada vez mais.

Em 79, Ratliff achou que o programa já estava maduro e pronto para enfrentar o mercado, e ele foi anunciado com o nome de VULCAN. Foi, entretanto, um fracasso: não se chegou a vender sequer 50 cópias.

Alguns meses depois, um grande distribuidor de programas de micros, George Tate, tomou conhecimento do dBASE. Testou-o e ficou entusiasmado.

Achou que o que faltava a Ratliff era suporte comercial, uma vez que seu trabalho era muito bom. Rapidamente providenciaram-se algumas mudanças cosméticas no programa, seu nome foi mudado para dBASE II (nunca houve o dBASE I, tratava-se de estratégia comercial, para apresentar o produto como um melhoramento), e uma nova empresa (ASHTON TATE) foi criada para vender este produto.

Em pouco tempo, dBASE tornou-se um marco na história de softwares para microcomputadores.

A grande vantagem de um banco de dados é retirar a amarração entre programas e dados.

A explicação desta vantagem é que dados são muito mais estáveis do que os procedimentos (=programas). Separando-os garante-se que longevidade aos dados, que em geral são os mais custosos para adquirir.

O dbase como linguagem era interpretado, o que trazia desempenho pífio nas aplicações, além da eventual falta de segurança (qualquer um podia alterar os dados). Para corrigir estas duas deficiências, logo surgiram inúmeros compiladores, dos quais o mais famoso foi o CLIPPER.

Eis um programa em CLIPPER, por acaso um trecho do programa que emite provas aleatórias e diferentes:

```
w_nomq = space(8)
w_file = "LIXO    "
@ 08,45 say "Nome Arquivo: " get w_file pict "@!"
@ 09,5 say "Qual a turma ? " get w_turm pict "@!"
@ 09,40 say "OU arquivo de nomes " get w_nomq pict "@!"
oba := 1
do while oba == 1
   oba := 0
   W_mess := "    "
   @ 11,15 say "Questao 1:"
   @ 12,20 say "Topico: " get w_top1
   @ 12,50 say "Grupo: " get w_gru1 pict "99"
```

2.10 Natural

NATURAL é uma linguagem de quarta geração que pode ser utilizada, indistintamente por usuários finais trabalhando em auto-serviço, ou por programadores profissionais em suas atividades normais no CPD.

Embora possa ler arquivos sequenciais, o Natural é mais indicado para ler bancos de dados ADABAS, que a partir da década de 90 foi o padrão de bancos de dados em nosso país e em boa parte do mundo.

Eis um exemplo de programa em Natural:

```
0010 AT TOP OF PAGE
0020    DO
0030       WRITE 20T "INFORMACOES SALARIAIS PARA SANTA  CATARINA"
0040       SKIP 1
0050    DOEND
0060 FIND PESSOAL WITH ESTADO = "SC"
0070    SORTED BY CIDADE
0080 AT BREAK OF CIDADE
0090    DISPLAY  NOTITLE  "NOME DA/CIDADE" OLD(CIDADE)
0100             "TOTAL/SALARIOS" SUM(SALARIO) (EM=ZZZ,ZZZ,ZZZ)
0110             "SALARIO/MEDIO"  AVER(SALARIO)
0120             "SALARIO/MAXIMO" MAX(SALARIO)
0130             "NUMERO/PESSOAS" COUNT(CIDADE) (EM=ZZ99)
0140 AT END OF DATA
0150    DO
0160       SKIP 1
0170       WRITE "TOTAL DE TODOS OS SALARIOS" TOTAL(SALARIO)
0180    DOEND
0190 END
```

2.11 Pascal

É uma linguagem de programação estruturada que recebeu este nome em homenagem ao matemático Blaise Pascal. Foi criada em 1970 pelo suíço Niklaus Wirth, tendo em mente encorajar o uso de código estruturado.

Segundo o autor, Pascal foi criada simultaneamente para ensinar programação estruturada e para ser utilizada em sua fábrica de software. A linguagem reflete a liberação pessoal de Wirth após seu envolvimento com a especificação de ALGOL 68, e sua sugestão para essa especificação, o ALGOL W.

A linguagem é extremamente bem estruturada e muito adequada para ensino de linguagens de programação. É provavelmente uma das linguagens mais bem resolvidas entre as linguagens estruturadas, e certamente um dos exemplos de como uma linguagem especificada por uma pessoa pode ser bem melhor do que uma linguagem especificada por um comitê.

Pascal originou uma enorme gama de dialetos, podendo também ser considerada uma família de linguagens de programação. Grande parte de seu sucesso se deve a criação, na década de 80, da linguagem Turbo Pascal, inicialmente disponível para computadores baseados na na arquitetura 8086 (com versões para 8080 no seu início).

Pascal é normalmente uma das linguagens de escolha para ensinar programação, junto com Scheme, C e Fortran. Só mais recentemente o Java entrou nesta lista.

Comercialmente, a linguagem foi sucedida pela criação da linguagem Object Pascal, atualmente utilizada nas IDEs Borland Delphi, Kylix e Lazarus. Academicamente, seus sucessores são as linguagens subsequentes de Niklaus Wirth: Modula-2 e Oberon.

A partir da versão 2005, o Delphi passou a se referir a sua linguagem de programação como Delphi Language.

Assim como a Linguagem C, que é padronizado pela ANSI (Ansi C), o Pascal possui padrões pela ISO, como o Pascal Standard e o Advanced Pascal. Um exemplo:

```
program Teste;
var a,b:integer;
uses crt;
begin
   writeln ('Digite um número para A');
   readln (a);
   writeln ('Digite o número para B');
   readln (b);
   if (a > b) then     { Se A é maior que B então }
       writeln ('A é maior que B')
   else                { Senão... }
       writeln ('B é maior que A');
end
```

2.12 C

C é uma linguagem de programação estruturada e padronizada criada na década de 1970 por Dennis Ritchie e Ken Thompson para ser usada no sistema operacional UNIX. Desde então espalhou-se por muitos outros sistemas operacionais, e tornou-se uma das linguagens de programação mais usadas.

C tem como ponto-forte a sua eficiência e é a linguagem de programação de preferência para o desenvolvimento de software básico, apesar de também ser usada para desenvolver aplicações. É também muito usada no ensino de ciências da computação, mesmo não tendo sido projetada para estudantes e apresentando algumas dificuldades no seu uso. Outra característica importante de C é sua proximidade com a linguagem de máquina, que permite que um projetista seja capaz de fazer algumas previsões de como o software irá se comportar ao ser executado.

C tem como ponto fraco a falta de proteção que dá ao programador. Praticamente tudo que se expressa em um programa em C pode ser executado, como por exemplo pedir o vigésimo membro de um vetor com apenas dez membros. Os resultados muitas vezes totalmente inesperados e os erros são difíceis de encontrar.

Muitas linguagens de programação foram influenciadas por C, sendo que a mais utilizada atualmente é C++, que por sua vez foi uma das inspirações para Java. O exemplo:

```
#include <stdio.h>
struct pessoa
{
        unsigned short idade;
        char nome[51];        /*vector de 51 chars para o nome*/
        unsigned long bi;
}; /*estrutura declarada*/
int main(void) {
        struct pessoa Sousa={16,"Diogo Sousa",123456789};
                                    /*declaracao de uma variavel tipo struct
                                            pessoa com o nome Sousa*/
        printf("Idade: %d\n",(int)Sousa.idade);  /* "%d" espera um int */
        printf("Nome: %s\n",Sousa.nome);
        printf("BI: %lu\n",Sousa.bi);
        return 0;
}
```

2.13 C++

C++ é uma extensão da linguagem C, corrigindo algumas de suas deficências originais (como por exemplo, a incapacidade de C de generalizar o tipo de dado em uma estrutura particular), além de incluir a manipulação de objetos e a simplificação de algumas tarefas (como entrada/saída padrão, por exemplo). Mas, C++ se saiu muito bem por herdar uma compatibilidade com o legado. (Um programa fonte C já é um programa fonte C++). Essa idéia transformou o conjunto C/C++ no grande campeão de uso no mundo da informática. O TIOBE (www.tiobe.com) de junho de 2018 apresenta os seguintes valores de uso no mundo: java com 15.5%, C com 14.9%, c++ com 8.3% e python com 5.7%. Se somarmos c com c++ o resultado é 23.2%, e esse conjunto se sagra campeão da lista. A seguir, o mesmo programa feito em C e em C++. Compare e tire suas conclusões

```
#include<stdio.h>
int main(){
```

```
   long p,q,r;
   while (scanf("%ld %ld", &p, &q) != EOF){
      if(q>p) r=q-p;
      else r=p-q.
      printf("%ld\n",r);
   }
}
```

Agora a mesma coisa em C++

```
#include<iostream>
void main(){
   long long a,b,c;
   while (cin>>a>>b){
      if(b>a)
     c=b-a;
      else
     c=a-b.
      cout<<c<<endl;
   }
}
```

2.14 Java

Java é uma linguagem de programação orientada a objeto desenvolvida na década de 90 pelo programador James Gosling, na empresa Sun Microsystems. Diferentemente das linguagens convencionais, que são compiladas para código nativo, a linguagem Java é compilada para um ”bytecode” que é executado por uma máquina virtual.

Desde seu lançamento, em maio de 1995, a plataforma Java foi adotada mais rapidamente do que qualquer outra linguagem de programação na história da computação. Em 2003 Java atingiu a marca de 4 milhões de desenvolvedores em todo mundo. Java continuou crescendo e hoje é uma referência no mercado de desenvolvimento de software. Java tornou-se popular pelo seu uso na Internet e hoje possui seu ambiente de execução presente em web browsers, mainframes, SOs, celulares, palmtops e cartões inteligentes, entre outros.

Principais Características da Linguagem Java

A linguagem Java foi projetada tendo em vista os seguintes objetivos:
Orientação a objeto - Baseado no modelo de Smalltalk e Simula67
Portabilidade - Independência de plataforma - “write once run anywhere”
Recursos de Rede - Possui extensa biblioteca de rotinas que facilitam a cooperação com protocolos TCP/IP, como HTTP e FTP
Segurança - Pode executar programas via rede com restrições de execução
Sintaxe similar a Linguagem C/C++
Facilidades de Internacionalização - Suporta nativamente caracteres Unicode
Simplicidade na especificação, tanto da linguagem como do ”ambiente”de execução (JVM)
É distribuída com um vasto conjunto de bibliotecas (ou APIs)
Possui facilidades para criação de programas distribuídos e multitarefa (múltiplas linhas de execução num mesmo programa)
Desalocação de memória automática por processo de coletor de lixo (garbage collector)
Carga Dinâmica de Código - Programas em Java são formados por uma coleção de classes armazenadas independentemente e que podem ser carregadas no momento de utilização.

Eis um exemplo:

```
public abstract class Animal {
   public abstract void fazerBarulho();
}

public class Cachorro extends Animal {
   public void fazerBarulho() {
       System.out.println("AuAu!");
   }
}

public class Gato extends Animal {
   public void fazerBarulho() {
```

```
        System.out.println("Miau!");
    }
}
```

2.15 PHP

PHP (um acrônimo recursivo para “PHP: Hypertext Preprocessor”) uma linguagem de programação de computadores interpretada, livre e muito utilizada para gerar conteúdo dinâmico na Web.

A linguagem surgiu por volta de 1994, como um subconjunto de scripts Perl criados por Rasmus Lerdof, com o nome Personal Home Page Tools. Mais tarde, em 1997, foi lançado o novo pacote da linguagem com o nome de PHP/FI, trazendo a ferramenta Forms Interpreter, que era na verdade um interpretador de comandos SQL.

Trata-se de uma linguagem modularizada, o que a torna ideal para instalação e uso em servidores web. Diversos módulos são criados no repositório de extensões PECL (PHP Extension Community Library) e alguns destes módulos são introduzidos como padrão em novas versões da linguagem. Muito parecida, em tipos de dados, sintaxe e mesmo funções, com a linguagem C e com a C++. Pode ser, dependendo da configuração do servidor, embutida no código HTML. Existem versões do PHP disponíveis para os seguintes sistemas operacionais: Windows, Linux, FreeBSD, Mac OS, OS/2, AS/400, Novell Netware, RISC OS, IRIX e Solaris

Construir uma página dinâmica baseada em bases de dados é simples, com PHP. Este provê suporte a um grande número de bases de dados: Oracle, Sybase, PostgreSQL, InterBase, MySQL, SQLite, MSSQL, Firebird etc, podendo abstrair o banco com a biblioteca ADOdb, entre outras.

PHP tem suporte aos protocolos: IMAP, SNMP, NNTP, POP3, HTTP, LDAP, XML-RPC, SOAP. É possível abrir sockets e interagir com outros protocolos. E as bibliotecas de terceiros expandem ainda mais estas funcionalidades.

Veja um exemplo de PHP

```
<? ...
      $tipss =mysql_result($result,0,'SSERVTIPSS');
      $porss =mysql_result($result,0,'SSERVPORSS');
      $daali =mysql_result($result,0,'SSERVDAALI');
      $anexo =mysql_result($result,0,'SSERVANEXO');
      $query = "select * from ANDAM where ANDAMNSERV ='$nserv1' "  ;
      $result = mysql_query($query);
      $quantos = mysql_num_rows($result);
      $quantos++;
      ?>
      <table > <tr>
      <td>Numero da solicitacao de servico<td><? echo $nserv ?><tr>
      <td>autor<td><? echo $nomeu ?><tr>
      <td>cliente<td><? echo $clien ?><tr>
      <td>CA responsavel<td><? echo $cares ?><tr>
      <td>interlocutor no cliente<td><? echo $intcl ?><tr>
      <td>fone<td><? echo $fonic ?><tr>
      <td>email<td><? echo $emaic ?><tr>
```

2.16 J

Para os que acharam APL uma linguagem meio sem pés nem cabeça, eis aqui J. Olhando para o J, o APL passa a ser tão comportada quanto um COBOL da década de 70.

Para entender o J, precisamos estudar a vida do cara que inventou o APL, o canadense Ken Iverson. Na minha opinião, o sujeito foi um gênio. Coloco-o sem nenhum medo de errar na galeria dos grandes matemáticos da humanidade, talvez o primeiro (junto com Mandelbroot) que tenha realmente conseguido casar com sucesso a matemática e a computação.

Depois de propor o APL como uma notação matemática (década de 50), de liderar o grupo que converteu o APL em linguagem de programação (década de 60 na IBM) e de liderar a popularização da linguagem (anos 70 e começos dos 80), em meados dos 80, ele chegou a algumas conclusões:

- o uso de um alfabeto não usual (para ser educado), restringia o uso do APL (lembremos que ainda não havia o windows e portanto para usar APL havia que comprar hardware especializado - e caro).
- o desenvolvimento continuado por 30 anos da linguagem apontou algumas inconsistências teóricas no modelo. Não esqueçamos que o Iverson era um matemático da pesada
- o custo que o APL sempre teve inviabilizava seu uso pelos menos aquinhoados

Dessas elocubrações nasceu a linguagem J. O nome não tem explicação, exceto a dada por um de seus autores (”Why ’J’? It is easy to type.”). Há quem diga que J segue ”I”no alfabeto, sendo este ”I”a notação Iverson. Seja como for, J não tem nada a ver com Java. Para maiores detalhes veja www.jsoftware.com.

Eis a seguir um programa J:

```
gerasima =: 3 : 0
r=.(2$y)$(1+?200$2 2 2 2 2 3 3 3 4 4 4 5 6)*2+-<.1.5*?200$2 1 1 1 2 1 2
xx=.(y?(<.y*1.6))
z=.+/|:r*($r)$xx
r,.((y,1)$z),.(y,1)$xx
)
```

Equivale ao programa gerasima do workspace vivo128, que gera um sistema de n incógnitas e n equações, depois o resolve para obter os termos independentes e possibilitar propor ao aluno que descubra as incógnitas.

2.17 Lua

Lua é inteiramente projetada, implementada e desenvolvida no Brasil, por uma equipe na PUC-Rio (Pontifícia Universidade Católica do Rio de Janeiro). Lua nasceu e cresceu no Tecgraf, o Grupo de Tecnologia em Computação Gráfica da PUC-Rio. Atualmente, Lua é desenvolvida no laboratório Lablua. Tanto o Tecgraf quanto Lablua são laboratórios do Departamento de Informática da PUC-Rio. É, ao que eu saiba, a única iniciativa brasileira na área de linguagens a obter alguma aceitação mundial.

O projeto e a evolução de Lua foram apresentados em junho de 2007 na HOPL III, a 3a Conferência da ACM sobre a História das Linguagens de Programação. Essa conferência ocorre a cada 15 anos (a primeira foi em 1978 e a segunda em 1993) e somente poucas linguagens são apresentadas a cada vez. A escolha de Lua para a HOPL III é um importante reconhecimento do seu impacto mundial. Lua é a única linguagem de programação de impacto desenvolvida fora do primeiro mundo, estando atualmente entre as 20 linguagens mais populares na Internet (segundo o índice TIOBE). (Dados obtidos em www.lua.org)

Eis um exemplo de lua

```
function fatorial(n)
  if n < 2 then
    return 1
  else
    return n * fatorial(n-1)
  end
end

function exemplo()
  print("Ola mundo\n\n")

  print("Fatorial de seis: " .. fatorial(6) .. "\n")

  print("Tchau......\n")
end

-- execução da função
exemplo()
```

Capítulo 3

Python

A pergunta: **Porque Python ?**

- É uma linguagem moderna (nasceu em 1990) e herdou tudo de bom que as demais linguagens já tinham nessa época. Uma coisa que ela não herdou foi a preocupação exigente de desempenho: Em 90, já havia computação abundante. Com isso a linguagem é redonda e não impõe trancos e barrancos para funcionar. [1]

- Multiplataforma: Unix, Windows, Apple, celular android, raspberry e outros ambientes menos famosos rodam exatamente o mesmo programa.

- Freeware: pode copiar, baixar, instalar, vender, ceder... pode fazer qualquer coisa sem infringir a lei e sem dever explicações a ninguém.

- Uso no mainstream: Olhem o que diz o Philip Guo, um blogueiro da CACM "oito dos top-10 Departamentos de Ciência da Computação e 27 dos top-39 Departamentos de Ciência da Computação das principais universidades nos EUA usam Python nos cursos introdutórios de CS0 e CS1" (CS0 e CS1 são os cursos introdutórios de Computer Science lá). As 39 universidades pesquisadas por ele incluem: MIT, Berkeley, Stanford, Un. Columbia, UCLA, UIUC, Cornell, Caltech, Un. Michigan, Carnegie Mellon, entre outras [2]

- Muito Usada: o índice TIOBE [3] lista o Python como a quinta linguagem mais usada no mundo em 2017. . Em junho de 2018, Python já é a quarta linguagem mais usada. Perde apenas para java, c e c++. E, corre por fora...

  Se você olhar o estudo da TIOBE https://www.tiobe.com/tiobe-index/ verá o bonito papel desempenhado pelo Python.

Linguagem	17	12	07	02	97	92
Java	1	1	1	1	12	-
C	2	2	2	2	1	1
C++	3	3	3	3	2	2
C#	4	4	7	17	-	-
Python	5	7	6	11	27	-
VB.NET	6	19	-	-	-	-
PHP	7	6	4	5	-	-
JavaScript	8	9	8	8	19	-
COBOL	25	28	17	9	3	10
Lisp	32	12	14	12	9	5
Prolog	33	32	26	15	20	12
Pascal	106	15	19	97	8	3

  Consulta em 07/2017

  Uma pequena explicação desta tabela: ela lista de 5 em 5 anos (o que no mundo da computação é quase uma eternidade) a lista dos top-10 ou as linguagens de programação mais usadas. O começo é 1992, e daí de 5 em 5 anos, por 30 anos ou 6 ciclos, até o último em 2017. Diversos fenômenos podem ser observados, eis alguns:

  – A performance notável de Java, cuja explicação parece estar na palavra *portabilidade* ou seja o mesmo programa fonte rodando em inúmeras plataformas distintas. Isso é conseguido pelo conceito de java machine. Foge ao

[1] Em compensação, quando se organiza uma maratona de programação, para um determinado problema, impõe-se 1 segundo de CPU quando ele é resolvido em C++ e pelo menos 10 (10 vezes mais) para quando ele é resolvido usando Python

[2] https://cacm.acm.org/ blogs/blog-cacm/176450 -pyhon-is-now-the -most-popular-introductory- teaching-language-at- top-u-s-universities/ fulltext, acessado em agosto/2017

[3] www.tiobe.com, acessado em agosto/17

escopo deste texto descrever o conceito, mais em resumo é uma capa ou camada de compatibilidade que se adiciona ao ambiente de maneira a tornar ou permitir a mesma coisa sobre coisas diferentes.

  - A derrocada do Pascal, nascido como uma execelente ferramenta de aprendizado e que em algum momento se perde completamente.
  - A consistência de C e seus agregados (c++ e c#), seu sucesso pode ser descrito como eficiência e o possível alto nível de portabilidade.
  - A morte anunciada e posteriormente executada do COBOL. Ele só permanece com alguma relevância graças ao legado ou aos milhões de linha de código que ainda precisam ser mantidos vivos.
  - Quase a mesma coisa pode ser dita das linguagens da inteligência artificial: Lisp e Prolog. Elas migram para a generalidade: Java, C e Python.
  - A irrupção da Internet: PHP, Javascript e Python, por trás dos panos. Inclusive da Internet das Coisas (aqui com Java).

- Pacotes, pacotes, pacotes. A comunidade Python mantém o PYPI (Python Package Index) que é um repositório freeware com pacotes de programação: hoje ele tem mais de 110.000 pacotes. Só para ter uma idéia, um pacote destes, o NUMPY (numerical computation in Python) tem um manual de referência de mais de 1500 páginas.

- Inteligência Artificial: Quando o Google resolveu aposentar o DistBelief e criar uma nova ferramenta de Machine Learning, escolheu a linguagem Python para desenvolvê-la: nasceu o Tensor Flow, que está por trás de inúmeras iniciativas google: tradução, classificação de imagens, elaboração de perfis etc. [4] Logo na entrada do produto o aviso: pode usar em Java, C++ ou Go. Mas, se quiser usar a funcionalidade completa do produto, use-o em Python, que ele está feito nessa linguagem.

- Fácil: parece portugol, ou vá lá: inglêsgol. Não tem caracteres sobrando: sem ponto e vírgula, sem chaves nem colchetes. O que se escreve, ele executa e raramente dá erro.

[4] www.tensorflow.org

Capítulo 4

Instalação e alô mundo

Python é freeware, não custa nada, você pode copiar e usar para o que desejar, sem pedir licença a ninguém, e pode inclusive cobrar pelo software que você desenvolver usando Python. A partícula *free* em freeware significa liberdade para fazer o que você quiser sem nenhum tipo de restrição e não gratuidade, como alguns pensam.

Existem versões de Python para quase todos os sistemas operacionais conhecidos, eis alguns: AIX (da IBM), AROS (Amiga), AS/400, HPUnix, Linux, MacOS, MS-DOS, OS/2, PalmOS, Playstation, Solaris, VMS, Windows (32 e 64), WindowsCE, entre outros.

Esta multiplicidade de versões acrescenta uma qualidade imensa aos programas Python: desde que eles não usem nenhum recurso específico de uma dessas arquiteturas, serão 100% (ou quase isso) portáveis podendo rodar o mesmo programa em todas essas plataformas.

4.1 Versão

As versões de Python são numeradas em 3 níveis: O primeiro número identifica a versão global que atualmente pode ser 2 ou 3. Além das versões 0 e 1, quase experimentais, a linguagem começou sua carreira global com a versão 2, que chegou a ser bastante usada, muito software foi construído com ela, e esse fato impede agora que ela seja totalmente substituída pela versão 3, o que seria o melhor dos mundos. Se você está começando agora, não há o que pensar: deve-se escolher a versão 3. O problema é que na mudança de 2 para 3 perdeu-se alguma compatibilidade o que impede a livre transição de uma a outra. A versão 2 já está com sua sentença de morte estabelecida, ela só será suportada até 2020. Desse ponto em diante quem quiser usá-la o fará por sua conta e risco.

O segundo número indica a sub-versão, que diferencia algum nível maior de correção e sobretudo de lançamento de novidades. Finalmente o terceiro número vai sendo incrementado à medida em que pequenas correções e melhoramentos vão sendo construídos. Eis uma pequena versão das modificações:

Python 3.6.5	liberado em 28 Março 2018
Python 3.6.4	19 Dezembro 2017
Python 3.6.0	23 Dezembro 2016
Python 3.5.1	07 Dezembro 2015
Python 3.3.7	19 Setembro 2017
Python 3.2.6	11 Outubro 2014
Python 3.0	3 Dezembro 2008
Python 2.7.8	1 Julho 2014
Python 2.6.2	14 Abril 2009
Python 2.1.3	8 Abril 2002
Python 1.4	25 Outubro 1996

Existe tamanha diversidade no mundo Python que é impossível tratar todas as alternativas de uso. Primeiro, vai-se escolher o Python 3 e depois vai-se olhar 3 delas: a mais tradicional possível, sob Windows (nas 2 versões, sob 32 e sob 64 bits) e também uma instalação alternativa que permite instalar o Python em um pendrive para usá-lo em qualquer computador, mesmo sem os direitos de administrador nesse computador e o Python interativo.

4.2 sob Windows

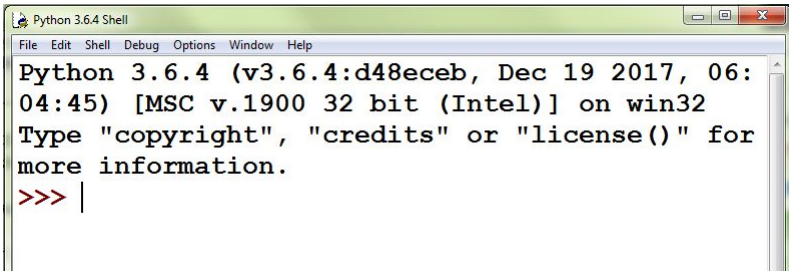
A primeira decisão é sob qual ambiente de windows você está. Uma maneira simples de descobrir é solicitar o *Painel de Controle*, e aqui solicitar a aba *System*. Isto mostra um texto com algumas características do sistema e lá no meio do texto, vem a frase “System type”. O que aparecer aí deve servir de guia para a escolha do Python a baixar e instalar: 32 ou 64 bits.

Daí, deve-se ir a `https://www.python.org/downloads/` e escolher a última versão de distribuição. Usualmente há versões mais novas que a última, mas são conhecidas como experimentais e não são recomendáveis, pois ainda não estão cristalizadas ou consolidadas. Deve-se escolher sempre o Python 3 e não o 2, que este vai desaparecer nos próximos anos e só está nessa lista para quem precisa usar e suportar software antigo.

No processo de instalação, especial preocupação deve ser tida ao verificar em qual diretório a instalação vai ser feita. Eu recomendo criar um diretório novo (não usar a sugestão do instalador) no primeiro nível, logo abaixo do nível raiz (como em `c:/python3`). Sob este diretório, o instalador vai criar diversos sub-diretórios (DLL, doc, etc, include, LIB, libs, Scripts, tcl e tools). É importante olhar isto, pois para instalar novos pacotes é preciso entrar no diretório Scripts, como se verá adiante. Logo é preciso saber onde está.

Depois de instalar o Python, deverá aparecer no seu menu inicial um ícone com 2 cobrinhas (uma azul e outra amarela)

o desenho padrão do Python. O nome deve identificar qual a versão instalada e quando se clicar nele deve-se abrir uma tela inicial do python, como em



Nessa tela minimalista, surgem algumas informações como versão, data e hora da liberação desta versão, tipo de sistema sob qual ela está rodando. Depois algumas opções disponíveis e finalmente o símbolo

```
>>>
```

Ele é o *prompt* (prontidão) que é um conceito usado por praticamente todos os softwares interativos para sinalizar ao operador que o software está *pronto* para aceitar e executar comandos. Enquanto este símbolo não aparecer, isso significa que o processador está ocupado fazendo alguma coisa e não pode ser interrompido pelo operador.

Aqui já se podem escrever comandos Python como em

```
>>> 1+2
3
>>>
```

Aqui pediu-se ao Python a execução de $1+2$ e ele respondeu 3, seguido de um novo *prompt*. Esta modalidade de uso do Python e conhecida como interativa e neste caso ele funciona como se fosse uma calculadora. Basta comandar alguma coisa e ele a executará.

4.3 Alo Mundo

Tradicionalmente o primeiro programa que um novato deve construir em qualquer novo ambiente é chamado `Alo Mundo`. Sua função é apenas mostrar esta mensagem e sua execução com sucesso sinaliza que o processo de criar um programa fonte, salvá-lo, compilá-lo e depois executá-lo foi bem sucedido. Vamos a isso, então:

1. na tela do Python, escolha `File`

2. no menu que vai aparecer, escolha `New File` e lembre que estes dois passos podem ser comandados apertando `Ctrl N`

3. Vai-se abrir uma nova tela em branco, muito parecida com a tela do Python. Parece Python, mas não é Python e sim o bloco de notas do Python[1]

4. Na tela do bloco de notas do Python escreva o seguinte

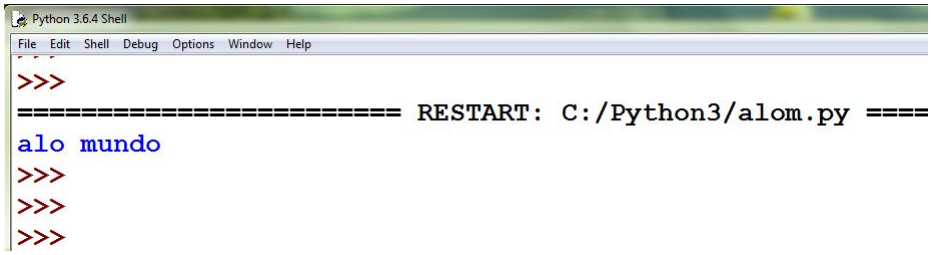
   ```
   print("alo mundo")
   ```

[1] Uma maneira enviezada de identificar qual tela é qual é esta: o bloco de notas do python tem o comando Run enquanto a tela do Python não tem este comando.

5. Agora você vai comandar a compilação (interpretação) e execução desse script (tecnicamente isto não é um programa e sim um script: é uma diferença semântica meio irrelevante, mas será estudada depois). Quem faz isto é a tecla `F5`. Ou também o comando `Run`, já destacado.

6. O Python antes de qualquer coisa vai pedir para você um nome sob o qual o programa (ou script) vai ser salvo. Esta é uma excelente exigência do Python: garante que aconteça o que acontecer, o programa que você escreveu e está tentando executar, será salvo em disco, sem risco de que ele se perca, caso o pior (tela azul da morte, por exemplo) aconteça. O nome que você der terá agregado o sufixo `.py` e será salvo no lugar onde você informar [2]

7. Tendo sido salvo e compilado com sucesso a tela do Python vai mostrar



   Esta tela mostra a execução do script que você criou e a mensagem sendo mostrada.

8. Isto significa que você teve sucesso na instalação do python e na criação e execução do seu primeiro programa.

4.4 Winpython

Esta implementação aparece aqui por algumas razões, sendo a principal a **portabilidade** do ambiente de desenvolvimento:

- Permite usar Python em máquinas nas quais você não é o administrador, coisa que o Python padrão exige para ser instalado.
- Como consequência, este Python é portável, e pode ser carregado em um pendrive ou similar, ainda que haja alguma demora no processo de carga: o winpython é enorme.
- Winpython é um ambiente científico, pelo que já vem com SciPi e Numpy devidamente instalados, além de eventualmente outros pacotes importantes.

4.5 Ipython

É um python interativo, no qual fica mais evidente o diálogo entre usuário e Python. Na sua versão original (Python) é uma linguagem de programação que conta com um enfoque interativo também. No Ipython, ao contrário, é um ambiente interativo, que por acaso também dá acesso a uma linguagem de programação.

Ele aperece aqui nesta lista pois é a versão disponibilizada nos laboratórios do curso de engenharia elétrica da UFPR. Neste ambiente, eis o caminho das pedras:

1. Usar seu login para entrar no sistema.
2. Na tela de atividades, digitar IP e aguardar. Escolher Ipython.
3. Carregar também o editor GEDIT. Manter ambos abertos. (Ipython e GEDIT)
4. Escrever o código python no GEDIT e salvar o programa e sua execução no diretório /eletrica, com o nome de aa.py
5. Dentro do Ipython executar `%run aa.py`
6. Para corrigir, edite no GEDIT, salve e reexecute dentro do Ipython.

4.6 Python na WEB

Uma alternativa disponível a quem tenha acesso à web e não pretenda (ou não possa) instalar nada em sua máquina é usar um ambiente python disponível na web. Há vários, mas um funcional é `https://www.pythonanywhere.com`. Aqui você precisa criar um login e depois disso passa a acessar um ambiente python completo. Já estão instalados nele vários pacotes importantes (teste: numpy, sympy, matplotlib, django). Este ambiente é perfeito para uma consulta rápida.

[2] Sugestão para quem está começando: compre e use um pendrive para guardar seus programas: eles irão com você. E não esqueça de fazer cópias deles – o famoso backup.

Capítulo 5

Variáveis, sua entrada e saída

Uma variável é um objeto do mundo do software que pode ter algumas abordagens. A primeira, é aquela da matemática, quando se escreve $y = f(x)$ e se quer dizer que y é uma variável que depende do valor de x. O que existe aí, nessa expressão, é uma função que associa o valor de y ao valor de x. Tanto x quanto y nesse enfoque são variáveis: isto é são valores eventualmente desconhecidos, mas que têm um valor qualquer. Outra abordagem é da ciência da computação: nela, uma variável é um pedacinho de memória dentro de um programa (que está devidamente registrado nessa memória) e que armazena um determinado valor. Cada variável precisa ter um nome, para que possa ser referenciada. Esse nome precisa ser único (não se pode ter duas variáveis com o mesmo nome), e depois que ela foi criada, usar essa variável num comando qualquer é exatamente a mesma coisa que usar o valor que a variável tem. Por exemplo, se a variável A tem o valor 10, tanto faz escrever $A + 1$ como $10 + 1$, só que no primeiro caso, a generalidade é maior, pois independentemente do valor que ela tiver, depois de executar $A + 1$, ter-se-á um valor uma unidade maior do que o valor anterior de A.

5.1 Atribuição

O comando importante aqui é o chamado **atribuição** . É ele que permite:

- criar uma variável
- alterar o valor de uma variável

As duas funções só se diferenciam pelo existência prévia da variável. Se ela não existe, usar a atribuição significa criar a variável. Se ela já existia (e portanto tinha um valor qualquer) usar a atribuição nela vai alterar o valor dela para este, que é citado agora na atribuição.

O formato do comando é

```
variável = valor
```

Onde `variável` é o nome da variável, `=` identifica o comando de atribuição e `valor` é o valor que a variável passará a ter depois desta atribuição. Por exemplo:

`A=10`	A variável de nome A é criada (ou alterada) e passa a ter o valor 10.
`X=-33.4`	Agora o valor de X é o número negativo -33.4
`NM='alfa'`	A variável agora se chama NM e seu valor é uma palavra, que é 'alfa'. Atente para a presença das aspas, elas se

Algumas observações importantes. O caráter de atribuição é = e ele é lido como `recebe`. Então, o comando $A = 10$ deve ser lido como `A recebe 10`. Outra observação é que este sinal tem outras funções no mundo. Por exemplo, na matemática o sinal de igual é isso mesmo: para indicar uma igualdade. Igualdade não é a mesma coisa que atribuição. O problema é que se não se usar o igual, que outro símbolo pode-se usar ? Outras linguagens resolveram esse problema, usando dois caracteres (por exemplo, Pascal usa `:=`) enquanto outras usaram caracteres novos, como por exemplo APL, que usa ←.

A discussão de qual símbolo usar para a atribuição (que lembrando é um conceito que não existe na mátemática) é antiga e poderia ser aberta aqui. Mas, pegando carona no `C`, linguagem avô de boa parte do leque de linguagens usadas no século XXI, (tais como C++, Java, PHP, entre outras), o Python também associou o sinal de = à atribuição. A pergunta é: o que usar quando se falar sobre igualdade entre dois valores ? Neste caso, usam-se 2 caracteres iguais. Então, para perguntar se as variáveis A e B têm o mesmo valor, escreve-se $A == B$, já que se se escrever $A = B$, o que se estará mandando fazer é atribuir o valor da variável B para a variável A.

Voltando ao formato do comando de atribuição, acima descrito, vamos estudar o que pode ser `valor` citado no formato do comando. Como o nome sugere, `valor` pode ser um número como 13, 2780.4, -33 e assim por diante. Mas, também pode ser outra variável, como em `A=X`. Aqui se está mandando copiar o valor de `X` para a variável `A`. Note que não se sabe que valor é esse. Não importa, ele é copiado de `X` para `A`.

Finalmente, esse valor pode ser uma expressão aritmética como em `A=11+8`, ou em `A=V-3`, ou finalmente `A=cos(theta)`. No primeiro caso, `A` vale 19, no segundo vale o valor de `V` menos 3 e no terceiro caso `A` vale o coseno do angulo `theta`.

Agora pode-se ver a importância da presença das aspas no comando de atribuição: acompanhe no exemplo `A='alfa'` e `A=alfa`. No primeiro caso, é o conteúdo `alfa` que está sendo atribuído à variável A. No segundo pressupõe-se a existência de uma variável de nome `alfa` e é esta variável que tem seu valor (seja ele qual for) copiado para a variável `A`.

As aspas denotam um conteúdo alfanumérico, que estruturalmente é diferente de um conteúdo puramente numérico. Há muitas diferenças que ainda serão estudadas, mas uma importante é que conteúdos (e variáveis) numéricas podem ser usadas em expressões aritméticas enquanto variáveis alfanuméricas causam erros de execução quando se tenta envolvê-las em expressões numérico-aritméticas.

`>>> A=5.5`	Cria-se a variável A, flutuante, com o valor 5.5
`>>> B='oba'`	Cria-se B alfanumérica contendo 'oba'
`>>> A+B`	Tenta-se somar as duas variáveis. Há um erro

O erro é sinalizado pela mensagem `TypeError: unsupported operand type(s) for +: 'float' and 'str'` cuja tradução é: Erro de tipo: não se pode somar (+) um tipo float com um tipo str.

5.1.1 Atribuições malucas

O Python, na esteira das linguagens mais modernas, tem diversas alternativas de atribuição, que serão vistas agora. Na minha opinião, este capítulo pode ser ignorado, pois os acréscimos que serão vistos aqui, embora acrescentem riqueza ao ambiente, muitas vezes são únicos do Python e ao usá-los, afastamo-nos da facilidade de transpor nosso código para outras linguagens (C++ ou Java, as maiores candidatas). Eu considero transposição importante, cada um que julgue se deve ou não priorizá-la.

Múltiplo assinalamento

Pode-se atribuir o mesmo valor a diversas variáveis:

```
a=b=c=d=0
```

Neste caso, foram criadas (alteradas) as variáveis a, b, c e d todas com o valor zero.

Múltiplo assinalamento com diversos valores

O valor não precisa ser o mesmo, pode variar. Veja no exemplo

```
>>> a,b,c=5.6,0,'alfa'
>>> a
5.6
>>> b
0
>>> c
'alfa'
```

Note que a quantidade de variáveis à esquerda e de expressões à direita precisa ser a mesma, sob pena de um erro como em

```
    a,b,c=1,2,3,4
ValueError: too many values to unpack (expected 3)
```

Swap de variáveis

Em programação muitas vezes é necessário inverter o conteúdo de 2 variáveis. Em situação normal, isso exige o uso de uma variável auxiliar como em

```
>>> aux=a
>>> a=b
>>> b=aux
```

Python possui um mecanismo simples para realizar a troca sem usar variáveis adicionais. O mesmo exemplo acima pode ser feito assim

```
a,b=b,a
```

Cabeça e cauda

Em muitos ambientes de programação, notadamente em programação funcional, baseada em listas, como na linguagem LISP (List Processor) e mesmo na linguagem PROLOG (Processador Lógico) ambas muito usadas no reino da Inteligência artificial, há uma estratégia de programação, fortemente baseada em recursividade e que exige processar um elemento da lista de cada vez, diminuindo-a sempre em direção à lista vazia, que neste caso é o que se chama de *caso básico* (para entender melhor esta abordagem, veja algum texto sobre recursividade). Vou descrever o funcionamento disto como em Lisp, que foi quem começou com essa abordagem. Dada uma lista, por exemplo `[10,20,30,40]`, ela sempre pode ser dividida em cabeça (que em lisp se chama CAR) e corpo (que em lisp se chama CDR). A cabeça é o primeiro elemento da lista e o corpo é o que sobra quando a cabeça é eliminada da lista. Em lisp

```
> (car '(10 20 30 40))
10
> (cdr '(10 20 30 40))
(20 30 40)
```

Em LISP a cabeça é sempre o primeiro, em Python pode ser o primeiro ou o último. Veja na implementação

```
>>> seq=[10, 20, 30, 40]
>>> a,*b=seq
>>> a
10
>>> b
[20, 30, 40]

>>> *a,b=seq
>>> a
[10, 20, 30]
>>> b
40
```

Assinalar e operar ao mesmo tempo

Em programação é muito frequente escrever `a=a+1`. Este comando em Python pode ser escrito como `a+=1`. Note que o perigo reside – numa distração – em inverter os dois sinais. Ao se escrever `a=+1` está-se comandando outra coisa: a criação (ou alteração) da variável a com o valor de $+1$. Veja o perigo: você cometeu um erro, e o Python não avisa, faz o que você mandou (como sempre), mas eventualmente não era isso que você queria.

O par de sinais pode ser `+=`, além de `-=`,`*=`,`/=`,`%=`, `//=`, `**=` entre outros.

Exclusão de variável

Embora não seja usual, o programador pode excluir as variáveis quando estas deixarem de ser úteis. Não é usual, pois ao encerrar o Python automaticamente tudo o que foi usado é excluído e os recursos liberados. Mas, havendo necessidade o programador pode excluir variáveis com o comando `del`. Veja-se

```
>>> a=5
>>> b=6
>>> del a,b
>>> a
NameError: name 'a' is not defined
```

5.2 Tipos

Graças a essa distinção vista acima, pode-se pontuar a respeito de uma característica importante das variáveis. Cada variável terá um tipo que sinalizará como ela será manipulada depois de criada. Os principais tipos são: numérico inteiro, numérico real, alfanumérico e lógica.

Ao contrário de outras linguagens (como o C, por exemplo), as variáveis em Python não têm um tipo explicitamente declarado. É o seu valor que determina qual é o seu tipo. [1] Então,

`A=10.3`	A é uma variável do tipo real, que em Pythons é conhecida como `float`
`B=5`	A variável B é do tipo inteiro.
`C=10.0`	A variável C é tipo float
`HAL='ivo viu a uva'`	A variável HAL contém a frase `ivo viu a uva`
`GA=True`	GA é uma variável do tipo lógico que tem o valor de verdadeiro
`J=False`	J é uma variável do tipo lógico e tem o valor Falso

[1] Esta regra nem sempre é verdadeira, por exemplo, ao usar o pacote Numpy. Mas, por enquanto pode-se aceitá-la como verdadeira.

A variável do tipo lógico só admite dois valores `True` e `False`. Obviamente, estes dois nomes não servem para dar nomes a variáveis. É isto que se chama de palavra reservada .

Os tipos numéricos podem ser inteiros (`int`) ou reais, que no Python recebem o nome de `float`. Poder-se-ia argumentar que o tipo inteiro é desnecessário (já que o conjunto dos reais contém o conjunto dos inteiros ou $\mathbb{R} \subset \mathbb{Z}$), mas isto não é verdade, já que:

- Dentro do computador, a matemática inteira é dezena de vezes mais rápida que a matemática real. Então quando aplicável, sempre vale a pena ser mais eficiente.
- Mas, principalmente, o conjunto dos inteiros tem uma característica que o real não tem: a enumerabilidade. Tal característica é muito importante (por exemplo, na indexação) e ela exige que o tipo seja inteiro em certos momentos.

Feita esta distinção, os conteúdos que uma variável numérica pode ter são:

inteiro : os 10 dígitos `0,1,2,3,4,5,6,7,8 e 9`, um sinal opcional `+ ou -`.

flutuante : os mesmos dígitos e sinal e além destes, o sinal de ponto decimal. Note que usa-se em Python o ponto decimal e não a vírgula decimal, herança dos países de língua inglesa. Opcionalmente pode-se ter a letra `E` ou `e` significando exponencial. E, se presente, indica o expoente de 10, que deve ser multiplicado pelo número antes do E. O expoente também pode ter opcionalmente um sinal que pode `+` ou `-`. Exemplos:

`3E4`	Significa 3×10^4 ou 3000
`-4E2`	-4×10^2 ou -400
`5E-3`	5×10^{-3} ou $5 \times \frac{1}{10^3}$ ou ainda 5×0.001 que é 0.005.

5.2.1 Mudando o tipo

O tipo pode ser modificado, por exemplo em

`A=9`	Cria-se a variável A como inteiro
`B=6.5`	Cria-se B como flutuante
`A=B`	A que era inteiro passa a ser flutuante, pois recebeu 6.5
`A='Curitiba'`	A agora é uma variável string
`A=False`	Finalmente, A agora é uma variável lógica

Nenhuma dessas modificações causa algum tipo de erro ou de manifestação do Python. São ações normais e corriqueiras.

Em alguns momentos da programação, é necessário explicitamente modificar um tipo de variável. Esta necessidade já vai aparecer (ao tratar o comando `input`, por exemplo), mas ela vai ser vista aqui do ponto de vista formal. Para mudar o tipo de uma variável pode-se usar

`int(x)`	Pega o conteúdo de x e transforma-o (se possível) em um inteiro.
`float(x)`	Transforma x em flutuante
`str(x)`	Transforma x em uma string

Obviamente a transformação só vai acontecer se ela for possível. Não adianta, por exemplo, definir `A='alfa'` e pedir `int(A)`, já que 'alfa' não pode ser transformada em um número inteiro. Mas, se `B='123'` deve-se notar que B é uma variável string (por causa das aspas), mas ela pode ser transformada em inteira e neste caso, valerá 123, pois só é formada por dígitos numéricos. Já a conversão em string é sempre possível, e sempre que for ordenada será realizada.

Para a representação de números inteiros o Python usa um sistema de precisão ilimitada, que permite usar muitos (muitos !) dígitos, veja-se por exemplo o fatorial de 50 é
`30414093201713378043612608166064768844377641568960512000000000000`. O programador não precisa fazer nada (ao contrário de outras linguagens mais antigas como C ou C++). O Python simplesmente lança mão de quantas casas forem necessárias para a representação numérica inteira pedida.

Já na representação flutuante, podem surgir problemas derivados do fato de que o Python usa para representação interna um formato binário limitado composto de abcissa e mantissa. Em outras palavras, um número flutuante usa uma quantidade finita de bits para ser representado, o que faz com que haja perda de precisão nessa representação. Veja-se um exemplo prático (uma sessão real de Python)

```
>>>   a=0.1
>>>   print("{:2.5f}".format(a))
0.10000
>>>   print("{:2.30f}".format(a))
0.100000000000000005551115123126
```

O comando format acima, parece, e é, bem misterioso, mas o que ele pede é algo prosaico, que a variável a seja impressa, no primeiro caso, com 5 decimais e no segundo com 30 decimais. Observe o que aconteceu. O surgimento de números (aparentemente aleatórios) a partir da 18^a casa é resultado de imprecisões derivadas do uso do tipo flutuante. Perceba-se que não deve haver maiores problemas: um erro na 18^a casa não é para preocupar mas, ele está lá.

5.2.2 Nomes

A regra para dar nomes em Python (regras essas que se aplicam à nomeação de variáveis) são simples:

- uma única palavra, ou seja não se admitem espaços em branco dentro dos nomes.
- pode-se usar o caracter sublinha (`_`) para separar palavras no nome
- o primeiro caractere deve ser uma letra
- depois do primeiro pode-se usar letras ou números
- Letras maiúsculas são diferentes das letras minúsculas (ou seja `OBA` é diferente de `Oba` que não é a mesma coisa que `oba`).

O leitor não deve se preocupar muito com estas regras, substituindo-as por algum bom senso: os nomes devem ser autodeclarativos (sobre para que serve a variável), com nomes nem muito grandes (provocam erros de digitação) nem muito pequenos (para que serve mesmo a variável B ?). Uma regra muito usada por aí sugere usar várias palavras ou abreviaturas no nome, separando umas das outras pelo uso de maiúsculas. Usando esta regra pode-se ter

`SaldoConta`	uma variável para conter o saldo da conta
`LucroAntesIR`	auto-explicativo

Os detratores da regra acima dizem que não se deve misturar maiúsculas e minúsculas no nome, pois isto acaba sendo uma fonte de erros de digitação. Estes dizem que as variáveis devem ser todas minúsculas ou todas maiúsculas.

Enfim, a sugestão aqui é que cada leitor deve definir a sua regra de formação de nomes e se ater a ela, sempre visando a minimização dos erros de digitação.

A partir da versão 3, o Python pode aceitar caracteres acentuados na formação de nomes, pois aqui por padrão usa-se o conjunto de caracteres Unicode, através do Unicode Transformate Format-8 (UTF-8). Mas, pensando francamente, não parece ser uma boa idéia: se você definir uma variável de nome `SAÍDA` e depois chamá-la de `SAIDA`, para o Python serão duas variáveis distintas, e portanto aqui está uma nascente fonte de problemas. Tente não usar caracteres acentuados ao programar. Você não vai se arrepender. Conteúdos dentro de aspas (constantes) podem e devem usar caracteres acentuados: vamos homenagear a Língua Portuguesa, e dentro das aspas não se corre nenhum perigo. Como em `ESTADO='Paraná'`.

5.3 Expressões aritméticas

As variáveis e constantes numéricas podem ser operadas e interligadas usando-se as operações aritméticas usuais (e algumas não tão usuais) da matemática convencional. Para isso são usados os símbolos a seguir descritos, que buscam seguir – tanto quanto possível – as convenções usuais da escrita matemática. Por exemplo

```
>>> a = 5
>>> b = 3.6
>>> a + 4
9
>>> b * 3
10.8
>>> (a + b) / 2
4.3
>>> a - 6
-1
>>> a - -8
13
```

Note que a adição é representada pelo sinal de "mais" ($+$), a subtração por "menos" ($-$), a multiplicação pelo asterísco ($*$) e a divisão real pela barra ($/$).

Note também que a prioridade é determinada pelo uso de parênteses, que as vezes é redundante, mas nem sempre, pois se o comando fosse escrito como $2 + 8/2$ sem parênteses, o Python primeiro faria a divisão de $8/2$ dando 4 e depois a soma de 2 com 4 resultando 6. Ao se escrever $(2 + 8)/2$ primeiro a soma dando 10 e depois a divisão por 2, que dará 5 de resposta final. A seguir, uma tabela de algumas operações aritméticas:

Símb	Nome	Como funciona
$+$	adição	em $a+b$ o resultado é a adição entre a e b.
$-$	subtração	em $a-b$ o resultado é $a-b$.
$*$	multiplicação	Em $a*b$ o resultado é a multiplicação de a e b.
$/$	divisão real	Em a/b o resultado é um `float` contendo a divisão de a por b.
$//$	divisão inteira	Em $a//b$ o resultado é um inteiro contendo a divisão inteira de a por b.
%	Resto	Em $a\%b$ o resultado é o resto da divisão inteira de a por b, sendo que é um valor $0 \leq resto \leq b-1$.
$**$	Potência	Em $a**b$ o resultado de a^b.
abs	Módulo	Em $abs(a)$ o resultado é $-a$ se a é negativo e a se a é positivo.

Note a diferença entre as duas divisões

```
>>> 10/3
3.3333333333333335
>>> 10//3
3
>>> 10%3
1
```

5.4 Saída de Dados

Todo programa durante sua execução deve em princípio divulgar alguma coisa. Afinal para algo ele está sendo executado. Ainda que haja quase infinitas possíveis aplicações de programas, grande parte deles vai precisar dizer algo. Uma diferença importante entre Python 2 e Python 3 é que no 2, os parâmetros de print seguem o nome da função sem parênteses enquanto que os parênteses são obrigatórios no Python 3. Aliás este é um dos mecanismos para identificar se um programa originalmente foi escrito em 2 ou 3.

A maneira mais simples de obter o conteúdo de uma variável é apenas digitando seu nome seguido de ENTER. Veja

```
>>> a = 5
>>> b = a+2
>>> b
7
```

Uma maneira mais elaborada de saída é com o comando `print()`. É muito rico, com múltiplas opções, mas vamos começar lentamente. Na sua versão mais simples, o print admite uma lista de variáveis e eventualmente constantes a serem escritas na saída padrão quando o programa for executado. A saída padrão é um conceito importante em computação pessoal: trata-se do dispositivo que o computador usará para todas as saídas, quando nada for afirmado sobre ONDE fazer a saída. Em computação usa-se o conceito de *default* nestes casos. Então a saída *default* é conhecida como saída padrão e quase sempre é o monitor de vídeo. Vejam-se alguns exemplos

`print(a)`	Imprime o valor da variável a.
`print("o valor de a: ",a)`	Agora antes do valor da variável a, imprime-se a mensagem entre aspas. Ajuda a identificar as saídas
`print("a: ",a,"b: ",b)`	Imprime a e b identificando cada uma delas
`print("Desligue agora")`	Imprime apenas a mensagem
`print("Resultado ",x+22)`	Imprime a mensagem e depois o resultado de x+22. Note que o valor de x permanece inalterado

5.4.1 Opções da função print

Dentro do parênteses da função print, podem-se colocar algumas opções, a saber: `sep=` que indica qual o separador entre itens na função print. Por *default* é um espaço em branco, mas pode ser qualquer coisa. Veja os exemplos

```
>>> a=5
>>> b=6
>>> print(a,b)
5 6
>>> print(a,b,sep="#")
5#6
>>> print(a,b,sep="oba")
5oba6
>>>
```

Outra opção importante refere-se ao terminador da função. `end="
n"`. Por default, é uma ”nova linha”, que em programação é reconhecida pelo caracter ”

n". Mas, pode ser qualquer outra coisa. Chamar `print()` sem nenhum parâmetro, é o mesmo que chamar print com o parêmtro end. veja no exemplo

```
print("ufa",end="#")
ufa#
```

Finalmente, a terceira opção que vamos estudar é `file="nome"`. Ela indica para onde vai ser mandado o resultado de print. Se nada é dito, ele sempre vai para `sys.stdout` que é o nome oficial da saída padrão. Mas, o print pode ser mandado para qualquer lugar. Veja no exemplo

```
>>> saida=open("c:/p/python/saida","w")
>>> print("oba",file=saida)
>>> saida.close()
>>> saida=open("c:/p/python/saida","w")
>>> a
5
>>> print(a,file=saida)
>>> saida.close()
```

5.5 Entrada de dados

A grande maioria dos algoritmos (programas) precisam obter algum dado do operador antes de realizar o processamento. O comando básico aqui é `input`. Seu formato sempre será

```
variavel = input("mensagem")
```

O que acontece aqui:

- A mensagem citada no comando é impressa na saída padrão
- O programa é interrompido neste ponto
- depois que o usuário digitar alguma coisa e teclar `ENTER`, o que foi digitado é colocado na variável citada no comando
- O processamento segue sem mais interrupções.

Um detalhe importante é que os conteúdos SEMPRE serão carregados no formato de string. Sem transformações, estas variáveis não podem ser usadas, por exemplo, em computações numéricas. Para tanto elas deverão ser transformadas usando (entre outras) as funções: `int` converte algo para inteiro, `float` converte algo para flutuante, e já que estamos estudando conversões, tem-se também `str` que converte algo para string. Veja alguns exemplos

```
>>> idade=input("entre com a idade ")
entre com a idade 33
>>> idade
'33'
>>> idade=int(input("entre com a idade "))
entre com a idade 33
>>> idade
33
```

Perceba que no primeiro caso (sem a transformação `int`) a variável idade é do tipo string, o que se pode perceber nas aspas que são mostradas ('33'). No segundo caso, em que há a transformação de string para inteiro (representada pela chamada à função `int`, o conteúdo da variável é um número e portanto pode ser operada aritmeticamente.

5.5.1 Um programa completo usando input e print

Seja um programa para pedir o raio de um círculo e imprimir a área do mesmo usando a fórmula da geometria

$$S = \pi \times R^2$$

```
def area():
  a=float(input("Informe raio"))
  s=3.1415 * a
  print("a área é: ",s)
area()
```

Mais um exemplo

```
def prim2(n):
  import math
  if n==1:
    return False
  if n<4:
    return True
  if n%2==0:
    return False
  if n<9:
    return True
  if n%3==0:
    return False
  r=math.floor(math.sqrt(n))
  f=5
  while f<=r:
    if n%f==0:
      return False
    if (n%(f+2))==0:
      return False
    f=f+6
  return True

def enesprim(qual):
  count=1
  candidato=1
  while count<qual:
    candidato=candidato+2
    if prim2(candidato):
      count=count+1
  return(candidato)

aa=int(input("Informe qual a ordem do primo "))
bb=enesprim(aa)
print(bb)
```

Neste exemplo primeiro define-se a função prim2 (bastante otimizada) que recebendo um valor qualquer informa se ele é ou não é primo. Depois define-se a função enesprim que recebe um n[umero inteiro x e devolve o x-ésimo número primo. Finalmente, o script pede um número qualquer em aa, e com ele calcula o aa-ésimo número primo, e o joga em bb. Depois este número é impresso.

5.6 Variáveis Lógicas

Uma outra família de variáveis são as variáveis lógicas, geralmente obtidas como resultado de uma comparação ou pergunta. Essas variáveis só podem ter 2 valores que são Verdadeiro (em Python `True`) e Falso (`False`).

Usualmente, estas variáveis são obtidas ao operar condições. As principais condições em Python são

Símb	Nome	Como funciona
==	Igual	Em $a == b$, a resposta será True se a for igual a b e False em caso contrário.
>	Maior	Em $a > b$ a resposta será True se a for maior do que b. Será False em caso contrário.
<	Menor	Em $a < b$ a resposta será True se a for menor do que b. Será False em caso contrário.
>=	Maior ou igual	Em $a >= b$ a resposta será True se a for maior do que ou for igual a b. Será False em caso contrário.
<	Menor ou igual	Em $a <= b$ a resposta será True se a for menor do que ou igual a b. Será False em caso contrário.
! =	Diferente	Também conhecida como não igual, ela responde o contrário da operação igual ==.

5.6.1 Expressões Lógicas

Quando há necessidade de expressar condições compostas, as operações acima podem ser conectadas usando-se 3 conectores lógicos que são:

Símb	Nome	Como funciona
and	E	Em $c1$ *and* $c2$, o resultado será verdadeiro se ambas, $c1$ e $c2$ forem verdadeiras e Falso, senão
or	OU	Em $c1$ *or* $c2$, o resultado será verdadeiro se $c1$ for verdadeiro, ou se $c2$ for verdadeiro ou se ambas forem verdadeiras. Será falso em caso contrário.
not	NÃO	Em *not* $c1$ o resultado será verdadeiro se $c1$ for falsa e será falsa se $c1$ for verdadeiro. Note que este operador é unário (só se aplica a uma condição).

Deve-se lembrar que estas operações e conectores estão definidas na matemática, particularmente na lógica de predicados. Lá seus símbolos são: $\wedge$ para o conectivo E, $\vee$ para o conecttivo OU e finalmente o símbolo $\sim$ para a negação lógica.

Capítulo 6

Comandos condicionais

Uma característica onipresente em linguagens de programação é a capacidade de executar ou não um conjunto de instruções a depender de uma variável lógica. Dizendo de outra maneira, pode-se comandar uma condição qualquer e se ela for verdadeira (que no Python é `True`) então um conjunto de instruções será executado. Se a condição for falsa (em Python `False`), ou nada é executado, ou também pode ocorrer, um segundo conjunto alternativo de instruções é executado.

O comando condicional por excelência em Python é `if` e ele têm o seguinte formato

```
if condição:
   comando_1
   comando_2
   ...
comando_n
```

Para escrever o comando, deve-se substituir a palavra condição por alguma condição relacional ou lógica, que ao final informe um valor `True` ou `False`. A regra é que os comandos 1, 2, ... só serão executados se a condição for `True`. Já ao final do bloco, o comando n será executado sempre independentemente da condição. Quem determina se o comando está ou não vinculado ao if é o espacejamento no início da linha, chamada em programação de indentação.

Em outras linguagens, usam-se caracteres delimitadores para os blocos de comando ({ e } em C, Java e C++; *begin* e *end* em Pascal e Delphi), mas em Python consoante com a proposta de limpeza da linguagem, usam-se as margens para determinar as condições.

Como tudo na vida isto tem vantagens e desvantagens. Começando pelas primeiras, o código fica limpo e legível. Quem sempre exigia dos programadores que respeitassem as margens, agora ganha um aliado importante: se as margens forem bagunçadas o programa não funcionará bem. Além disso economizam-se caracteres dispensáveis.

Já a desvantagem decorre principalmente da inabilidade de muitos ambientes para manusear este tipo de arquivo. Muitos editores se arvoram em trocar um certo número de brancos por um ou mais caracteres de tabulação, e fazer isto é receita quase certa para problemas de operação dos programas. Além disso, muitos gerenciadores de download deixam de respeitar essas margens, fazendo com o que o recebedor de um programa fonte Python tenha sempre que revisar com cuidado o resultado de descargas de código.

Veja alguns exemplos

```
a = 5
if a==6:
   print('ufa')
print('agora')
```

Primeiro a recebe 5. Daí 5 é comparado para igual com 6. A resposta é `False`. Com isso, os comandos identados abaixo não são executados (no caso print('ufa')). Finalmente é impresso 'agora' já que esta impressão independe do resultado da comparação.

Quem inicia um bloco indentado é a presença dos dois pontos : após a condição. Ele sinaliza ao editor em uso que a seguir deve-se respeitar a nova margem. Quando o programador quiser encerrar o bloco (e por conseguinte, a condição) ele terá que recuar manualmente o cursor até o início da linha.

Um segundo formato para o comando condicional, conhecido como condicional composta é aquele que apresenta uma condição e dois caminhos: um para o True e outro para o resultado False. Agora a separação entre um e outro é sinalizado pela palavra `else` que também deve ser seguida por dois pontos. Acompanhe no exemplo

```
a = 7
if a>3:
   print('oba')
else:
   print('ufa')
print('foi')
```

Acompanhe o raciocínio: Primeiro a é comparado com maior para 3. A resposta é `True` já que $7 > 3$ e com isso a impressão de 'oba' é feita. Como o resultado foi True, salta-se o bloco referente ao else:. Encerra o comando a impressão de 'foi' que independe do resultado da comparação. Veja agora um trecho parecido

```
a = 2
if a>3:
   print('oba')
else:
   print('ufa')
print('foi')
```

Agora a comparação resulta False (já que $2 > 3$ é falso) e com isso a impressão de 'oba' é saltada e depois a impressão de 'ufa' acontece. Novamente 'foi' é impresso independentemente da comparação inicial.

6.1 Exercício

Nos exercícios a seguir, você deve simular o interpretador Python e descobrir qual valor vai ser impresso ao final. Considere todas as variáveis como sendo do tipo **inteiro**. Todos os códigos estão sintaticamente corretos. A sugestão , que **depois** de ter calculado o valor na ponta do lápis, você confira o resultado com um computador real.

Exercício 1

```
def AA():
   A=4
   B=5
   C=8
   D=6
   F=6
   G=6
   if F!=2:
      C=A+F
      if (D>=9)or(C!=4):
         F=D+B
      C=C-C
   else:
      D=D-B
      if (D==5)and(D<8):
         D=D-B
      else:
         if F<2:
            F=A+B
         else:
            F=C+D
         A=G-G
      B=A*C
   A=C+D
   print(G+A+B-D)
AA()
```

Exercício 2

```
def AA():
   A=9
   B=5
   C=9
   D=2
   F=3
   G=8
   if B>5:
      if (G>9)or(D<=5):
         C=C+D
      if D==6:
         C=G-G
```

```
      C=G*A
   else:
      G=G*C
      if (F<2)and(not(C<4)):
         A=G*F
         if C<=6:
            B=F*C
      else:
         if (C>5)or(not(G==3)):
            D=D+A
         else:
            A=F-B
         G=C-B
      B=G+D
   print(A+F+G-D)
AA()
```

Exercício 3

```
def AA():
   A=4
   B=7
   C=6
   D=2
   F=7
   G=9
   if C!=6:
      G=D+B
   if (not(G==4))and(not(B>5)):
      if B!=6:
         G=G-A
   G=G-C
   if (D!=6)or(not(A>4)):
      B=G*B
      if A<5:
         A=C*C
      else:
         G=C*B
   else:
      if (C!=9)or(B<=8):
         C=C*F
      else:
         A=B*F
      G=B*B
   C=B*C
   print(G+F+B-A)
AA()
```

Exercício 4

```
def AA():
   A=9
   B=7
   C=6
   D=9
   F=5
   G=6
   if A<6:
      C=F*A
   if (C<=7)and(A>6):
      if D==9:
         D=D+F
      else:
```



```
            if (F<3)or(not(F>2)):
               B=B*A
            else:
               if C>=2:
                  G=C-B
            A=A+G
         G=F+F
      F=B-F
      if (not(B<=8))and(not(A<2)):
         G=G+A
         if A!=8:
            C=F-A
         else:
            G=F-B
      else:
         if (D!=8)and(not(C==4)):
            C=G+G
         else:
            A=D-A
         G=F+G
      C=G*D
      print(A+F+C-D)
AA()
```

Exercício 5

```
def AA():
   A=9
   B=7
   C=6
   D=3
   F=4
   G=7
   if G>4:
      if (B<=9)or(F<5):
         if D!=4:
            if (not(A<=5))or(A>4):
               B=G+F
            else:
               G=B-F
            G=A+B
         else:
            B=F-D
         G=G*D
      else:
         C=A+B
      F=G*C
   else:
      if G>2:
         if (A>=5)or(C!=3):
            if B<5:
               G=F*B
            else:
               D=F*G
         else:
            F=A+A
      F=C*C
   A=G+D
   print(D+F+G-A)
AA()
```

Exercício 6

```
def AA():
   A=9
   B=5
   C=5
   D=5
   F=6
   G=8
   if G>6:
      A=F-F
      if (D==2)and(not(G==9)):
         D=F+B
      B=B*D
   else:
      C=D+F
      if (G==8)or(not(D<7)):
         C=D+F
      else:
         if B<2:
            F=D-F
         else:
            F=B+B
         G=F*C
      B=F+D
   C=F+B
   print(C+A+F-B)
AA()
```

Exercício 7

```
def AA():
   A=4
   B=7
   C=7
   D=9
   F=2
   G=5
   if C>=5:
      if (A<=3)and(G!=2):
         F=A-G
      if A<=9:
         A=F*D
      A=C-A
   else:
      B=D*G
      if (D<5)or(A!=4):
         B=B-G
         if B>=5:
            F=D+G
      else:
         if (A==4)or(G>=4):
            D=C*B
         else:
            F=C*G
         F=F*G
      C=A+A
   print(G+A+C-D)
AA()
```

Exercício 8

```
def AA():
   A=4
   B=5
```

```
   C=6
   D=2
   F=9
   G=5
   if C>6:
      G=C*B
   if (F<5)and(A<6):
      if A>9:
         G=G-C
   C=B+F
   if (D>=2)or(C<=2):
      C=B+A
      if D>=7:
         B=B*D
      else:
         D=B*F
   else:
      if (A<=2)or(G<4):
         D=F+A
      else:
         A=D*D
      A=F+B
   A=G*G
   print(B+C+D-A)
AA()
```

Respostas

1.	2.	3.	4.	5.	6.	7.	8.
11	5	-5	165	360	12	-8	34

Capítulo 7

Repetições

As repetições são estruturas da linguagem que asseguram a execução repetida de trechos de comandos, até que uma determinada condição for satisfeita. Note que este conceirto faz parte do noasso dia a dia.

Por exemplo, uma caixa dágua posta a encher o fará até que atinja a capacidade total. Um forno assando um bolo deverá operar até que o bolo esteja assado, uma esteira carregadeira de um navio opera até que o depósito do navio esteja cheio ou enquanto o silo abastacedor tiver conteúdo, parando na condição que primeiro for atingida, um pneu de carro receberá ar comprimido até que uma certa pressão seja atingida e a água do café vai esquentar até ferver ou um pouco antes.

O que não falta na nossa vida são operações repetitivas que ocorrem até que uma condição seja atendida, ou o que praticamente é a mesma coisa, enquanto certa condição perdurar.

7.1 while

Os comandos básicos em Python são `while` que pode ser traduzido por enquanto. Seu formato é

```
while <condição>:
   comando_1
   comando_2
   ...
   comando_i
comando_n
```

Quando ele for encontrado será assim interpretado: primeiro a condição associada ao while será avaliada. Se ela for falsa, ocorre um desvio desprezando todo o bloco indentado abaixo. No exemplo acima, se a condição for falsa, o próximo comando a ser executado é o $comando_n$. Agora, se a condição é verdadeira, o bloco indentado é executado. Quando ele acabar, há um retorno (daí a repetição) ao início do bloco e a reavliação da condição. Enquanto ela for verdadeira, o bloco fica em loop.

Logo, aqui vale uma advertência: quando usar um *while* garanta que em algum momento do bloco, a condição seja tornada falsa, sob pena do seu programa nunca mais sair daí.

Por exemplo, seja somar os 33 primeiros números ímpares:

```
i=1
q=0
p=1
while i<34:
   q=q+p
   p=p+2
   i=i+1
print(q)
```

O programa acima vai imprimir 1089, que é a soma procurada.

7.2 break

Um comando que permite modificar o funcionamento do while é o break. Normalmente ele é colocado dentro de uma condição, embora isto não seja obrigatório. Quando break é interpretado ele faz com que o ciclo do while, na verdade qualquer ciclo, seja interrompido, forçando uma saída do bloco indentado.

Exemplo, seja um programa para calcular o triplo de um número digitado. O programa deve operar até ser digitado um número negativo, que significará fim de processamento.

```
a=1
while 1==1:
  if a<0:
    break
  a=int(input("informe o valor "))
  print(3*a)
print('adeus')
```

Note que como vai-se usar o *break* para a saída do laço, escreve-se uma condição que sempre é verdadeira (conhecida na lógica como uma tautologia) , que neste caso é $1 == 1$, que, por óbvio, é sempre verdadeira.

Eis um trecho da execução desse programa:

```
========== RESTART: C:/Python3/teste.py ===========
informe o valor 2
6
informe o valor 33
99
informe o valor 8
24
informe o valor -1
-3
adeus
>>>
```

7.2.1 Repetição aninhada

Quando se cria um bloco de repetição, dentro do bloco, o Python admite qualquer comando, inclusive um outro comando de repetição. Então surgem os blocos aninhados que são muito importantes em programação. Acompanhe o exemplo

```
i=1
while i<=3:
   j=10
   while j<=30:
      j=j+10
      print (i,j)
   i=i+1
print('acabou')
```

que vai gerar a seguinte saída

```
(1, 20)
(1, 30)
(1, 40)
(2, 20)
(2, 30)
(2, 40)
(3, 20)
(3, 30)
(3, 40)
acabou
```

7.3 Continue

Este comando também serve para interromper um ciclo de processamento, mas diferentemente de `break` que encerra inapelavelmente o laço, este comando `continue` encerra o processamento d euma interação e avança para a próxima interação. Acompanhe a diferença entre os dois comandos

```
i=3
while i<10:
   print(i,' ')
   if i%7==0:
      break
   i=i+1
print('acabou')
```

Esta execução teve o seguinte resultado:

```
3
4
5
6
7
acabou
```

Agora acompanhe o seguinte exemplo usando continue

```
i=3
while i<10:
   print(i,' ')
   if i%7==0:
      continue
   i=i+1
print('acabou')
```

Neste caso, não ponha para rodar, pois o programa vai entrar em laço infinito. Veja porque: O processamento começa com i=3, 4, 5 r 6. Quando o valor 7 for atingido, o continue vai comandar um retoino à próxima interação, mas sem alterar o valor de i (já que esta alteração está depois do `continue`. Aí ela vai imprimir 7,7,7,7,.... e não vai parar nunca mais.

7.4 For

O For é o grande interador do Python. Ele é uma expansão do comando `for` do C++. Lá no C++ o for funciona como se fosse uma simplificação do while, mas aqui ele é mais do que isso. A mudança se deve ao conceito de interador. Numa rápida explicação um interador é uma generalização do conceito de índice. Ele é uma generalização já que o índexador só se aplica a conjuntos homogêneos indexados (conhecidos como arrays) e o interador se aplica a qualquer coleção de objetos Python. Pode ser um array, mas pode ser também registros em um arquivo, itens em um conjunto, etc etc.

Veja um exemplo radical disso

```
s="Curitiba - Paraná - Brasil"
qtd=0
for c in s:
  if c=="a":
    qtd=qtd+1
print("achei ",qtd," a")
```

Ao executar o código acima, ele vai responder `achei 4 a"`. Note que são 4 letras 'a', sendo que o 'a' acentuado de Paraná, não conta, já que é outro caracter.

7.5 Exercícios

Nos exercícios a seguir, você deve simular o interpretador Python e descobrir qual valor vai ser impresso ao final. Todos os códigos estao sintaticamente corretos. A sugestão é que **depois** de ter calculado o valor na ponta do lápis, você confira o resultado com um computador real.

Exemplos

Seja o programa

```
def AA():
   A=5
   B=5
   C=3
   if C>=18:
      while B<=18:
         C=C-3
         B=B+4
   else:
      C=C+4
      B=B+2
   print(C+C+B)
AA()
```

Cujo resultado é 21 e seja o seguinte código

```
def AA():
    A=4
    B=3
    C=6
    D=1
    C=C+6+3
    while C>6:
        B=D+7
        while A<=14:
            D=B+4
            A=A+2
        D=B+19
        C=C-1
    print(C+B+B)
AA()
```

cujo resultado é 438. Seja agora o programa

```
def AA():
    A=7
    B=3
    C=5
    D=6
    A=A+5+3
    while A>=4:
        B=D-4
        if A>=10:
            while C<=13:
                C=C+6
            B=D+3
        else:
            D=D-13
        if D>4:
            B=B+5
        else:
            B=D-5
        A=A-2
    print(B+B)
AA()
```

cujo resultado é -76.

Exercício 1

```
def AA():
    A=7
    B=6
    C=7
    while C<16:
        A=B+5
        B=A+5
        C=C+2
    print(A+B+C)
AA()
```

Informe no quadro próprio o valor que foi impresso.

Exercício 2

```
def AA():
    A=6
    B=2
    C=3
```

```
    if C<=20:
        while B<=20:
            C=A+5
            B=B+4
    else:
        A=C-8
        B=B+2
    print(C+A+B)
AA()
```

Exercício 3

```
def AA():
    A=8
    B=2
    C=2
    if B!=16:
        while C<16:
            B=B+8
            C=C+2
    else:
        while C<13:
            A=B+7
            C=C+2
    print(B+B+C)
AA()
```

Exercício 4

```
def AA():
    A=2
    B=1
    C=8
    D=6
    A=A+8+1
    while A>=5:
        C=D-5
        while B<=16:
            C=D-1
            B=B+6
        C=D-16
        A=A-1
    print(B+C+B)
AA()
```

Exercício 5

```
def AA():
    A=4
    B=2
    C=1
    D=5
    B=B+1+2
    while B>4:
        C=A+5
        if A>=9:
            while A<=12:
                C=C+2
                A=A+6
            C=D-16
        else:
            C=C-9
        B=B-4
```



```
   print(B+C)
AA()
```

Exercício 6

```
def AA():
   A=3
   B=7
   C=8
   D=1
   A=A+8+7
   while A>5:
      B=B+3
      if B>=6:
         while D<=16:
            D=D+5
         B=C-7
      else:
         C=B+15
      if D>5:
         C=C+8
      else:
         B=C+10
      A=A-2
   print(B+C)
AA()
```

Exercício 7

```
def AA():
   A=6
   B=2
   C=1
   D=3
   C=C+1+2
   while C>=6:
      A=C+3
      if D>=9:
         while D<14:
            D=D+4
         B=B-3
      else:
         A=A+12
      if D<8:
         B=B+6
      else:
         A=A+9
      C=C-2
   print(A+B)
AA()
```

Exercício 8

```
def AA():
   A=5
   B=7
   C=8
   D=4
   D=D+8+7
   while D>=3:
      A=D+3
      if B>5:
         while B<=19:
```

```
            C=A+2
            B=B+1
         C=C-18
      else:
         A=C-10
      D=D-4
   print(C+D)
AA()
```

Exercício 9

```
def AA():
   A=8
   B=2
   C=7
   D=5
   D=D+7+2
   while D>7:
      A=B+1
      while B<=12:
         C=C-1
         B=B+1
      C=C+12
      D=D-1
   print(C+B+C)
AA()
```

Exercício 10

```
def AA():
   A=1
   B=2
   C=6
   if A<=15:
      while B<15:
         A=A-7
         B=B+4
   else:
      while B<16:
         A=C+1
         B=B+2
   print(A+A+B)
AA()
```

Respostas

```
    1   124    39   132    28     1   119     8   -67   173   -36
```


Capítulo 8

Listas

Uma lista em Python é uma coleção de coisas. Em outras linguagens, essas coisas precisam ser de mesmo tipo, mas em Python, não. Podem ser coisas bem heterogêneas.

Cria-se uma lista em python usando-se os delimitadores colchetes. Uma lista é criada assim:

```
>>> a=[]
```

Criou-se aqui uma lista de nome `a`, inicialmente vazia. Para incluir coisas na lista, usa-se o método `append`, veja como

```
>>> a=[]
>>> a.append(1)
>>> a
[1]
>>> a.append('oi')
>>> a
[1, 'oi']
```

Outro jeito de criar uma lista é usando a função `list` como em

```
>>> b=list(range(6))
>>> b
[0, 1, 2, 3, 4, 5]
```

Note que o primeiro elemento da lista pode ser acessado para leitura fazendo-se

```
>>> a[0]
1
```

E podem ser alterados, como em

```
>>> a[0]='bacana'
>>> a
['bacana', 'oi']
```

Veja mais alguns exemplos, e estude-os...

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Os índices para as listas começam em 0 e crescem para a direita quando de usam números inteiros positivos. A função `len(lista)` devolve o tamanho da lista. Índices negativos vem da direita para a esquerda. `lista[-1]` devolve o último elemento da lista, `lista[-2]` o penúltimo e assim por diante. Veja isso tudo:

```
>>> x=[10,20,30,'oi','viva',4.5,99]
>>> x
[10, 20, 30, 'oi', 'viva', 4.5, 99]
>>> x[0]
10
>>> x[1]
20
>>> len(x)
7
>>> x[-1]
99
>>> x[-2]
4.5
```

8.1 Fatiamento

É uma operação fundamental em Python, pode ser usada em listas, tuplas, strings... É muito usada em MatLab, mas sinceramente eu não consigo dizer quem é o ovo e quem é a galinha, isto quem veio antes. Tanto faz. Uma fatia é uma especificação envolvendo um ou dois "dois pontos" e ela extrai um pedaço de uma lista, tupla ou string. A especificação é

```
[começo da fatia:final da fatia:incremento]
```

começo o começo da fatia é o índice do seu início, sempre relativo a zero. Se nada for escrito aqui o valor *default* é 0.

final o final da fatia é o próximo número ALÉM do final. Ou seja este número é compatível com o início da numeração em zero. O valor *default* é o tamanho da lista ou `len(lista)`.

incremento É um valor muitas vezes omitido (e neste caso o valor *default* é 1), mas quando presente ele indica quanto somar ao valor atual para obter o próximo valor.

Acompanhe alguns exemplos

```
>>> y=[11,21,31,41,51]
>>> y
[11, 21, 31, 41, 51]
>>> y[0:3]
[11, 21, 31]
>>> y[:3]
[11, 21, 31]
>>> y[2:]
[31, 41, 51]
>>> y[1:3:2]
[21]
>>> y[1:-1]
[21, 31, 41]
>>> y[::2]
[11, 31, 51]
>>> y[::-1]
[51, 41, 31, 21, 11]
>>> y[::-2]
[51, 31, 11]
>>> y[:3]
[11, 21, 31]
>>> y[3:]
[41, 51]
```

Da mesma forma que índices de string, índices de lista começam em 0, listas também podem ser concatenadas, fatiadas e multiplicadas:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Todas as operações de fatiamento devolvem uma nova lista contendo os elementos solicitados. Isto significa que o fatiamento a seguir retorna uma cópia rasa (shallow copy) da lista:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Diferentemente de strings, que são imutáveis, é possível alterar elementos individuais de uma lista:

```
>>>
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Atribuição à fatias também é possível, e isso pode até alterar o tamanho da lista ou remover todos os itens dela:

```
>>>
>>> # Substituir alguns itens:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remover alguns:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Inserir alguns:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Inserir uma cópia da própria lista no início
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Limpar a lista: substituir todos os items por uma lista vazia
>>> a[:] = []
>>> a
[]
```

A função embutida len() também se aplica a listas:

```
>>>
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

É possível aninhar listas (criar listas contendo outras listas), por exemplo:

```
>>>
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Veja a seção 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Observe que no último exemplo, p[1] e q na verdade se referem ao mesmo objeto! Mais tarde retornaremos a semântica dos objetos.

8.1.1 fatiamento em arrays

Acompanhe

```
>>> from numpy import *
>>> a=array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
>>> a
array([[ 1,  2,  3],
```

```
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> a[1,1]
5
>>> a[1:3,1]
array([5, 8])
>>> a[1:3,1:3]
array([[5, 6],
       [8, 9]])
>>> a[1:3,1:3]=777
>>> a
array([[  1,   2,   3],
       [  4, 777, 777],
       [  7, 777, 777],
       [ 10,  11,  12]])
```

8.2 Mais

Este texto descreve alguns pontos já abordados, porém com mais detalhes, e adiciona outros pontos e foi retirado de http://turing.com.br/pydoc/2.7/tutorial/datastructures.html

8.3 Mais sobre listas

O tipo list possui mais métodos. Aqui estão todos os métodos disponíveis em um objeto lista:

`list.append(x)`

Adiciona um item ao fim da lista; equivale a `a[len(a):] = [x].`

`list.extend(L)`

Prolonga a lista, adicionando no fim todos os elementos da lista L passada como argumento; equivalente `a a[len(a):] = L.`

`list.insert(i, x)`

Insere um item em uma posição especificada. O primeiro argumento é o índice do elemento antes do qual será feita a inserção, assim a.insert(0, x) insere no início da lista, e a.insert(len(a), x) equivale a a.append(x).

`list.remove(x)`

Remove o primeiro item encontrado na lista cujo valor é igual a x. Se não existir valor igual, uma exceção ValueError é levantada.

`list.pop([i])`

Remove o item na posição dada e o devolve. Se nenhum índice for especificado, a.pop() remove e devolve o último item na lista. (Os colchetes ao redor do i indicam que o parâmetro é opcional, não que você deva digitá-los daquela maneira. Você verá essa notação com frequência na Referência da Biblioteca Python.)

`list.index(x)`

Devolve o índice do primeiro item cujo valor é igual a x, gerando ValueError se este valor não existe

`list.count(x)`

Devolve o número de vezes que o valor x aparece na lista.

`list.sort()`

Ordena os itens na própria lista in place.

`list.reverse()`

Inverte a ordem dos elementos na lista in place (sem gerar uma nova lista).

Um exemplo que utiliza a maioria dos métodos::

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

(N.d.T. Note que os métodos que alteram a lista, inclusive sort e reverse, devolvem None para lembrar o programador de que modificam a própria lista, e não criam uma nova. O único método que altera a lista e devolve um valor é o pop)

8.3.1 Usando listas como pilhas

Os métodos de lista tornam muito fácil utilizar listas como pilhas, onde o item adicionado por último é o primeiro a ser recuperado (política "último a entrar, primeiro a sair"). Para adicionar um item ao topo da pilha, use append(). Para recuperar um item do topo da pilha use pop() sem nenhum índice. Por exemplo:

```
>>> pilha = [3, 4, 5]
>>> pilha.append(6)
>>> pilha.append(7)
>>> pilha
[3, 4, 5, 6, 7]
>>> pilha.pop()
7
>>> pilha
[3, 4, 5, 6]
>>> pilha.pop()
6
>>> pilha.pop()
5
>>> pilha
[3, 4]
```

8.3.2 Usando listas como filas

Você também pode usar uma lista como uma fila, onde o primeiro item adicionado é o primeiro a ser recuperado (política "primeiro a entrar, primeiro a sair"); porém, listas não são eficientes para esta finalidade. Embora appends e pops no final da lista sejam rápidos, fazer inserts ou pops no início da lista é lento (porque todos os demais elementos têm que ser deslocados).

Para implementar uma fila, use a classe collections.deque que foi projetada para permitir appends e pops eficientes nas duas extremidades. Por exemplo:

```
>>> from collections import deque
>>> fila = deque(["Eric", "John", "Michael"])
>>> fila.append("Terry")    # Terry chega
>>> fila.append("Graham")   # Graham chega
>>> fila.popleft()          # O primeiro a chegar parte
'Eric'
>>> fila.popleft()          # O segundo a chegar parte
'John'
>>> fila                    # O resto da fila, em ordem de chegada
deque(['Michael', 'Terry', 'Graham'])
(N.d.T. neste exemplo são usados nomes de membros do grupo Monty Python)
```

8.4 Ferramentas de programação funcional

Existem três funções embutidas que são muito úteis para processar listas: filter(), map(), e reduce().

filter(função, sequência) devolve uma nova sequência formada pelos itens do segundo argumento para os quais funcao(item) é verdadeiro. Se a sequencia de entrada for string ou tupla, a saída será do mesmo tipo; caso contrário, o resultado será sempre uma lista. Por exemplo, para computar uma sequência de números não divisíveis por 2 ou 3:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
map(funcao, sequencia) aplica funcao(item) a cada item da sequência e devolve uma lista formac

>>> def cubo(x): return x*x*x
...
>>> map(cubo, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Mais de uma sequência pode ser passada; a função a ser aplicada deve aceitar tantos argumentos quantas sequências forem passadas, e é invocada com o item correspondente de cada sequência (ou None, se alguma sequência for menor que outra). Por exemplo:

```
>>> seq = range(8)
>>> def somar(x, y): return x+y
...
>>> map(somar, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

Se None for passado no lugar da função, então será aplicada a função identidade (apenas devolve o argumento recebido). Se várias sequências forem passadas, a lista resultante terá tuplas formadas pelos elementos correspondentes de cada sequência. Isso se parece com a função zip(), exceto que map() devolve uma lista com o comprimento da sequência mais longa que foi passada, preenchendo as lacunas com None quando necessário, e zip() devolve uma lista com o comprimento da mais curta. Confira:

```
>>> map(None, range(5))
[0, 1, 2, 3, 4]
>>> map(None, range(5), range(3))
[(0, 0), (1, 1), (2, 2), (3, None), (4, None)]
>>> zip(range(5), range(3))
[(0, 0), (1, 1), (2, 2)]
```

A função reduce(função, sequencia) devolve um único valor construído a partir da sucessiva aplicação da função binária (N.d.T. que recebe dois argumentos) a todos os elementos da lista fornecida, começando pelos dois primeiros itens, depois aplicando a função ao primeiro resultado obtido e ao próximo item, e assim por diante. Por exemplo, para computar a soma dos inteiros de 1 a 10:

```
>>> def somar(x,y): return x+y
...
>>> reduce(somar, range(1, 11))
55
```

Se houver um único elemento na sequência fornecida, seu valor será devolvido. Se a sequência estiver vazia, uma exceção será levantada.

Um terceiro argumento pode ser passado para definir o valor inicial. Neste caso, redução de uma sequência vazia devolve o valor inicial. Do contrário, a redução se inicia aplicando a função ao valor inicial e ao primeiro elemento da sequência, e continuando a partir daí.

```
>>> def somatoria(seq):
...     def somar(x,y): return x+y
...     return reduce(somar, seq, 0)
...
>>> somatoria(range(1, 11))
55
>>> somatoria([])
0
```

Não use a função somatória deste exemplo; somar sequências de números é uma necessidade comum, e para isso Python tem a função embutida sum(), que faz exatamente isto, e também aceita um valor inicial (opcional).

8.5 List comprehensions ou abrangências de listas

Uma list comprehension é uma maneira concisa de construir uma lista preenchida. (N.d.T. literalmente, abrangência de lista mas no Brasil o termo em inglês é muito usado; também se usa a abreviação listcomp)

Um uso comum é construir uma nova lista onde cada elemento é o resultado de alguma expressão aplicada a cada membro de outra sequência ou iterável, ou para construir uma subsequência cujos elementos satisfazem uma certa condição.

Por exemplo, suponha que queremos criar uma lista de quadrados, assim:

```
>>> quadrados = []
>>> for x in range(10):
...     quadrados.append(x**2)
...
>>> quadrados
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
Podemos obter o mesmo resultado desta forma:

quadrados = [x**2 for x in range(10)]
```

Isso equivale a quadrados = map(lambda x: x**2, range(10)), mas é mais conciso e legível.

Uma abrangência de lista é formada por um par de colchetes contendo uma expressão seguida de uma cláusula for, e então zero ou mais cláusulas for ou if. O resultado será uma lista resultante da avaliação da expressão no contexto das cláusulas for e if.

Por exemplo, esta listcomp combina os elementos de duas listas quando eles são diferentes:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
Isto equivale a:

>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note como a ordem dos for e if é a mesma nos dois exemplos acima.

Se a expressão é uma tupla, ela deve ser inserida entre parênteses (ex., (x, y) no exemplo anterior).

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # criar uma lista com os valores dobrados
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtrar a lista para excluir números negativos
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # aplicar uma função a todos os elementos
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # invocar um método em cada elemento
>>> frutas = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [arma.strip() for arma in frutas]
['banana', 'loganberry', 'passion fruit']
>>> # criar uma lista de duplas, ou tuplas de 2, como (numero, quadrado)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # a tupla deve estar entre parênteses, do contrário ocorre um erro
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # achatar uma lista usando uma listcomp com dois 'for'
```



```
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A abrangência de lista é mais flexível do que map() e pode conter expressões complexas e funções aninhadas, sem necessidade do uso de lambda:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

8.5.1 Listcomps aninhadas

A expressão inicial de uma listcomp pode ser uma expressão arbitrária, inclusive outra listcomp.

Observe este exemplo de uma matriz 3x4 implementada como uma lista de 3 listas de comprimento 4:

```
>>> matriz = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
A abrangência de listas abaixo transpõe as linhas e colunas:

>>> [[linha[i] for linha in matriz] for i in range(len(matriz[0]))]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Como vimos na seção anterior, a listcomp aninhada é computada no contexto da cláusula for seguinte, portanto o exemplo acima equivale a:

```
>>> transposta = []
>>> for i in range(len(matriz[0])):
...     transposta.append([linha[i] for linha in matriz])
...
>>> transposta
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
e isso, por sua vez, faz o mesmo que isto:

>>> transposta = []
>>> for i in range(len(matriz[0])):
...     # as próximas 3 linhas implementam a listcomp aninhada
...     linha_transposta = []
...     for linha in matriz:
...         linha_transposta.append(linha[i])
...     transposta.append(linha_transposta)
...
>>> transposta
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Na prática, você deve dar preferência a funções embutidas em vez de expressões complexas. A função zip() resolve muito bem este caso de uso:

8.6 O comando del

Existe uma maneira de remover um item de uma lista conhecendo seu índice, ao invés de seu valor: o comando del. Ele difere do método list.pop(), que devolve o item removido. O comando del também pode ser utilizado para remover fatias (slices) da lista, ou mesmo limpar a lista toda (que fizemos antes atribuindo uma lista vazia à fatia a[:]). Por exemplo:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
```

```
>>> a
[]
```

del também pode ser usado para remover totalmente uma variável:

```
>>> del a
>>> a
Traceback (most recent call last):
  ...
NameError: name 'a' is not defined
```

Referenciar a variável a depois de sua remoção constitui erro (pelo menos até que seja feita uma nova atribuição para ela). Encontraremos outros usos para o comando del mais tarde.

8.7 Tuplas e sequências

Vimos que listas e strings têm muitas propriedades em comum, como indexação e operações de fatiamento (slicing). Elas são dois exemplos de sequências (veja Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange). Como Python é uma linguagem em evolução, outros tipos de sequências podem ser adicionados. Existe ainda um outro tipo de sequência padrão na linguagem: a tupla (tuple).

Uma tupla consiste em uma sequência de valores separados por vírgulas:

```
>>> t = 12345, 54321, 'bom dia!'
>>> t[0]
12345
>>> t
(12345, 54321, 'bom dia!')
>>> # Tuplas podem ser aninhadas:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'bom dia!'), (1, 2, 3, 4, 5))
```

Como você pode ver no trecho acima, na saída do console as tuplas são sempre envolvidas por parênteses, assim tuplas aninhadas podem ser lidas corretamente. Na criação, tuplas podem ser envolvidas ou não por parênteses, desde que o contexto não exija os parênteses (como no caso da tupla dentro a uma expressão maior).

Tuplas podem ser usadas de diversas formas: pares ordenados (x, y), registros de funcionário extraídos uma base de dados, etc. Tuplas, assim como strings, são imutáveis: não é possível atribuir valores a itens individuais de uma tupla (você pode simular o mesmo efeito através de operações de fatiamento e concatenação; N.d.T. mas neste caso nunca estará modificando tuplas, apenas criando novas). Também é possível criar tuplas contendo objetos mutáveis, como listas.

Um problema especial é a criação de tuplas contendo 0 ou 1 itens: a sintaxe usa certos truques para acomodar estes casos. Tuplas vazias são construídas por uma par de parênteses vazios; uma tupla unitária é construída por um único valor e uma vírgula entre parênteses (não basta colocar um único valor entre parênteses). Feio, mas funciona:

```
>>> vazia = ()
>>> upla = 'hello',    # <-- note a vírgula no final
>>> len(vazia)
0
>>> len(upla)
1
>>> upla
('hello',)
```

O comando t = 12345, 54321, 'hello!' é um exemplo de empacotamento de tupla (tuple packing): os valores 12345, 54321 e 'bom dia!' são empacotados juntos em uma tupla. A operação inversa também é possível:

```
>>> x, y, z = t
```

Isto é chamado de desempacotamento de sequência (sequence unpacking), funciona para qualquer tipo de sequência do lado direito. Para funcionar, é necessário que a lista de variáveis do lado esquerdo tenha o mesmo comprimento da sequência à direita. Sendo assim, a atribuição múltipla é um caso de empacotamento de tupla e desempacotamento de sequência:

8.8 Sets (conjuntos)

Python também inclui um tipo de dados para conjuntos, chamado set. Um conjunto é uma coleção desordenada de elementos, sem elementos repetidos. Usos comuns para sets incluem a verificação eficiente da existência de objetos e a eliminação de itens duplicados. Conjuntos também suportam operações matemáticas como união, interseção, diferença e diferença simétrica.

Uma pequena demonstração:

```
>>> cesta = ['uva', 'laranja', 'uva', 'abacaxi', 'laranja', 'banana']
>>> frutas = set(cesta)   # criar um conjunto sem duplicatas
>>> frutas
set(['abacaxi', 'uva', 'laranja', 'banana'])
>>> 'laranja' in frutas   # testar se um elemento existe é muito rápido
True
>>> 'capim' in frutas
False
>>>
>>> # Demonstrar operações de conjunto em letras únicas de duas palavras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # letras unicas em a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                             # letras em a mas não em b
set(['r', 'd', 'b'])
>>> a | b                             # letras em a ou em b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                             # letras tanto em a como em b
set(['a', 'c'])
>>> a ^ b                             # letras em a ou b mas não em ambos
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

N.d.T. A sintaxe de sets do Python 3.1 foi portada para o Python 2.7, tornando possível escrever 10, 20, 30 para definir set([10, 20, 30]). O conjunto vazio tem que ser escrito como set() ou set([]), pois sempre representou um dicionário vazio, como veremos a seguir. Também existe uma sintaxe para set comprehensions, que nos permite escrever x*10 for x in [1, 2, 3] para construir 10, 20, 30.

8.9 Dicionários

Outra estrutura de dados muito útil embutida em Python é o dicionário, cujo tipo é dict (ver Mapping Types — dict). Dicionários são também chamados de "memória associativa" ou "vetor associativo" em outras linguagens. Diferente de sequências que são indexadas por inteiros, dicionários são indexados por chaves (keys), que podem ser de qualquer tipo imutável (como strings e inteiros). Tuplas também podem ser chaves se contiverem apenas strings, inteiros ou outras tuplas. Se a tupla contiver, direta ou indiretamente, qualquer valor mutável, não poderá ser chave. Listas não podem ser usadas como chaves porque podem ser modificadas in place pela atribuição em índices ou fatias, e por métodos como append() e extend().

Um bom modelo mental é imaginar um dicionário como um conjunto não ordenado de pares chave-valor, onde as chaves são únicas em uma dada instância do dicionário. Dicionários são delimitados por chaves: , e contém uma lista de pares chave:valor separada por vírgulas. Dessa forma também será exibido o conteúdo de um dicionário no console do Python. O dicionário vazio é .

As principais operações em um dicionário são armazenar e recuperar valores a partir de chaves. Também é possível remover um par chave:valor com o comando del. Se você armazenar um valor utilizando uma chave já presente, o antigo valor será substituído pelo novo. Se tentar recuperar um valor usando uma chave inexistente, será gerado um erro.

O método keys() do dicionário devolve a lista de todas as chaves presentes no dicionário, em ordem arbitrária (se desejar ordená-las basta aplicar o a função sorted() à lista devolvida). Para verificar a existência de uma chave, use o operador in.

A seguir, um exemplo de uso do dicionário:

```
>>> d={}
>>> d
{}
>>> d[123]='oba' #a chave 123 corresponde a 'oba'
>>> d
```

```
{123: 'oba'}
>>> d[124]='fui' #a chave 124 a 'fui'
>>> d
{123: 'oba', 124: 'fui'}

>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

O construtor dict() produz dicionários diretamente a partir de uma lista de chaves-valores, armazenadas como duplas (tuplas de 2 elementos). Quando os pares formam um padrão, uma list comprehensions pode especificar a lista de chaves-valores de forma mais compacta.

```
>>>
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use uma list comprehension
{2: 4, 4: 16, 6: 36}
```

N.d.T. A partir do Python 2.7 também existem dict comprehensions (abrangências de dicionário). Com esta sintaxe o último dict acima pode ser construído assim x: x**2 for x in (2, 4, 6).

Mais adiante no tutorial aprenderemos sobre expressões geradoras, que são ainda mais adequados para fornecer os pares de chave-valor para o construtor dict().

Quando chaves são strings simples, é mais fácil especificar os pares usando argumentos nomeados no construtor:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

N.d.T. Naturalmente, por limitações da sintaxe, essa sugestão só vale se as chaves forem strings ASCII, sem acentos, conforme a regras para formação de identificadores do Python.

8.10 Técnicas de iteração

Ao percorrer um dicionário em um laço, a variável de iteração receberá uma chave de cada vez:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k in knights:
...     print k
...
gallahad
robin
```

Quando conveniente, a chave e o valor correspondente podem ser obtidos simultaneamente com o método iteritems().

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Ao percorrer uma sequência qualquer, o índice da posição atual e o valor correspondente podem ser obtidos simultaneamente usando a função enumerate():

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Para percorrer duas ou mais sequências simultaneamente com o laço, os itens podem ser agrupados com a função zip().

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
```

What is your name? It is lancelot. What is your quest? It is the holy grail. What is your favorite color? It is blue. N.d.T. O exemplo acima reproduz um diálogo do filme Monty Python - Em Busca do Cálice Sagrado. Este trecho não pode ser traduzido, exceto por um decreto real.

Para percorrer uma sequência em ordem inversa, chame a função reversed() com a sequência na ordem original.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

Para percorrer uma sequência de maneira ordenada, use a função sorted(), que retorna uma lista ordenada com os itens, mantendo a sequência original inalterada.

```
>>> cesta = ['uva', 'laranja', 'uva', 'abacaxi', 'laranja', 'banana']
>>> for fruta in sorted(cesta):
...     print fruta
...
abacaxi
banana
laranja
laranja
uva
uva
>>> for fruta in sorted(set(cesta)):  # sem duplicações
...     print fruta
...
abacaxi
banana
laranja
uva
```

8.10.1 Métodos de dicionários

.clear()

Remove todos os itens do dicionário

```
>>> d={'chave':1, 'chave2': 2}
>>> d
{'chave': 1, 'chave2': 2}
>>> d.clear()
>>> d
{}
```

.get()

Retorna o valor associado à chave no dicionário. Se a chave não existe retorna `valor`. Se `valor` não é usado o padrão é `None`.

```
>>> d={'arroz':20, 'feijão':30}
>>> d.get('ovo')
>>> print(d.get('ovo'))
None
>>> print(d.get('ovo',33))
33
```

.keys()

Retorna uma lista com todas as chaves do dicionário.

8.11 Mais sobre condições

As condições de controle usadas em while e if podem conter quaisquer operadores, não apenas comparações.

Os operadores de comparação in e not in verificam se um valor ocorre (ou não ocorre) em uma dada sequência. Os operadores is e is not comparam se dois objetos são na verdade o mesmo objeto; isto só é relevante no contexto de objetos mutáveis, como listas. Todos os operadores de comparação possuem a mesma precedência, que é menor do que a prioridade de todos os operadores numéricos.

Comparações podem ser encadeadas: Por exemplo a < b == c testa se a é menor que b e também se b é igual a c.

Comparações podem ser combinadas através de operadores booleanos and e or, e o resultado de uma comparação (ou de qualquer outra expressão), pode ter seu valor booleano negado através de not. Estes possuem menor prioridade que os demais operadores de comparação. Entre eles, not é o de maior prioridade e or o de menor. Dessa forma, a condição A and not B or C é equivalente a (A and (not B)) or C. Naturalmente, parênteses podem ser usados para expressar o agrupamento desejado.

Os operadores booleanos and e or são operadores short-circuit: seus argumentos são avaliados da esquerda para a direita, e a avaliação para quando o resultado é determinado. Por exemplo, se A e C são expressões verdadeiras, mas B é falsa, então A and B and C não chega a avaliar a expressão C. Em geral, quando usado sobre valores genéricos e não como booleanos, o valor do resultado de um operador atalho é o último valor avaliado na expressão.

É possível atribuir o resultado de uma comparação ou outra expressão booleana para uma variável. Por exemplo:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Observe que em Python, diferente de C, atribuição não pode ocorrer dentro de uma expressão. Programadores C podem resmungar, mas isso evita toda uma classe de problemas frequentemente encontrados em programas C: digitar = numa expressão quando a intenção era ==.

8.12 Brincando com listas

Criando uma lista de números em sequência:

```
# python 2:
lista = range(100)

# python 3:
lista = list(range(100))
```

Criando uma lista com list comprehensions:

```
>>> lista = [x*2 for x in range(100)]
```

Percorrendo uma lista com for in:

```
>>> for x in lista:
    # faça algo com x
```

Percorrendo uma lista, obtendo os valores e seus índices:

```
>>> for indice, valor in enumerate(lista):
    print "lista[%d] = %d" % (indice, valor)
```

Percorrendo um pedaço de uma lista usando slicing:

```
>>> for x in lista[40:60]:
    # faça algo com x
```


Percorrendo uma lista de trás pra frente definindo o passo do slicing como -1:

```
>>> for x in lista[::-1]:
    # faça algo com x
Ou:
for x in reversed(lista):
    # faça algo com x
```

Percorrendo uma lista ordenada:

```
>>> for x in sorted(lista):
    # faça algo com x
```

Acessando o último elemento de uma lista com o índice -1:

```
>>> print lista[-1]
```

Copiando uma referência para uma lista:

```
>>> nova_ref = lista
>>> nova_ref is lista
True
```

Copiando de verdade uma lista:

```
>>> nova_lista = lista[:]
>>> nova_lista is lista
False
Ou, usando o módulo copy:
>>> import copy
>>> nova_lista = copy.copy(lista)
>>> nova_lista is lista
False
```

Ou caso lista contivesse listas aninhadas e quiséssemos fazer com que estas também fossem completamente copiadas, e não somente referenciadas, usaríamos a função deepcopy():

```
>>> nova_lista = copy.deepcopy(lista)
>>> nova_lista is lista
False
```

Embaralhando os elementos de uma lista (in-place):

```
>>> import random
>>> random.shuffle(lista)  # altera a própria lista
```

Obtendo uma amostra aleatória (de 10 elementos) de uma lista:

```
>>> print random.sample(lista, 10)
[729, 9025, 2401, 8100, 5776, 784, 1444, 484, 6241, 7396]
```

Obtendo um elemento aleatório de uma lista:

```
>>> random.choice(lista)
7744
```

Gerando uma lista com 10 números aleatórios, com valores entre 0 e 99:

```
>>> lista_aleatoria = random.sample(range(0, 100), 10)
```

Obtendo o maior elemento de uma lista:

```
>>> lista = range(0, 10)
>>> print max(lista)
9
O menor:
>>> print min(lista)
0
```

Pegando somente os elementos de índice par:

```
>>> print lista[::2]
[0, 2, 4, 6, 8]
Os de índice ímpar:
>>> print lista[1::2]
[1, 3, 5, 7, 9]
```

Somando todos os elementos de uma lista:

```
>>> print sum([1, 2, 3, 4])
10
```

Juntando duas listas, formando pares de elementos:

```
>>> lista = zip(range(0, 5), range(5, 10))
>>> print lista
[(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
```

Separando os elementos de uma lista de forma intercalada:

```
>>> lista = range(0, 10)
>>> intercaladas = lista[::2], lista[1::2]
>>> print intercaladas
([0, 2, 4, 6, 8], [1, 3, 5, 7, 9])
```

Transformando uma lista de strings em uma string CSV:

```
>>> lista = ["ola", "mundo", "aqui", "estamos"]
>>> csv_values = ','.join(lista)
>>> print csv_values
ola,mundo,aqui,estamos
```

Aplicando uma função (neste caso, anônima) a todos elementos de uma lista:

```
>>> lista = range(1, 11)
>>> print map(lambda x: x*-1, lista)
[-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
```

Filtrando os elementos de uma lista de acordo com um critério:

```
>>> def criterio(x): return x >= 0
>>> print range(-5, 5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> print filter(criterio, range(-5, 5))
[0, 1, 2, 3, 4]
```

Retirando os elementos cujo valor é zero (ou melhor, cujo valor é avaliado como False):

```
>>> print filter(None, range(-2, 2))
[-2, -1, 1]
```


Capítulo 9

Strings

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando print produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\
diversas linhas de texto assim como se faria em C.\n\
```

Strings podem ser indexadas; como em C, o primeiro caractere da string tem índice 0 (zero). Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Assim como na linguagem Icon, substrings podem ser especificadas através da notação de slice (fatiamento ou intervalo): dois índices separados por dois pontos.

```
>>> palavra[4]
'a'
>>> palavra[0:2]
'Aj'
>>> palavra[2:4]
'ud'
```

Índices de fatias têm defaults úteis; a omissão do primeiro índice equivale a zero, a omissão do segundo índice equivale ao tamanho da string sendo fatiada.:

```
>>> palavra[:2]     # Os dois primeiros caracteres
'Aj'
>>> palavra[2:]     # Tudo menos os dois primeiros caracteres
'udaZ'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> palavra[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]
'xjudaZ'
>>> 'Splat' + palavra[5]
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: s[:i] + s[i:] é igual a s.

Intervalos fora de limites são tratados "graciosamente" (N.d.T: o termo original "gracefully" indica robustez no tratamento de erros): um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]
'judaZ'
>>> palavra[10:]
''
>>> palavra[2:1]
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Por exemplo:

```
>>> palavra[-1]   # O último caractere
'Z'
>>> palavra[-2]   # O penúltimo caractere
'a'
>>> palavra[-2:]  # Os dois últimos caracteres
'aZ'
>>> palavra[:-2]  # Tudo menos os dois últimos caracteres
'Ajud'
```

Observe que -0 é o mesmo que 0, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]
'A'
```

Intervalos fora dos limites da string são truncados, mas não tente isso com índices simples (que não sejam fatias):

```
>>> palavra[-100:]
'AjudaZ'
>>> palavra[-100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Uma maneira de lembrar como slices funcionam é pensar que os índices indicam posições entre caracteres, onde a borda esquerda do primeiro caractere é 0. Assim, a borda direita do último caractere de uma string de comprimento n tem índice n, por exemplo:

```
 0   1   2   3   4   5   6
 +---+---+---+---+---+---+
 | A | j | u | d | a | z |
 +---+---+---+---+---+---+
-6  -5  -4  -3  -2  -1
```

A primeira fileira de números indica a posição dos índices 0...6 na string; a segunda fileira indica a posição dos respectivos índices negativos. Uma fatia de i a j consiste em todos os caracteres entre as bordas i e j, respectivamente.

Para índices positivos, o comprimento da fatia é a diferença entre os índices, se ambos estão dentro dos limites da string. Por exemplo, comprimento de palavra[1:3] é 2.

A função built-in (embutida) len() devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'
>>> len(s)
29
```

Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange Strings, e as strings Unicode descritas na próxima seção, são exemplos de sequências e implementam as operações comuns associadas com esses objetos.

9.0.1 String Methods

Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas. Eis alguns

.capitalize()

Transforma o primeiro caractere de uma string em maísculo. Ignora caracteres acentuados ou que não sejam letras.

```
>>> "curitiba".capitalize()
'Curitiba'
>>> "ágora".capitalize()
ágora
```

.center()

Centraliza a string usando espaços à esquerda e à direita, até atingir a largura.

```
>>> "paraná".center(20)
'       paraná       '
>>> "paraná".center(2)
'paraná'
```

.count()

Conta a quantidade de substrings dentro da string

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

.find()

Retorna a primeira ocorrência do substring dentro da string. Caso nada seja encontrado, retorna -1.

```
>>> "abracadabra".count("ab")
2
>>> "abracadabra".count("a")
5
```

Avaliadores

São eles: `.isalpha()`, `.isdecimal()`, `.isdigit()`, `.islower()`,`.isnumeric()`, `.isspace()`, `.istitle()`, `.isupper()`. Retornam `True` caso a string seja composta exclusivamente de alfanuméricos, decimais, dígitos, minúsculos, numéricos, espaços, títulos (a primeira letra de cada palavra em maiúsculo) e maiúsculas. Todos ignoram as letras acentuadas.

.join()

Concatena todos os elementos dentro de iterável usando a string especificada como concatenador. A string pode ser vazia. Os elementos do iterável devem ser string.

```
>>> a=['agua', 'fogo', 'xabu']
>>> ''.join(a)
'aguafogoxabu'
>>> '\n'.join(a)
'agua\nfogo\nxabu'
>>> print('\n'.join(a))
agua
fogo
xabu
```


.partition()

Divide a string em uma tupla com 3 elementos: a parte anterior à primeira ocorrência do `separador`, o separador e a parte posterior ao separador

```
>>> "curitibaparanabrasil".partition('ra')
('curitibapa', 'ra', 'nabrasil')
>>> "curitibaparanabrasil".partition('z')
('curitibaparanabrasil', '', '')
```

.replace()

Substitui n ocorrências da substring `antiga` pela `nova`. Se n não é fornecido, ele substitui todas.

```
>>> "curitibaparanabrasil".replace("ra","agora")
'curitibapaagoranabagorasil'
>>> "curitibaparanabrasil".replace("ra","zz",1)
'curitibapazznabrasil'
```

.split()

Divide a string transformando-a em uma lista usando um separador n vezes. Se o separador não é informado ele usa um espaço em branco.

```
>>> "curitibaparanabrasil".split('a')
['curitib', 'p', 'r', 'n', 'br', 'sil']
>>> "curitiba parana brasil".split()
['curitiba', 'parana', 'brasil']
```

Muitos outros

Aqui estão só alguns, talvez os mais úteis e/ou principais. Consulte a documentação da versão de Python que você está usando. Estão todos lá.

9.0.2 String Formatting Operations

As operações de formatação de strings antigas, que acontecem quando strings simples e Unicode aparecem à direita do operador % são descritas dom mais detalhes nesta seção.

Strings Unicode

Um novo tipo para armazenar textos foi introduzido: o tipo unicode. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja http://www.unicode.org/) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma "code page" (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software (comumente escrito como i18n porque internationalization é 'i' + 18 letras + 'n'). Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'
u'Hello World !'
```

O u antes das aspas indica a criação de uma string Unicode. Se você desejar incluir caracteres especiais na string, você pode fazê-lo através da codificação Unicode-Escape de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal 0x0020 (o espaço em branco) naquela posição.

Os outros caracteres são interpretados usando seus valores ordinais como valores ordinais em Unicode. Se você possui strings literais na codificação padrão Latin-1 que é usada na maioria dos países ocidentais, achará conveniente que os 256 caracteres inferiores do Unicode coincidem com os 256 caracteres do Latin-1.

Além dessas codificações padrão, Python oferece todo um conjunto de maneiras de se criar strings Unicode a partir de alguma codificação conhecida.

A função embutida unicode() dá acesso a todos os codecs Unicode registrados (COders e DECoders). Alguns dos codecs mais conhecidos são: Latin-1, ASCII, UTF-8, e UTF-16. Os dois últimos são codificações de tamanho variável para armazenar cada caractere Unicode em um ou mais bytes. (N.d.T: no Brasil, é muito útil o codec cp1252, variante estendida do Latin-1 usada na maioria das versões do MS Windows distribuídas no país, contendo caracteres comuns em textos, como aspas assimétricas “x” e ‘y’, travessão —, bullet • etc.).

Para converter uma string Unicode em uma string de 8-bits usando uma codificação específica, basta invocar o método encode() de objetos Unicode passando como parâmetro o nome da codificação destino. É preferível escrever nomes de codificação em letras minúsculas.

Se você tem um texto em uma codificação específica, e deseja produzir uma string Unicode a partir dele, pode usar a função unicode(), passando o nome da codificação de origem como segundo argumento.

9.0.3 Manipulação de bits

Embora seja algo meio fora de moda, eventualmente você vai precisar manusear bits em strings. Eis algumas funções

```
>>> bin(97)
'0b1100001'
>>> int('1100001',2)
97
>>> chr(97)
'a'
>>> ord('a')
97
>>> bin(ord('a'))
'0b1100001'
>>> chr(int('1100001',2))
'a'
>>> chr(int('01100001',2))
'a'
```


Capítulo 10

Funções

Podemos criar uma função que escreve a série de Fibonacci até um limite arbitrário:

```
>>> def fib(n):    # escrever série de Fibonacci até n
...     """Exibe série de Fibonacci até n"""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Agora invocamos a funçao que acabamos de definir:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A palavra reservada def inicia a definição de uma função. Ela deve ser seguida do nome da função e da lista de parâmetros formais entre parênteses. Os comandos que formam o corpo da função começam na linha seguinte e devem ser indentados.

Opcionalmente, a primeira linha lógica do corpo da função pode ser uma string literal, cujo propósito é documentar a função. Se presente, essa string chama-se docstring. (Há mais informação sobre docstrings na seção Strings de documentação.) Existem ferramentas que utilizam docstrings para produzir automaticamente documentação online ou para imprimir, ou ainda permitir que o usuário navegue interativamente pelo código. É uma boa prática incluir sempre docstrings em suas funções, portanto, tente fazer disto um hábito.

A execução de uma função gera uma nova tabela de símbolos, usada para as variáveis locais da função. Mais precisamente, toda atribuição a variável dentro da função armazena o valor na tabela de símbolos local. Referências a variáveis são buscadas primeiramente na tabela local, então na tabela de símbolos global e finalmente na tabela de nomes embutidos (built-in). Portanto, não se pode atribuir diretamente um valor a uma variável global dentro de uma função (a menos que se utilize a declaração global antes), ainda que variáveis globais possam ser referenciadas livremente.

Os parâmetros reais (argumentos) de uma chamada de função são introduzidos na tabela de símbolos local da função no momento da invocação, portanto, argumentos são passados por valor (onde o valor é sempre uma referência para objeto, não o valor do objeto). [1] Quando uma função invoca outra, uma nova tabela de símbolos é criada para tal chamada.

Uma definição de função introduz o nome da função na tabela de símbolos atual. O valor associado ao nome da função tem um tipo que é reconhecido pelo interpretador como uma função definida pelo usuário. Esse valor pode ser atribuído a outros nomes que também podem ser usados como funções. Esse mecanismo serve para renomear funções:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Conhecendo outras linguagens, você pode questionar que fib não é uma função, mas um procedimento, pois ela não devolve um valor. Na verdade, mesmo funções que não usam o comando return devolvem um valor, ainda que pouco interessante. Esse valor é chamado None (é um nome embutido). O interpretador interativo evita escrever None quando ele é o único resultado de uma expressão. Mas se quiser vê-lo pode usar o comando print:

```
>>> fib(0)
>>> print fib(0)
None
```

É fácil escrever uma função que devolve uma lista de números série de Fibonacci, ao invés de exibi-los:

```
>>> def fib2(n): # devolve a série de Fibonacci até n
...     """Devolve uma lista a com série de Fibonacci até n."""
...     resultado = []
...     a, b = 0, 1
...     while a < n:
...         resultado.append(a)    # veja mais adiante
...         a, b = b, a+b
...     return resultado
...
>>> f100 = fib2(100)    # executar
>>> f100                # exibir o resultado
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este exemplo, como sempre, demonstra algumas características novas:

O comando return termina a função devolvendo um valor. Se não houver uma expressão após o return, o valor None é devolvido. Se a função chegar ao fim sem o uso explícito do return, então também será devolvido o valor None. O trecho resultado.append(a) invoca um método do objeto lista resultado. Um método é uma função que "pertence" a um objeto e é chamada através de obj.nome_do_metodo onde obj é um objeto qualquer (pode ser uma expressão), e nome_do_metodo é o nome de um método que foi definido pelo tipo do objeto. Tipos diferentes definem métodos diferentes. Métodos de diferentes tipos podem ter o mesmo nome sem ambiguidade. (É possível definir seus próprios tipos de objetos e métodos, utilizando classes, veja em Classes) O método append() mostrado no exemplo é definido para objetos do tipo lista; ele adiciona um novo elemento ao final da lista. Neste exemplo, ele equivale a resultado = resultado + [a], só que mais eficiente.

10.1 Mais sobre definição de funções

É possível definir funções com um número variável de argumentos. Existem três formas, que podem ser combinadas.

10.1.1 Parâmetros com valores default

A mais útil das três é especificar um valor default para um ou mais parâmetros formais. Isso cria uma função que pode ser invocada com um número menor de argumentos do que ela pode receber. Por exemplo:

```
def confirmar(pergunta, tentativas=4, reclamacao='Sim ou não, por favor!'):
    while True:
        ok = raw_input(pergunta).lower()
        if ok in ('s', 'si', 'sim'):
            return True
        if ok in ('n', 'no', 'não', 'nananinanão'):
            return False
        tentativas = tentativas - 1
        if tentativas == 0:
            raise IOError('usuario nao quer cooperar')
        print reclamacao
```

Essa função pode ser invocada de várias formas:

- fornecendo apenas o argumento obrigatório: confirmar('Deseja mesmo encerrar?')
- fornecendo um dos argumentos opcionais: confirmar('Sobrescrever o arquivo?', 2) ou
- fornecendo todos os argumentos: confirmar('Sobrescrever o arquivo?', 2, 'Escolha apenas s ou n')

Este exemplo também introduz o operador `in`, que verifica se uma sequência contém ou não um determinado valor.

Os valores default são avaliados no momento a definição da função, e no escopo em que a função foi definida, portanto:

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
irá exibir 5.
```

Aviso importante: Valores default são avaliados apenas uma vez. Isso faz diferença quando o valor default é um objeto mutável como uma lista ou dicionário (N.d.T. dicionários são como arrays associativos ou HashMaps em outras linguagens; ver Dicionários).

Por exemplo, a função a seguir acumula os argumentos passados em chamadas subsequentes:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Esse código vai exibir:

```
[1]
[1, 2]
[1, 2, 3]
```

Se você não quiser que o valor default seja compartilhado entre chamadas subsequentes, pode reescrever a função assim:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
\subsection{Argumentos nomeados}
Funções também podem ser chamadas passando keyword arguments (argumentos nomeados) no formato
\begin{verbatim}
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

aceita um argumento obrigatório (voltage) e três argumentos opcionais (state, action, e type). Esta função pode ser invocada de todas estas formas:

```
parrot(1000)                                    # 1 arg. posicional
parrot(voltage=1000)                            # 1 arg. nomeado
parrot(voltage=1000000, action='VOOOOOM')       # 2 arg. nomeados
parrot(action='VOOOOOM', voltage=1000000)       # 2 arg. nomeados
parrot('a million', 'bereft of life', 'jump')   # 3 arg. posicionais
# 1 arg. positional e 1 arg. nomeado
parrot('a thousand', state='pushing up the daisies')
```

mas todas as invocações a seguir seriam inválidas:

```
parrot()                     # argumento obrigatório faltando
parrot(voltage=5.0, 'dead')  # argumento posicional depois do nomeado
parrot(110, voltage=220)     # valur duplicado para o mesmo argument
parrot(actor='John Cleese')  # argumento nomeado desconhecido
```

Em uma invocação, argumentos nomeados devem vir depois dos argumentos posicionais. Todos os argumentos nomeados passados devem casar com os parâmetros formais definidos pela função (ex. actor não é um argumento nomeado válido para a função parrot), mas sua ordem é irrelevante. Isto também inclui argumentos obrigatórios (ex.: parrot(voltage=1000) funciona). Nenhum parâmetro pode receber mais de um valor. Eis um exemplo que não funciona devido a esta restrição:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Quando o último parâmetro formal usar a sintaxe **nome, ele receberá um dicionário (ver Dicionários ou Mapping Types — dict [online]) com todos os parâmetros nomeados passados para a função, exceto aqueles que corresponderam a parâmetros formais definidos antes. Isto pode ser combinado com o parâmetro formal *nome (descrito na próxima subseção) que recebe uma tupla (N.d.T. uma sequência de itens, semelhante a uma lista imutável; ver Tuplas e sequências) contendo todos argumentos posicionais que não correspondem à lista da parâmetros formais. (*nome deve ser declarado antes de **nome.) Por exemplo, se definimos uma função como esta:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

Ela pode ser invocada assim:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

e, naturalmente, produziria:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note que criamos uma lista de chaves keys ordenando o resultado do método keys() do dicionário keywords antes de exibir seu conteúdo; se isso não fosse feito, os argumentos seriam exibidos em uma ordem não especificada.

10.1.2 Listas arbitrárias de argumentos

Finalmente, a opção menos usada possibilita que função seja invocada com um número arbitrário de argumentos. Esses argumentos serão empacotados em uma tupla (ver Tuplas e sequências). Antes dos argumentos em número variável, zero ou mais argumentos normais podem estar presentes.

```
def escrever_multiplos_items(arquivo, separador, *args):
    arquivo.write(separador.join(args))
```

10.1.3 Desempacotando listas de argumentos

A situação inversa ocorre quando os argumentos já estão numa lista ou tupla mas ela precisa ser explodida para invocarmos uma função que requer argumentos posicionais separados. Por exemplo, a função range() espera argumentos separados, start e stop. Se os valores já estiverem juntos em uma lista ou tupla, escreva a chamada de função com o operador * para desempacotá-los da sequência:

```
>>> range(3, 6)     # chamada normal com argumentos separados
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)    # chamada com argumentos desempacotados de uma lista
[3, 4, 5]
```

Da mesma forma, dicionários podem produzir argumentos nomeados com o operador **:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
```

```
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
```

– This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !

10.1.4 Construções lambda

Atendendo a pedidos, algumas características encontradas em linguagens de programação funcionais como Lisp foram adicionadas a Python. Com a palavra reservada lambda, pequenas funções anônimas podem ser criadas. Eis uma função que devolve a soma de seus dois argumentos:

```
lambda a, b: a+b.
```

Construções lambda podem ser empregadas em qualquer lugar que exigiria uma função. Sintaticamente, estão restritas a uma única expressão. Semanticamente, são apenas açúcar sintático para a definição de funções normais. Assim como definições de funções aninhadas, construções lambda podem referenciar variáveis do escopo onde são definidas (N.d.T isso significa que Python implementa closures, recurso encontrado em Lisp, JavaScript, Ruby etc.):

```
>>> def fazer_incrementador(n):
...     return lambda x: x + n
...
>>> f = fazer_incrementador(42)
>>> f(0)
42
>>> f(1)
43
```

10.1.5 Strings de documentação

A comunidade Python está convencionando o conteúdo e o formato de strings de documentação (docstrings).

A primeira linha deve ser um resumo curto e conciso do propósito do objeto. Por brevidade, não deve explicitamente se referir ao nome ou tipo do objeto, uma vez que estas informações estão disponíveis por outros meios (exceto se o nome da função for o próprio verbo que descreve a finalidade da função). Essa linha deve começar com letra maiúscula e terminar com ponto.

Se existem mais linhas na string de documentação, a segunda linha deve estar em branco, separando visualmente o resumo do resto da descrição. As linhas seguintes devem conter um ou mais parágrafos descrevendo as convenções de chamada ao objeto, seus efeitos colaterais, etc.

O parser do Python não remove a indentação de comentários multi-linha. Portanto, ferramentas que processem strings de documentação precisam lidar com isso, quando desejável. Existe uma convenção para isso. A primeira linha não vazia após a linha de sumário determina a indentação para o resto da string de documentação. (Não podemos usar a primeira linha para isso porque ela em geral está adjacente às aspas que iniciam a string, portanto sua indentação real não fica aparente.) Espaços em branco ou tabs "equivalentes" a esta indentação são então removidos do início das demais linhas da string. Linhas indentação menor não devem ocorrer, mas se ocorrerem, todos os espaços à sua esquerda são removidos. Para determinar a indentação, normalmente considera-se que um caractere tab equivale a 8 espaços.

Eis um exemplo de uma docstring multi-linha:

```
>>> def minha_funcao():
...     """Não faz nada, mas é documentada.
...
...     Realmente ela não faz nada.
...     """
...     pass
...
>>> print minha_funcao.__doc__
Não faz nada, mas é documentada.

Realmente ela não faz nada.
```

Capítulo 11

Objetos em Python

Historicamente o paradigma de objetos nasceu como uma das reações da comunidade de software ao problema cada vez mais importante da complexidade crescente do software. É só você olhar para os lados (bancos, companhias aéreas, governos, grandes corporações, hospitais, um carro moderno, etc etc) para ver que a humanidade literalmente lança seu futuro nas mãos do software e seja o que Deus quiser.

A idéia do objeto é encapsular (esconder, impedir que alguém indeterminado faça alterações, coisas desse estilo) dados e procedimentos de maneira a garantir a integridade de uns e de outros. Dito de outra maneira, o paradigma de objetos interpõe um intermediário (o objeto) entre os ativos tecnológicos (dados e procedimentos) e seus usuários (programas e programadores que os manipulam).

Isto é necessário já que é uma evidência empírica que boa parte dos bugs (erros) de programa derivam de efeitos colaterais de manipulação inadvertida de áreas comuns de dados dentro dos programas. Sobretudo, se não são os maiores erros em número, certamente são os mais difíceis de corrigir por (aparentemente) muito pouco terem a ver com as ações que os causaram. Em outras palavras quando a relação causa $\Leftrightarrow$ efeito de um erro é tênue ou escondida, sua correção demora e custa muito mais.

A explicação parece complexa e o é. Razão pela qual – na minha opinião – a metodologia de objetos se aplica maravilhosamente a sistemas grandes e/ou complexos, mas para aprender a programar, ou mesmo para aplicações simples é um exagero seu uso. É como matar um mosquito usando uma bala de canhão.

Mas aqui, de novo, a superioridade de Python: ele pode ser aprendido e usado o tempo todo como uma linguagem imperativa: como se viu este é o melhor jeito de aprender a programar. Quando (e se) lá na frente, você sentir falta das qualidades inerentes ao paradigma orientado ao objeto, *voilà*, não é que Python é 100% orientado ao objeto?

Dito isto, um objeto em Python é uma entidade que tem dados e procedimentos associados, e devidamente encobertos (encapsulados). Um objeto é definido em python pelo uso da cláusula *class*. Veja como:

```
class carro:
```

Note que a definição termina por `:` o que caracteriza que ela continua. A próxima coisa é definir um objeto python de nome `__init__`. Veja

```
  def __init__(self):
```

Este objeto é o *construtor* e ele é automaticamente chamado pelo python toda vez que um objeto é instanciado (definido e usado pela primeira vez). A variável *self* é o objeto propriamente dito. Note que o construtor é uma função, pelo que também termina por `:`.

Nesta função construtora definem-se os valores iniciais do objeto que passarão a fazer parte dele assim que o objeto for criado (instanciado). Acompanhe no exemplo completo

```
class carro:
  def __init__(self):
    self.kilo = 0
    self.marca = 'peugeot'
    self.portas = 5
```

Aqui definiu-se um objeto *carro* e disse-se que a kilometragem do carro é 0, a marca é peugeot e a quantidade de portas é 5.

Depois dessa definição, pode-se criar quantos objetos se quiserem, como em

```
carro1 = carro()
carro2 = carro()
carro2.portas = 3
```

Neste caso, há 2 carros: carro1 e carro2. Em particular, o carro2 tem um número diferente de portas, a saber, 3. Acompanhe

```
print(carro1.portas)
print(carro2.portas)
```

O python responderá 5 para a primeira impressão e 3 para a segunda.

11.1 Um exemplo de objeto: um formigueiro

A seguir uma prova de como é poderoso o conceito de objeto. Eis uma possível simulação de um formigueiro, que pode ser descrito como

- Uma coletividade de inúmeras (50, 100, 200 ? quantas ?) formigas
- As formigas são similares, isto é, todas têm o mesmo comportamento,
- Mas cada uma é um indivíduo separado das demais, e nesta implementação, uma não interfere na outra.

Se não se usasse a orientação a objeto, haveria que programar a coisa na unha, controlando para cada formiga seus dados, e lutando bravamente para que os dados de cada formiga não fossem (inadvertidamente) alterados por outra formiga. O software seria grande, complexo, sujeito a largos processos de depuração e correção.

Ao usar objetos, veja-se as vantagens adquiridas.

- Programa-se apenas 1 formiga. Não importa quantas formigas existam. Este enfoque permite que o comportamento dessa formiga seja tão complexo e rebuscado quanto se queira. Afinal é só uma formiga.
- Cada formiga enxerga seus dados e pode manipulá-los sem medo. Não há como uma formiga alterar dados de outra: lembre-se que no programa só há uma formiga.
- A criação de várias formigas é tarefa simplória: Definido o objeto, muitas formigas são criadas em um bloco for, que gera instâncias diferentes da mesma formiga. Agora, passar de 10 a 1000 formigas é apenas mudar o valor final de um laço for.
- O programa fica menor. Seus erros de implementação são sensivelmente menores e portanto sua implementação mais rápida.
- Os bugs misteriosos não aparecem.

```
# exemplo de objetos (formigas em um gramado)
import turtle, random
class formiga:
  def __init__(self,i): #aqui define-se o comportamento inicial da formiga
                        # um parâmetro passado é um número i
    self.t=turtle.Pen() #cria-se uma turtle de nome t no objeto e ela tem o método pen
    tcor=["violet","red","yellow","green","blue","brown","orange","pink","black",
          "purple","olive","fuchsia","blueviolet","plum",
          "coral","cornsilk","chartreuse","cyan","aqua","crimson"] #as diversas cores
#    ind=random.randint(0,7)
    self.t.color=tcor[i%len(tcor)]    #a cor é atribuida de maneira circular
    self.t.ondex=200-random.randint(0,399) #a formiga começa em x= -200..200 randomicamente
    self.t.ondey=200-random.randint(0,399) # idem para y
    self.t.direcao=random.randint(0,359)   # ganha uma direção randômica 0..360
    self.t.potencia=random.randint(1,11)   # ganha uma potência randômica 1..11
    self.t.penup()                         # levanta a caneta
    self.t.goto(self.t.ondex,self.t.ondey) # vai para onde ela está
    self.t.pendown()                       # abaixa a caneta

  def anda(self):  # como a formiga anda
    self.t.penup()                         # levanta a caneta
    self.t.goto(self.t.ondex,self.t.ondey) # vai para onde ela está
    self.t.pensize(self.t.potencia)        # associa a grossura do traço à potência
    self.t.pendown()                       # abaixa a caneta
    self.t.pencolor(self.t.color)          # associa a cor da caneta
    dist=random.randint(25,70)             # dist= randomica 25..70
    ndir=random.randint(0,359)             # ndir= direção randômica 0..359
    self.t.right(ndir)                     # manda girar à direita ndir
    self.t.fd(dist)                        # manda andar dist
    x=self.t.pos()                         # descobre onde parou (x,y)
```

```
    self.t.ondex=x[0]                    # guarda onde parou (x)
    self.t.ondey=x[1]                    # guarda onde parou (y)


x=[0]*60 # criam-se 60 formigas
for i in range(0,len(x)):
  x[i]=formiga(i)   # cada formiga ganha a cor i
  x[i].anda()       # anda
for j in range(len(x)):   realizam-se 14400 (60*60*4) movimentos
  for i in range(len(x)):   # cada formiga (são 60) realiza 4 movimentos a cada ciclo
    x[i].anda()
  for i in range(len(x)):
    x[i].anda()
  for i in range(len(x)):
    x[i].anda()
  for i in range(len(x)):
    x[i].anda()
```

11.1.1 Herança

Em uma relação de herança, existem no mínimo 2 classes envolvidas: A mais genérica chamada superclasse ou classe pai. A outra é a mais específica, chamada de subclasse, classe filha ou ainda derivada. É comum em uma classe derivada sobrecarregar os métodos da classe pai.

Capítulo 12

Arquivos

A função open() devolve um objeto arquivo, e é frequentemente usada com dois argumentos: open(nome_do_arquivo, modo).

```
>>> f = open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

O primeiro argumento é uma string contendo o nome do arquivo. O segundo argumento é outra string contendo alguns caracteres que descrevem o modo como o arquivo será usado. O parâmetro mode pode ser 'r' quando o arquivo será apenas lido, 'w' para escrever (se o arquivo já existir seu conteúdo prévio será apagado), e 'a' para abrir o arquivo para adição; qualquer escrita será adicionada ao final do arquivo. A opção 'r+' abre o arquivo tanto para leitura como para escrita. O parâmetro mode é opcional, em caso de omissão será assumido 'r'.

No Windows, 'b' adicionado a string de modo indica que o arquivo será aberto em modo binário. Sendo assim, existem os modos compostos : 'rb', 'wb', e 'r+b'. O Windows faz distinção entre arquivos texto e binários: os caracteres terminadores de linha em arquivos texto são alterados ao ler e escrever. Essa mudança automática é útil em arquivos de texto ASCII, mas corrompe arquivos binários como .JPEG ou .EXE. Seja cuidadoso e use sempre o modo binário ao manipular tais arquivos. No Unix, não faz diferença colocar um 'b' no modo, então você pode usar isto sempre que quiser lidar com arquivos binários de forma independente da plataforma.

N.d.T. Para ler arquivos de texto contendo acentuação e outros caracteres não-ASCII, a melhor prática desde o Python 2.6 é usar a função io.open(), do módulo io, em vez da função embutida open(). O motivo é que io.open() permite especificar a codificação logo ao abrir um arquivo em modo texto para leitura ou escrita. Desta forma, a leitura do arquivo texto sempre devolverá objetos unicode, independente da codificação interna do arquivo no disco. E ao escrever em um arquivo aberto via io.open(), basta enviar strings unicode, pois a conversão para o encoding do arquivo será feita automaticamente. Note que ao usar io.open() sempre faz diferença especificar se o arquivo é binário ou texto, em todos os sistemas operacionais: o modo texto é o default, mas se quiser ser explícito coloque a letra t no parâmetro mode (ex. rt, wt etc.); use use a letra b (ex. rb, wb etc.) para especificar modo binário. Somente em modo texto os métodos de gravação e leitura aceitam e devolvem strings unicode. Em modo binário, o método write aceita strings de bytes, e os métodos de leitura devolvem strings de bytes também.

12.1 Métodos de objetos arquivo

Para simplificar, o resto dos exemplos nesta seção assumem que um objeto arquivo chamado f já foi criado.

Para ler o conteúdo de um arquivo, invoque f.read(size), que lê um punhado de dados devolvendo-os como uma string de bytes str. O argumento numérico size é opcional. Quando size é omitido ou negativo, todo o conteúdo do arquivo é lido e devolvido; se o arquivo é duas vezes maior que memória da máquina, o problema é seu. Caso contrário, no máximo size bytes serão lidos e devolvidos. Se o fim do arquivo for atingido, f.read() devolve uma string vazia ().

```
>>> f.read()
'Texto completo do arquivo.\n'
>>> f.read()
''
```

O método f.readline() lê uma única linha do arquivo; o caractere de quebra de linha (`'\n'`) é mantido ao final da string, só não ocorrendo na última linha do arquivo, se ela não termina com uma quebra de linha. Isso elimina a ambiguidade do valor devolvido; se f.readline() devolver uma string vazia, então é certo que arquivo acabou. Linhas em branco são representadas por um `'\n'` – uma string contendo apenas o terminador de linha.

```
>>> f.readline()
'Primeira linha do arquivo.\n'
```

```
>>> f.readline()
'Segunda linha do arquivo.\n'
>>> f.readline()
''
```

O método f.readlines() devolve uma lista contendo todas as linhas do arquivo. Se for fornecido o parâmetro opcional sizehint, será lida a quantidade especificada de bytes e mais o suficiente para completar uma linha. Frequentemente, isso é usado para ler arquivos muito grandes por linhas, sem ter que ler todo o arquivo para a memória de uma só vez. Apenas linhas completas serão devolvidas.

```
>>> f.readlines()
['Primeira linha do arquivo.\n', 'Segunda linha do arquivo.\n']
```

Uma maneira alternativa de ler linhas do arquivo é iterar diretamente pelo objeto arquivo. É eficiente, rápido e resulta em código mais simples:

```
>>> for line in f:
        print line,
```

gerará o seguinte resultado

```
Primeira linha do arquivo.
Segunda linha do arquivo.
```

Essa alternativa é mais simples, mas não oferece tanto controle. Como as duas maneiras gerenciam o buffer do arquivo de modo diferente, elas não devem ser misturadas.

O método f.write(string) escreve o conteúdo da string de bytes para o arquivo, devolvendo None.

```
>>> f.write('Isto é um teste.\n')
```

N.d.T. Neste exemplo, a quantidade de bytes que será escrita no arquivo vai depender do encoding usado no console do Python. Por exemplo, no encoding UTF-8, a string acima tem 18 bytes, incluindo a quebra de linha, porque são necessários dois bytes para representar o caractere 'é'. Mas no encoding CP1252 (comum em Windows no Brasil), a mesma string tem 17 bytes. O método f.write apenas escreve bytes; o que eles representam você decide.

Ao escrever algo que não seja uma string de bytes, é necessário converter antes:

```
>>> valor = ('a resposta', 42)
>>> s = str(valor)
>>> f.write(s)
```

N.d.T. Em particular, se você abriu um arquivo f com a função embutida open(), e deseja escrever uma string Unicode x usando f.write, deverá usar o método unicode.encode() explicitamente para converter x do tipo unicode para uma string de bytes str, deste modo: f.write(x.encode('utf-8')). Por outro lado, se abriu um arquivo f2 com io.open(), pode usar f2.write(x) diretamente, pois a conversão de x – de unicode para o encoding do arquivo – será feita automaticamente.

O método f.tell() devolve um inteiro long que indica a posição atual de leitura ou escrita no arquivo, medida em bytes desde o início do arquivo. Para mudar a posição utilize f.seek(offset, de_onde). A nova posição é computada pela soma do deslocamento offset a um ponto de referência especificado pelo argumento de_onde. Se o valor de de_onde é 0, a referência é o início do arquivo, 1 refere-se à posição atual, e 2 refere-se ao fim do arquivo. Este argumento pode ser omitido; o valor default é 0.

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Vai para o sexto byte do arquivo
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Vai para terceiro byte antes do fim
>>> f.read(1)
'd'
```

Quando acabar de utilizar o arquivo, invoque f.close() para fechá-lo e liberar recursos do sistema (buffers, descritores de arquivo etc.). Qualquer tentativa de acesso ao arquivo depois dele ter sido fechado resultará em falha.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

É uma boa prática usar o comando with ao lidar com objetos arquivo. Isto tem a vantagem de garantir que o arquivo seja fechado quando a execução sair do bloco dentro do with, mesmo que uma exceção tenha sido levantada. É também muito mais sucinto do que escrever os blocos try-finally necessários para garantir que isso aconteça.

```
>>> with open('/tmp/workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

Objetos arquivo têm métodos adicionais, como isatty() e truncate() que são usados com menos frequência; consulte a Referência da Biblioteca Python para mais informações.

12.2 O módulo pickle

Strings podem ser facilmente escritas e lidas de um arquivo. Números exigem um pouco mais de esforço, uma vez que o método read() só devolve strings, obrigando o uso de uma função como int() para produzir o número 123 a partir da string '123' . Entretanto, quando estruturas de dados mais complexas (listas, dicionários, instâncias de classe,etc) estão envolvidas, o processo se torna bem mais complicado.

Para não obrigar os usuários a escrever e depurar constantemente código para salvar estruturas de dados, Python oferece o módulo padrão pickle. Este é um módulo incrível que permite converter praticamente qualquer objeto Python (até mesmo certas formas de código!) para uma string de bytes. Este processo é denominado pickling (N.d.T. literalmente, "colocar em conserva", como picles de pepinos em conserva). E unpickling é o processo reverso: reconstruir o objeto a partir de sua representação como string de bytes. Enquanto estiver representado como uma string, o objeto pode ser facilmente armazenado em um arquivo ou banco de dados, ou transferido pela rede para uma outra máquina.

Se você possui um objeto qualquer x, e um objeto arquivo f que foi aberto para escrita, a maneira mais simples de utilizar este módulo é:

```
pickle.dump(x, f)
```

Para reconstruir o objeto x, sendo que f agora é um arquivo aberto para leitura:

```
x = pickle.load(f)
```

(Existem outras variações desse processo, úteis quando se precisa aplicar sobre muitos objetos ou o destino da representação string não é um arquivo; consulte a documentação do módulo pickle na Referência da Biblioteca Python.)

O módulo pickle é a forma padrão da fazer objetos Python que possam ser compartilhados entre diferentes programas Python, ou pelo mesmo programa em diferentes sessões de execução; o termo técnico para isso é objeto persistente. Justamente porque o módulo pickle é amplamente utilizado, vários autores que escrevem extensões para Python tomam o cuidado de garantir que novos tipos de dados, como matrizes numéricas, sejam compatíveis com esse processo.

Capítulo 13

Pacotes em Python

13.1 Conceito

13.2 Instalação

13.3 uma amostra

Uma pequena amostra dos mais de 100.000 pacotes existentes no ambiente Python

13.4 LISTA DE PACOTES PYTHON

PACOTE	FINALIDADE
TENSORFLOW	Implementa redes neurais em python. Autoria: google
PYEASYGA	Implementa algoritmos de algoritmos genéticos
DEAP	Implementa algoritmos de computação evolutiva, incluindo GA
BOKEH	Apresentação de dados: tabelas, desenhos e inúmeras formas de apresentação de dados
MATPLOTLIB	Apresentação de dados
PYGAME	Toda a infraestrutura para escrever games em python
TKINTER	Interface TK para escrever aplicativos gráficos em python
NUMPY	Computação numérica em python
TURTLE	Simulação da linguagem LOGO
RANDOM	Todas as tratativas para manuseio de números aleatórios
ASTROPY	manuseia arquivos digitais FITS
DJANGO	Framework web
FLASK	Framework web
HUG	Framework web
KERAS	Machine learning, mais fácil que o tensor flow
SCIKIT-LEARN	Machine learning
THEANO	Machine learning
OpenCV	Visão computacional
ORANGE3	Framework para data mining
SCRAPY	Um crawler (spider fuçadeira) em python
PYODBC	ODBC em python (para acessar bancos de dados)
SPHINX	Documentador em python (originalmente desenvolvido para documentar o python, mas pode ser usado em outros ambientes)
REQUESTS	Requisições HTTP
COLLECTIONS	Mantém histórico de processamentos em contendores. Elabora diversos processamentos auxiliares em listas.
HEAPQ	recupera os n maiores ou menores em uma coleção usando um *heap*
RE	Processamento extensivo de expressões regulares, por exemplo pesquisando e substituindo strings.
UNICODEDATA	Normalização e compatibilização de strings UTF-8, UTF-16, ASCII, Latin1, entre muitos outros.
TEXTWRAP	reformatar texto em diversas modalidades.

DECIMAL	Implementa a *general decimal arithmetic specification* da IBM.
CMATH	Aritmética e álgebra complexa.
RANDOM	Funções para números aleatórios.
DATETIME	conversões e aritmética envolvendo diferentes unidades de tempo.
ITERTOOLS	ferramentas adicionais de iteração. Por exemplo, ele gera todas as combinações ou permutações de um contendor.
GZIP e BZ2	Comprime e descomprime conteúdos usando GZIP ou BZ2.
FNMATCH	processa nomes de arquivos e diretórios.
OS	interfaceia com o sistema operacional.
PYSERIAL	Manusear a interface serial do computador.
PICKLE	Serializar objetos.
CSV	Manusear arquivos tipo CSV.
JSON	Manusear dados JSON.
XML	Manusear objetos XML.
BASE64	Converter de e para Base64.
STRUCT	Manusear objetos binários complexos, registros variáveis aninhados, etc.
PANDAS	Análise de dados envolvendo estatísticas, séries temporais e técnicas relacionadas.
FUNCTOOLS	Ferramentas adicionais para definição de funções. (como wrapper=encapsular uma função qualquer com processamento adicional, por exemplo um timmer).
QUEUE	manipulação de filas e listas.
WEAKREF	Indireções com referências fracas (compatibilidade com gabrage collection).
AST	Abstract Sintax Tree, ou criar AST a partir de código fonte Python, permitindo análise e manipulação.
DIS	disassembly de código python.
SYS	Manipulação de PATHs entre outras coisas.
URLIB	Serviços de web.
SOCKET	Serviços de socket.
TIME	Manipulação de tempo.
IPADDRESS	manipulação de endereços IP.
CGI	Interfaces baseada em REST.
XMLRPC	Procedimentos remotos.
HMAC	Handshake para autenticação.
SSL	camadas SSL para segurança.
THREADING	Manipulação de threadings.
CONFIGPARSER	Manipulação de arquivos de configuração.
LOGGING	Manipular funções de logging.
RESOURCE	Impõe limites de memória ou de CPU a programas (unix).
WEBBROWSER	Iniciar um browser web.
QRCODE	Gerador e manipulador de QRCODES.
PYSIM	Programação simbólica (à moda do Maple)
MATPLOTLIB	Gráficos em python

Capítulo 14

Pacote: Numpy

Para usar o numpy:

Importação Se quiser chamar depois os módulos como `np.modulo` importe assim

```
>>> import numpy as np
```

Se quiser chamar direto sem usar a notação de ponto, importe assim

```
>>> from numpy import *
```

Para criar matriz O elemento `array` é central em numpy:

```
a=np.array([[1,2,3],[3,4,5],[6,6,7]])
ou
a=np.array([(1,2,3),(3,4,5),(6,6,7)])

>>> a
array([[1, 2, 3],
       [3, 4, 5],
       [6, 6, 7]])
```

ambos jeitos devolvem. A explicação para isto é que na criação, tanto faz fornecer uma lista ou uma tupla.

Para indexar

```
a[1][2] é igual a 5 OU a[1,2] que igualmente é 5
```

in É usado para testar se um elemento está num array ou não:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> 2 in a
True
>>> 0 in a
False
```

Criação de PA A função arange() cria uma PA, cujos termos são a=np.arange(vl_inicial, até_vl_final, incremento), como em

```
a=np.arange(1,5,0.5)
>>> a
array([ 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

Atributos de ndarray Um ndarray tem os seguintes atributos:

.ndim = número de dimensões

.shape = uma tupla com as dimensões

```
>>> a
array([1, 2, 3, 4])
>>> a.shape
(4,)
```

Cópia de array Para criar um array como cópia de outro, fazer

```
s3 = s2.copy()
```

Reorganizar um array Para reorganizar um array:

```
>>> a1 = np.arange(0.0, 12.0, 1.0)
>>> a1
[ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.]
>>> a2 = a1.reshape( (2,6) )
>>> print a2
[[ 0. 1. 2. 3. 4. 5.]
[ 6. 7. 8. 9. 10. 11.]]
```

Reorganizar in place Para reorganizar in-place

```
>>> a1
[ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.]
>>> a1.resize([2,6])
>>> a1
[[ 0. 1. 2. 3. 4. 5.]
[ 6. 7. 8. 9. 10. 11.]]
```

Média Para retornar a média, variância, soma e produto:

```
A.mean() -> média dos valores em A
A.var() -> variancia dos valores de A
A.sum() -> soma
A.prod() -> produto
A.max() -> maior valor
A.min() -> menor valor
A.argmax() -> índice do maior valor
A.argmin() -> índice do menor valor
unique(a) -> retorna os valores únicos
             (sem repetição) de a
```

Funções Universais Para funções universais (ufuncs)

```
>>> a=[1,2,3,4]
>>> b=[0.1, 0.7, 8, 9]
>>> a
[1, 2, 3, 4]
>>> b
[0.1, 0.7, 8, 9]
>>> a+b
[1, 2, 3, 4, 0.1, 0.7, 8, 9]
>>> a=np.array([1,2,3,4])
>>> b=np.array([0.1, 0.7, 8, 9])
>>> a+b
array([  1.1,   2.7,  11. ,  13. ])
```

Broadcasting Para broadcasting (aplicar um escalar a um array)

```
>>> a
array([1, 2, 3, 4])
>>>
>>> a+3
array([4, 5, 6, 7])
>>> b*5
array([  0.5,   3.5,  40. ,  45. ])
```

multiplicação matricial Multiplicação matricial:

```
>>> a1 = np.array([[1, 2, -4], [3, -1, 5]])
>>> a2 = np.array([[6, -3], [1, -2], [2, 4]])
>>> np.dot(a1,a2)
array([[  0, -23],
       [ 27,  13]])
```

Produto vetorial O produto vetorial é

```
>>> x = (1, 2, 0)
>>> y = (4, 5, 6)
>>> print np.cross(x, y)
[12 -6 -3]   para resolver isto ache o
             determinante da matriz
i j k
1 2 0
4 5 6, com resultados em i, j e k
```

determinante Para obter o determinante de uma matriz

```
>>> m = np.array(((2,3), (-1, -2)))
>>> m
[[ 2 3]
[-1 -2]]
>>> np.linalg.det(m)
-1.0
```

Matriz inversa E a matriz inversa

```
>>> good = np.array(((2.1, 3.2), (4.3, 5.4)))
>>> np.linalg.inv(good)
[[-2.23140496 1.32231405]
[ 1.7768595 -0.8677686 ]]
```

Sistema de equações lineares A solução de um sistema de equações lineares por exemplo

```
2x + y = 19
x - 2y = 2
cuja solução é (x=8 and y=3):
>>> coeffs = np.array([[2,1], [1,-2]])
>>> coeffs
[[ 2 1]
[ 1 -2]]
>>> consts = np.array((19, 2))
>>> consts
[19 2]
>>> np.linalg.solve(coeffs, consts)
[ 8. 3.]
```

Transposta

```
>>> a = np.array(range(6), float).reshape((2, 3))
>>> a
array([[ 0., 1., 2.],
[ 3., 4., 5.]])
>>> a.transpose()
array([[ 0., 3.],
[ 1., 4.],
[ 2., 5.]])
```


Concatenação

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])

>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7,8]], float)
>>> np.concatenate((a,b))
array([[ 1., 2.],
[ 3., 4.],
[ 5., 6.],
[ 7., 8.]])
>>> np.concatenate((a,b), axis=0)
array([[ 1., 2.],
[ 3., 4.],
[ 5., 6.],
[ 7., 8.]])
>>> np.concatenate((a,b), axis=1)
array([[ 1., 2., 5., 6.],
[ 3., 4., 7., 8.]])
```

Funções matemáticas abs, sign, sqrt, log, log10, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, and arctanh.

floor, ceil, and rint give the lower, upper, or nearest (rounded) integer:

Lista de nomes definidos Para retornar uma lista de nomes definidos

```
>>> dir()
```

subpacotes Lista de subpacotes que ajudam na programação:

subpacote	Objetiv
core	basic objects
lib	additional utilities
linalg	basic linear algebra
fft	discrete Fourier transforms
random	random number generators
distutils	enhanced build, distrib improvements
testing	unit-testing utility functions
f2py	automatic wrapping of Fortran code

Geração de aleatórios Compreendidos entre 0 e 1.

```
>>> np.random.rand(3,2)
array([[ 0.14022471, 0.96360618], #random
       [ 0.37601032, 0.25528411], #random
       [ 0.49313049, 0.94909878]]) #random
```

Inteiros aleatórios Veja o exemplo

```
>>> randint(low[, high, size]) # Return random integers
        from low (inclusive) to high (exclusive)
```

Para mostrar

```
>>> import matplotlib.pyplot as plt
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums=d2+d2
>>> count,bins,ignored = plt.hist(dsums,11,normed=True)
>>> plt.show()
```

Distribuição normal Esta função retorna simulações de uma distribuição normal. Para obter $N(\mu, \sigma^2)$, basta fazer $sigma * np.random.randn(...) + mu$. Onde μ = média da distribuição e σ^2 é a variância. Veja-se o exemplo

```
>>> 2.5 * np.random.randn(2, 4) + 3
array([[-4.4940, 4.0095, -1.8181, 7.2971], #random
       [ 0.3992, 4.6845,  4.9939, 4.8405]]) #random
```

Cria um array $2, 4$ contendo simulações de $3, 6.25$. Lembre que $2.5^2 = 6.25$.

Distribuição binomial Cria simulações desta distribuição. Usa 2 parâmetros: `n`, que significa o número de tentativas e `p` que vem a ser a probabilidade de sucesso da distribuição ($0 \leq p \leq 1$). A função retorna para cada simulação o número de sucessos. Lembrando que a probabilidade é

$$P(N) = \left(\begin{array}{c} n \\ N \end{array} \right) p^N (1-p)^{n-N}$$

) Veja-se o exemplo:

```
>>> n=10
>>> p=0.5
>>> s=np.random.binomial(n,p,1000)
```

O exemplo acima faz 1000 simulações, cada uma jogando 10 moedas e informando a quantidade de caras em cada simulação.

Ordenação

```
>>> a = np.array([[1,4],[3,1]])
>>> np.sort(a) # sort along the last axis
array([[1, 4],
       [1, 3]])
>>> np.sort(a,axis=None) #sort flattened array
array([1, 1, 3, 4])
>>> np.sort(a,axis=0) # sort along first axis
array([[1, 1],
       [3, 4]])
```

Desvio padrão em arrays

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949
>>> np.std(a, axis=0)
array([ 1., 1.])
>>> np.std(a, axis=1)
array([ 0.5, 0.5])
```

14.1 Varejão

14.1.1 Para gravar arquivo de números

```
def testagra(a):
    f=open("f:/p/up2017/ofid/primos.lis","w")
    f.write(str(a)[1:-1])
    f.close()
```

14.1.2 Para ler o arquivo acima

e converter os números em uma lista

```
def leiapri():
    f=open("f:/p/up2017/ofid/primos.lis","r")
    a=f.read()
    pri=list(map(int,a.split(", ")))
```


Capítulo 15

Pacote: Sympy

O Symbolic Python é um pacote Python que implementa um sistema de computação simbólica (CAS=Computer Algebra System) no ambiente Python. Para entender o que é um sistema deste tipo, acompanhe:

computação numérica sqrt(8) é igual a 2.8284271247461903

computação simbólica sqrt(8) é igual a 2*sqrt(2)

Sendo que aqui tem-se uma constatação: ir do numérico para o simbólico não é possível, mas ir do simbólico para o numérico é fácil (neste caso basta fazer `sqrt(8).evalf()` cuja resposta será a mesma de cima – até porque é calculada do mesmo jeito: 2.8284271247461903).

Fica evidente que a computação simbólica abrange a computação numérica mas é muito (muito !) mais do que ela.

As vantagens de usar o sympy versus o Maple (por exemplo) são pelo menos duas: sympy é de graça (Maple custa 1.000 dólares) e por ser em python admite que você use tudo o que já sabe de python e também permite usar os mais de 135.000 pacotes também de graça.

Instalação e uso O sympy precisa ser instalado. Na versão para windows, você deve localizar o diretório scripts e nele fazer `pip install sympy` em um computador que tenha acesso à Internet. Se quiser usar winpython, nele o sympy já está pré-instalado. Num ou noutro, precisa importar o pacote. Faça `from sympy import *` ou algum comando equivalente.

symbols Como você está num ambiente 100% python, precisa dizer que algumas variáveis são simbólicas. Isto é feito pelo método `symbols` como em

```
from sympy import symbols
>>> x,y = symbols('x y')
>>> expr = x+2*y
>>> expr
x + 2*y
>>> expr + 1
x + 2*y + 1
>>> expr - x
2*y
>>> x*expr
x*(x + 2*y)
>>> expand(x*expr)
x**2 + 2*x*y
>>> factor(x**2 + 2*x*y)
x*(x + 2*y)
```

O sympy pode fazer todo o tipo de computação simbólica, como simplificar, calcular derivadas, integrais e limites, resolver equações, trabalhar com matrizes, e muito mais, tudo de maneira simbólica. Ele inclui módulos para plotagem, impressão, geração de código, física, estatística, combinatória, teoria dos números, geometria, lógica, etc. Vejamos um exemplo para calcular: a derivada de

$$sin(x) \times e^x$$

,

$$\int ((e^x \times sin(x) + e^x \times cos(x))dx$$

e também

$$\int_{-\infty}^{\infty} sin(x^2)dx$$

```
>>> from sympy import *
>>> x, t, z, nu = symbols('x t z nu')
>>> init_printing(use_unicode=True)
>>> diff(sin(x)*exp(x), x)
 x           x
e .sin(x) + e .cos(x)
>>> integrate(exp(x)*sin(x) + exp(x)*cos(x), x)
 x
e .sin(x)
>>> integrate(sin(x**2), (x,-oo, oo))
```

Substituição

```
>>> expr = cos(x)+1
>>> expr.subs(x,y)
cos(y) + 1
>>> expr.subs(x,0)
2
>>> expr = x**y
expr = expr.subs(y, x**y)
>>> expr
x**(x**y)
>>> expr = x**3 + 4*x*y - z
>>> expr.subs([(x,2),(y,4),(z,0)])
40
```

Sympify Não confundir com `simplify`. Transforma um string em uma expressão SymPy.

```
a = "x**2 + 3*x - 1/2"
ex = sympify(a)
ex.subs(x,2)
19/2
```

evalf Permite achar o valor em ponto flutuante de uma expressão

```
>>> ex = sqrt(8)
>>> ex.evalf(20)  # 20=tamanho
2.8284271247461900976
>>> ex2 = cos(2*x)
>>> ex2.evalf(subs={x:2.4})
0.087498983439446 4
```

Simplify simplificação de expressões.

```
>>> simplify(sin(x)**2 + cos(x)**2)
1
>>> simplify((x**3+x**2-x-1)/(x**2+2*x+1))
x - 1
```

Expand Expande expressões.

```
>>> expand((x+1)**2)
 2
x + 2.x + 1
>>> expand((x+2)*(x-3))
 2
x . x - 6
```

Factor Fatora expressões.

```
>>> factor(x**2*z + 4*x*y*z + 4*y**2*z)
           2
z.(x + 2.y)
```

Outros comandos collect = coleciona potências comuns
cancel = transforma polinômio na forma p/q
apart = executa uma decomposição em farções parciais
trigsimp = simplificação trigonométrica
expand_trig = expansão trigonométrica
factorial, binomial, gamma, power, powers, ...

Derivadas podem ser chamadas com `diff(expressao)` e com `Derivative(expressao)`.

```
>>> diff(cos(x),x)
-sin(x)
```

Integrais o comando que integra é `integrate`. O comando `Integral` serve para representar a integral e pode ser calculado depois com `x.doit()`. Atente que integrate é com minúscula e Integral é com maiúscula. Acompanhe

```
>>> x = Integral(log(x)**2,x)
>>> x
  /
 |
 |    2
 | log (x) dx
 |
/
>>> x.doit()
     2
x.log (x) - 2.x.log(x) + 2.x
>>>
sin(x)
```

Limites

```
>>> limit(sin(x)/x, x, 0)
1
```

Solve Em Sympy a igualdade em uma equação é dada por Eq como em $x^2 = 1 \Rightarrow$ `Eq(x**2,1)` e para resolver esta equação escreve-se

```
>>> solveset(Eq(x**2,1), x)
{-1, 1}
```

Note que isto pode ser escrito como `solveset(Eq(x**2-1,0), x)` e neste caso pode ser também `solveset(x**2-1, x)` (já que é igual a zero). Para resolver um sistema de equações o comando é `linsolve`, veja

```
>>>linsolve([x+y+z-1,x+y+2*z-3],(x,y,z))
{(-y-1, y, 2)}
>>>linsolve(([1,1,1,1],[1,1,2,3])),(x,y,z))
{(-y-1, y, 2)}
```

Para sistemas não lineares, o comando é `nonlinsolve`.

```
>>>nonlinsolve([x*y-1,x-2],x,y)
{(2, 1/2)}
```

Para equaçoes diferenciais, use `dsolve`.

Matrizes Para fazer matrizes em Sympy use o objeto `Matrix`, como em `Matrix([[1, -1], [3, 4], [0, 2]])`. Matrizes não são imutáveis, podendo ser alteradas.

Para a matriz `M`, sua forma é $M.shape$. Para achar a inversa fazer $M**-1$. Para achar a transposta, faça $M.T$. Para o determinante, use $M.det$.

```
>>> M=Matrix([[1,2,3],[10,20,51],[8,-1,9]])
>>> M
+1   2   3 +
|          |
|10  20  51|
|          |
+8   -1  9 +
>>> M**-1
```



```
+ 11                  +
| --    -1/17   2/17  |
| 17                  |
|                     |
|106                  |
|---    -5/119  -1/17 |
|119                  |
|                     |
|-10                  |
|--    1/21     0     |
+ 21                  +
>>> M.T
+1   10   8  +
|            |
|2   20   -1 |
|            |
+3   51   9  +
```

Segue a documentação tratando de teoria dos números (basicamente as tratativas referentes a números primos, depois criptografia, depois matemática concreta, depois geometria, depois integração, lógica, séries, ... a documentação do pacote tem (em maio de 2018) 1.878 páginas.

15.0.1 Exercício

1. Resolva (usando `solveset`) a equação:

$$3x^3 + 4x^2 + 32x + 256 = 0$$

no domínio dos números reais.

2. Resolva a integral definida a seguir,

$$\int_6^8 (3 \times sin(x))$$

de maneira numérica, usando o comando `Integral` de SymPy nos intervalos dados e depois calcule o valor numérico da integral usando o método `as_sum` com 50 intervalos e tomando como parâmetro o ponto médio de cada intervalo: (Forneça a resposta com 3 casas decimais)

```
>>> init_printing(use_unicode=True)
>>> e = Integral(...funçao...,(x,a,b))
>>> e
>>> e.as_sum(50,'midpoint').evalf()
```

1	2

15.0.2 Resposta

-4 3.317

Capítulo 16

Pacote: Astropy

16.1 Padrão FITS

FITS é uma sígla que significa *Flexible Image Transport System* e é um padrão aberto que define um formato de arquivo digital usado para armazenar, transmitir e processar imagens (originalmente) científicas. O padrão nasceu na área de astronomia justamente para permitir o intercâmbio de imagens digitais obtidas em telescópios de todas as naturezas possíveis. O arquivo é suficientemente genérico para acomodar muitos dados científicos além da imagem em si. Por exemplo, é comum haver dados de fotometria, informações de calibração, filtros usados além de muitas informações sobre a imagem em si, o que chamamos metadados. O padrão FITS surgiu em 1981 e evoluiu gradualmente desde então, a versão mais recente é a 3 de 2008. O mote do FITS é *once FITS, always FITS* e ele tem a pretensão de ser legível e manuseável mesmo depois que o software e o hardware usados para criar a imagem original não existam mais. Repare que este requisito inviabiliza o uso de formatos proprietários (GIF) bem como aqueles dependentes de alguma tecnologia específica (JPG, BMP, RAW, PDF, etc). O formato ganhou visibilidade quando foi escolhido pela Biblioteca do Vaticano para disponibilizar imagens de documentos daquele lugar, talvez a mais importante biblioteca de documentos antigos do mundo, repleta de manuscritos dos séculos V, VIII, XI e assim por diante. Suas principais características:

Não proprietário O formato FITS é mantido pela comunidade científica internacional, em particular a comunidade de astrônomos e astrofísicos e como tal ele não é formato proprietário. Atualmente a especificação é mantida pelo Grupo de Trabalho FITS da IAU, União Internacional de Astronomia.

Bem documentado O formato FITS está completamente documentado na Referência FITS, livremente disponível. A versão 3 foi publicada no *Astronomy and Astrophysics* volume 524 de dezembro de 2010.

Largamente adotado O formato é comumente usado pelos astrônomos. Há muitas aplicações de software para criar e ver os arquivos, além de permitir a importação e exportação em outros formatos também usuais.

Transparência Uma grande vantagem é a simplicidade do formato. Nada é comprimido (o que introduziria grande risco em caso da corrupção de um único bit...) e os dados são guardados ora em ASCII puro, ora em formatos de codificação padronizados.

Auto contido Todas as informações necessárias para representar corretamente a imagem (na tela ou na impressora) estão incluídas dentro do arquivo. Links externos ou fontes de recursos externos não são permitidos.

Independência de dispositivo Não há nenhuma dependência a algum hardware ou software específicos em nenhum ponto do arquivo. Esta característica é muito importante na compatibilidade para a frente, pois não temos idéia do que vem por aí.

16.1.1 Formato

O arquivo é formado por conjuntos de header + dados, tantos quantos houver interesse. O primeiro conjunto é chamado primário e é obrigatório. Quando só tem o primário o arquivo é conhecido como Arquivo FITS básico e quando tem uma ou mais extensões seguindo o conjunto primário é chamado de Arquivo FITS multi-extensão. Cada conjunto (chamado na documentação de *FITS structure*) é formado por um número inteiro de blocos FITS, cada um com 2880 bytes e escrito em ASCII puro ($2880 = 80 \times 36$). Todos estes blocos ficam juntos, antes dos dados. Por ASCII puro entende-se caracteres de 32 até 126.

O bloco de header contém uma série de palavras chaves seguida de seu valor. Assim, cada header tem espaço para 36 palavras chave. O último bloco de header deve conter a palavra chave `END` que marca o fim lógico do header. O que sobrar de espaço depois, deve ser prenchido com espaços.

O conjunto de dados que vem depois, (se presente) deve consistir de um arranjo (array) de dados de 1 a 999 dimensões (isto é definido pela palavra chave `NAXIS` O array inteiro é representado por um stream contínuo de bits que começa no primeiro bit do primeiro bloco de dados. Cada dado consiste de um número fixo de bits como determinado na palavra

chave BITPIX. O índice do eixo 1 é o que varia mais rapidamente, depois o do eixo 2, e assim por diante. O último eixo (NAXIS) é o que varia mais lentamente. Não há espaço ou qualquer caractere especial entre o último valor em uma linha ou plano e o primeiro valor da próxima linha ou plano. A contagem começa em 1. Veja na figura como é isso:

```
A(1, 1, ..., 1),
A(2, 1, ..., 1),
...,
A(NAXIS1, 1, ..., 1),
A(1, 2, ..., 1),
A(2, 2, ..., 1),
...,
A(NAXIS1, 2, ..., 1),
...,
A(1, NAXIS2, ..., NAXISm),
...,
A(NAXIS1, NAXIS2, ..., NAXISm)
```

Se o bloco de dados não preenche o último bloco ele deve ser preenchido com bits zero.

O bloco de header é formado por registros de 80 caracteres (36 por bloco) e eles consistem de uma palavra chave, um indicador de valor (se necessário), um valor opcional e um comentário opcional. As palavras podem aparecer em qualquer ordem, com poucas exceções. A palavra chave deve ter até 8 posições, alinhada à esquerda e com brancos à direita, sempre em letras maiúsculas. As posições 9 e 10 podem ter "=␣". A seguir o valor da palavra chave e opcionalmente um comentário. Entre o valor e o comentário deve-se escrever "␣/␣". Em todo este conteúdo (inclusive comentários) deve-se usar apenas os caracteres ASCII de 32 até 126 – hexadecimal 20 até 7E. Variáveis de valor lógico devem ser representadas por T ou F.

Eis as principais palavras chave:

SIMPLE Deve conter um valor verdade (T) se este arquivo obedece ao padrão FITS.

BITPIX Contém um inteiro e indica o número de bits que representa cada valor no array de dados. Os valores possíveis são: 8, 16, 32, 64 – para inteiros – e -32 ou -64 para ponto flutuante.

NAXIS Inteiro não negativo não maior que 999 indicando quantos eixos tem o array de dados.

NAXIS**n** Esta palavra deve estar presente para todos os valores de n de 1 até NAXIS. Indica o tamanho da dimensão do array de dados.

END Indica final do cabeçalho. Esta palavra chave não tem valor associado

Veja-se um exemplo disto:

```
SIMPLE   = T / file does conform to FITS standard
BITPIX   = 16 / number of bits per data pixel
NAXIS    = 2 / number of data axes
NAXIS1   = 250 / length of data axis 1
NAXIS2   = 300 / length of data axis 2
OBJECT   = 'Cygnus X-1'
DATE     = '2008-10-27'
END
```

O número total de bits do array de dados primário, sem contar o preenchimento eventualmente necessário para completar o bloco de 2880 caracteres, é $N_{bits} = |BITPIX| \times (NAXIS1 \times NAXIS2 \times ... \times NAXISm)$

16.1.2 A cor no arquivo FITS

Paradoxalmente, a cor não tem um significado fixo no arquivo FITS. Ela pode (deve) ser livremente interpretada numa negociação entre gravador e leitor. O formato lava as mãos sobre isso. Para imagens coloridas do uso no nosso dia a dia, pode-se fazer a seguinte combinação: true color (8 bits=0..255) por canal R, G, B e mais um quarto canal de transparência. Total 32 bits por pixel. Neste caso, pode-se usar o inteiro de 32 bits no registro do arquivo. Se uma maior precisão de cor é exigida (como por exemplo na captura de pergaminhos e iluminuras da Biblioteca Vaticana), pode-se fazer: 16 bits por canal R, G ou B chegando portanto a 65.536 níveis em cada cor, e mais um canal de transparência com o mesmo nível de valores. Total 64 bits por pixel, usando-se neste caso os inteiros de 64 bits. Ou pode-se definir qualquer outra combinação, eventualmente usando-se outros esquemas de cor (como CMYB, por exemplo). O que resta importante no formato, é que cada valor sempre tem uma configuração que é múltipla de 8 bits, ou seja não há aqui os problemas de truncamento, padding ou stuffing bits de outros formatos.

16.1.3 Como ver arquivos FITS

Existe uma grande variedade de visualizadores de arquivos FITS, e como tudo no padrão, são livres para download e instalação. Um programa que é bem completo e funciona inclusive como editor para este tipo de arquivo é o FV, que tem versões para plataformas windows, unix e Mac. Seu site é https://heasarc.gsfc.nasa.gov/ftools/fv/. Vale uma visita. Outro é DS9, que pode ser baixado em http://ds9.si.edu/site/Home.html.

Especial referência deve ser feita em relação à linguagem Python. Usando a vocação desse ambiente, (106.000 pacotes em mar/17), há um pacote completo chamado ASTROPY que dá toda a funcionalidade para operar magnificamente com arquivos FITS. Mais detalhes em www.astropy.org, onde há um tutorial muito bom.

Para manusear este tipo de arquivo deve-se usar pelo menos 3 ferramentas:

- Um editor ASCII convencional (NOTEPAD++ por exemplo) para estudar o bloco de header
- Um editor de formato binário por exemplo wxHexEditor ou então o site http://www.onlinehexeditor.com/. Este último funciona online e não exige instalação de software. Aqui examina-se os blocos de dados do arquivo FITS
- Um visualizador de arquivos FITS como o FV ou DS9 que permitem examinar completamente o arquivo

Capítulo 17

Pacote: Pygames e Turtle

Hello world com Pygames

Seja o seguinte programa de hello world

```
background_image_filename = 'c:\python\pedrofepo.jpg'
mouse_image_filename = 'c:\python\selo_coxa.jpg'
import pygame
from pygame.locals import *
from sys import exit
pygame.init()
screen = pygame.display.set_mode((1200, 800),0,32)
pygame.display.set_caption("hello")
background = pygame.image.load(background_image_filename).convert()
mouse_cursor = pygame.image.load(mouse_image_filename).convert_alpha()
while True:
  for event in pygame.event.get():
    if event.type == QUIT:
      pygame.quit()
      exit()
    screen.blit(background, (0,0))
    x, y = pygame.mouse.get_pos()
    x-= mouse_cursor.get_width() /2
    y-= mouse_cursor.get_height() /2
    screen.blit(mouse_cursor, (x,y))
    pygame.display.update()
```

Este programa carrega duas imagens (a segunda com transparência), cria um ciclo eterno e nele deixa passar todos os eventos (é o que faz `event in pygame.event.get():`) Se o evento for `QUIT`, isto significa que o usuário teclou no 'X' que fecha a janela e portanto o programa deve ser encerrado. Senão, o programa desenha a tela principal começando em 0,0 (que é o canto esquerdo superior). Daí, ele pega a posição do cursor nas variáveis x,y (de uma vez só, olha aí o modernismo de python). Calcula um novo x,y (não entendi o porque do -, mas enfim), e desenha o cursor nessa nova posição. Finalmente, o conjunto todo é atualizado através do comando `pygame.display.update()`.

tratando eventos

O programa a seguir lista os eventos à medida em que eles ocorrem

```
import pygame
from pygame.locals import *
from sys import exit
pygame.init()
SCREEN_SIZE = (800, 600)
screen = pygame.display.set_mode(SCREEN_SIZE, 0, 32)
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            exit()
        print(event)
    pygame.display.update()
```

Veja-se a seguir um programa de manuseio de listas (similar ao da pág 118-119 de [Men10].

```
V=[1,4,5,6,77,"aa",8,45,46,28,33,34,39]
P=[]
I=[]
for e in V:
  x=str(e)
  print(x)
  print(x.isnumeric())
  if x.isnumeric():
    if e%2 == 0:
      P.append(e)
    else:
      I.append(e)
  else:
    print("deu xabu")
print("Pares",P)
print("Impares",I)

>>> ... executando ...
Pares [4, 6, 8, 46, 28, 34]
Impares [1, 5, 77, 45, 33, 39]
```

Criando um dicionário:

```
def criahash():
  import random
  x = range(100)
  som = 0
  di = {}
  for i in x:
    aa = random.randint(1,1000)
    print("%d - %d\n"%(i+1,aa))
    som = som + aa
    di[aa]=i+1
  som= som / 100
  print("media final = %8.2f" % som)
  print(di)
```

Implementando os números de fibonacci usando um dicionario para guardar o já calculado

```
def fibona(n):
  # supoe um difibo = {0:1, 1:1}
  if n in difibo:
    return difibo[n]
  else:
    novo = fibona(n-1) + fibona(n-2)
    difibo[n] = novo
    return novo
```

Brincando com o Logo

```
import turtle,random
# x=10
# y=0
t=turtle.Pen()
turtle.bgcolor('black')
cor=['red','blue','green','yellow','pink','aquamarine','tomato','ivory','magenta']
for x in range(10,400,5):
  t.width(random.randint(1,3))
  t.forward(x)
#  t.circle(x)
  t.left(90)
#  x=x+random.randint(1,5)
#  x=x+5
```

```
#  y=y+1
#  t.pencolor(cor[y%9])
  t.pencolor(cor[random.randint(0,7)])
```

Como importar um módulo escrito alhures (por exemplo, o módulo `cpf.py` que está em `f:\p\n\024`. Abra uma sessão python *as usual*. Daí emita os seguintes comandos

```
import sys
```

Daí peça para ver se deu certo, pedindo para ver a variável `sys.path`. Ele deve responder com a lista de diretórios de path. Emita o comando

```
sys.path.append('f:\\p\\n\\024')
```

Se você pedir para olhar a variável acima, verá que o novo diretório foi incluído sem problemas. Agora, emita o comando

```
from cpf import cpf
```

E agora pode usar a função cpf definida dentro daquele módulo fazendo

```
aa=cpf([1,1,1,2,2,2,3,3,3])
aa
```

Para criar uma lista com n zeros, fazer

```
lis = [0] * 33 # cria uma lista com 33 zeros
```


Capítulo 18

Pacote: Matplotlib

Este é um dos pacotes disponíveis no mundo Python para elaborar melhor as saídas na forma gráfica. Existem outros, vale a pena uma pesquisa no PyPi (Python Package Index) .

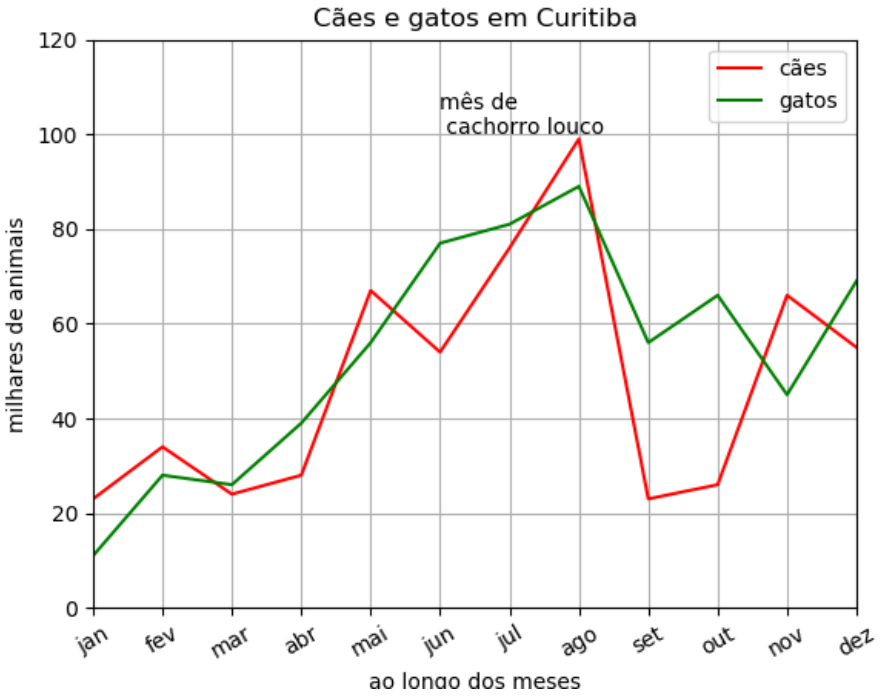
Pacote maneiro para fazer desenhos e manipulações de imagens em Python. O pacote a ser instalado é o matplotlib (pip install matplotlib) e ele exige um monte de dependências (o python, numpy, além de inúmeras outras).

18.1 Modos de uso

Há 2 modos de uso do pacote: usar a interface procedural `pyplot` ou usar a API nativa da matplotlib, esta mais orientada a objetos. A primeira é constituída por métodos (como xlabel, legend, etc) e elas podem ser usadas de duas maneiras: se chamadas sem argumentos, devolvem o valor atual do parâmetro e se se chamadas com parâmetros elas atualizam este valor. Veja a seguir uma chamada usando este método:

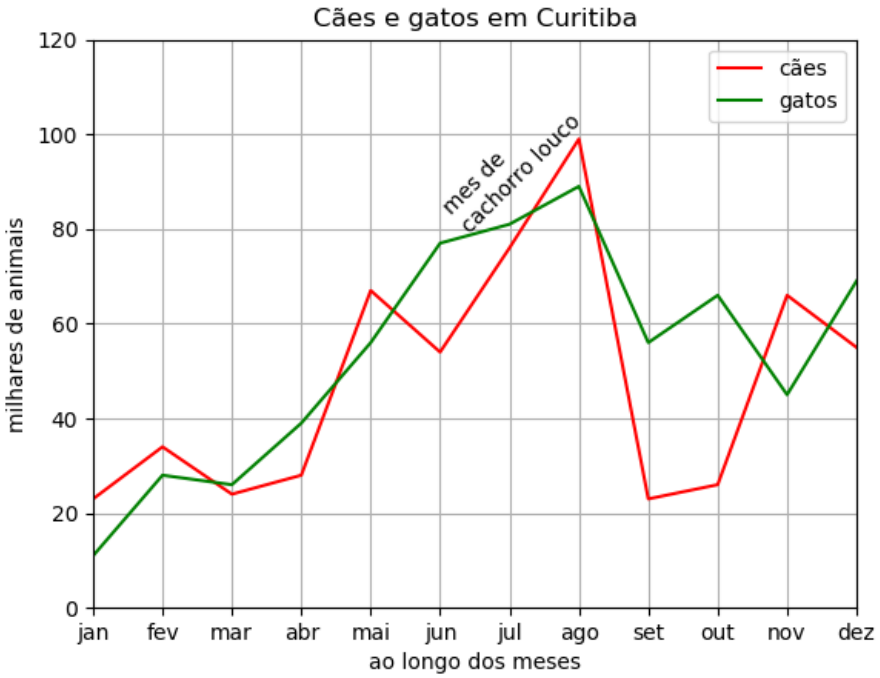
18.2 Para criar um gráfico simples, via métodos

```
# caogato
import matplotlib.pyplot as plt
import numpy as np
def caogato():
 ca=[23,34,24,28,67,54,76,99,23,26,66,55]
 ga=[11,28,26,39,56,77,81,89,56,66,45,69]
 plt.plot(ca,'r-',label='cães')
 plt.plot(ga,'g-',label='gatos')
 plt.legend()
 plt.title('Cães e gatos em Curitiba')
 plt.xlabel('ao longo dos meses')
 plt.ylabel('milhares de animais')
 plt.axis([0,11,0,120])
 plt.xticks(np.arange(12),('jan','fev','mar',
 'abr','mai','jun','jul','ago','set','out',
 'nov','dez'),rotation=30)
 plt.text(5,100,'mês de\n cachorro louco')
 plt.grid()
 plt.savefig('c:/p/python/caogato.png')
 plt.show()
caogato()
```



18.3 O mesmo gráfico, via API

```
#caogato2
import matplotlib.pyplot as plt
import numpy as np
def caogato2():
  ca=[23,34,24,28,67,54,76,99,23,26,66,55]
  ga=[11,28,26,39,56,77,81,89,56,66,45,69]
  fig,axe=plt.subplots()
  axe.plot(ca,'r-',label='cães')
  axe.plot(ga,'g-',label='gatos')
  axe.legend()
  axe.set_title('Cães e gatos em Curitiba')
  axe.set_xlabel('ao longo dos meses')
  axe.set_ylabel('milhares de animais')
  axe.axis([0,11,0,120])
  axe.set_xticks(range(12))
  axe.set_xticklabels(['jan','fev','mar',
  'abr','mai','jun','jul','ago','set','out',
  'nov','dez'])
  axe.grid()
  axe.text(5,100,'mes de \ncachorro louco',rotation=45)
  plt.savefig('c:/p/python/caogato2.png')
  plt.show()
caogato2()
```



t

No caso de usar a API, cada ação corresponde a dois métodos na API. Por exemplo, no caso de `xlabel`, há o método `axe.set_xlabel` para colocar coisas lá e o método `axe.get_xlabel` para recuperar o que tem lá.

18.4 O método plot()

Serve para plotar `y` versus `x` com linhas e/ou marcadores. Vejamos alguns exemplos:

```
plot(y)
```

Marca os valores de `y` usando `x` os índices do array `0..n-1`. Veja um exemplo

```
def exeplo1():
  y=[12,3,50,33]
  plt.plot(y)
  plt.show()
```

Agora, x pode ser explicitamente estabelecido, através de pares `x,y`. Veja o exemplo:

```
def exeplo2():
  x=[8,17,33,66]
  y=[12,3,50,33]
  plt.plot(x,y)
  plt.show()
```

Um segundo conjunto de dados (`x2,y2`) pode ser desenhado junto

```
def exeplo3():
  x=[8,17,33,66]
  y=[12,3,50,33]
  x2=[0,3,7,12]
  y2=[33,88,55,77]
  plt.plot(x,y,x2,y2)
  plt.show()
```

Mas isto pode ser feito em duas chamadas a `plot`, com o mesmo resultado, como em

```
def exeplo4():
  x=[8,17,33,66]
  y=[12,3,50,33]
  x2=[0,3,7,12]
  y2=[33,88,55,77]
  plt.plot(x,y)
  plt.plot(x2,y2)
  plt.show()
```

A formatação das coisas pode ser feita com propriedades de `Line2D` ou com palavras chave. Assim, ambos os valores abaixo terão o mesmo resultado

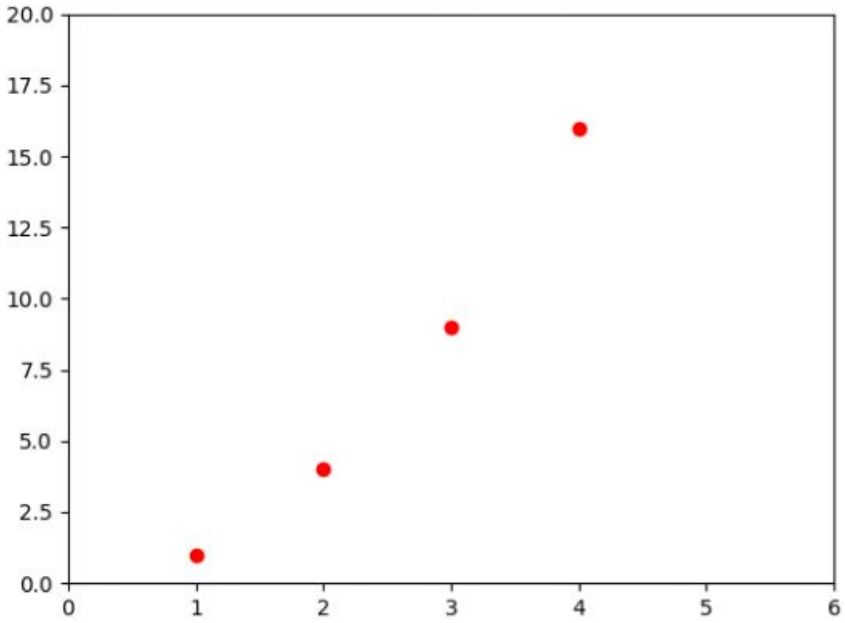
```
>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o',
    linestyle='dashed',
    linewidth=2, markersize=12)
```

As cores são: 'b'=blue, 'g'=green, 'r'=red, 'c'=ciano, 'm'=magenta, 'y'=yellow, 'k'=black e 'w'=white. Os marcadores são '.'=ponto, 'o'=bola, 'v'=triângulo, "=triângulo, '<'=triângulo, '>'=triângulo, 's'=quadrado, 'p'=pentágono, '*'=estrela, '+'=mais, 'x'=vezes, 'D'=diamante, 'd'=diamante estreito, '|'=linha vertical, '_'=linha horizontal, '-'=linha sólida, '–'=linha picotada, '-.'=linha ponto, ':'=linha pontilhada (tem mais...)

18.5 Um exemplo bobinho

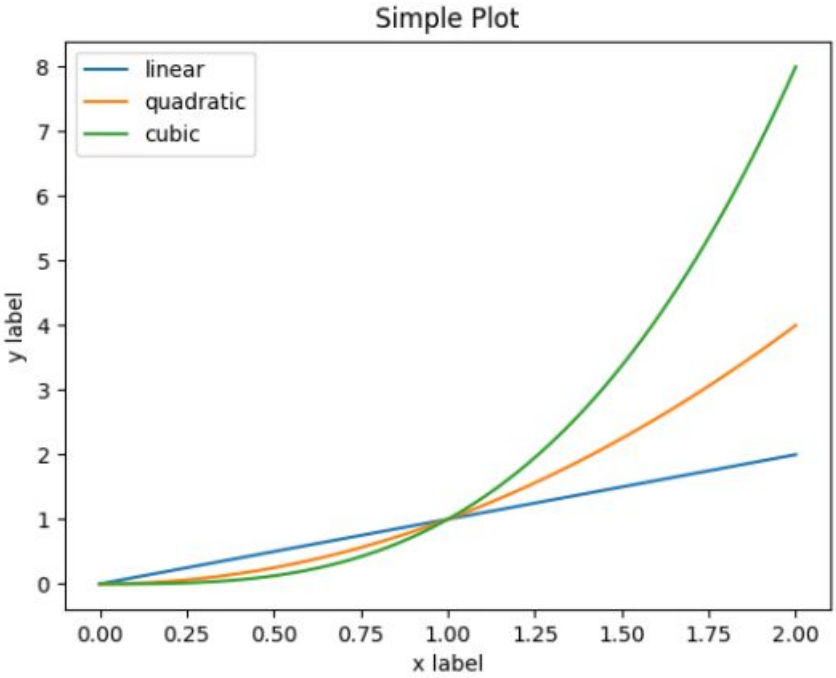
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

`ro` significa 'red circles'. `b-` seria risco azul. `r--` é red dashes, `bs` é blue squares e `g^` é green triangles.



18.6 Um exemplo de plotagem simples

```
x = np.linspace(0, 2, 100) # pontos espaçados
plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
plt.show()
```



18.7 Tortas

Eis o formato do comando `pie`

```
matplotlib.pyplot.pie(x, explode=None,
labels=None, colors=None, autopct=None,
pct-distance=0.6, shadow=False,
labeldistance=1.1, startangle=None,
radius=None, counterclock=True,
wedgeprops=None, textprops=None,
center=(0, 0), frame=False,
rotatelabels=False, hold=None, data=None)
```

As fatias estão dadas por `x`. Cada fatia ocupa $x/sum(x)$. Se a soma de x é menor do que 1, os valores de x indicam a fração e não são normalizados. As fatias são desenhadas em sentido anti-horário a partir do eixo x.

explode O parâmetro *explode* se não for *None* será um array de tamanho $len(x)$ indicando qual a fração do raio que cada fatia deve ser deslocada.

labels é uma sequencia de strings indicando o que é cada fatia.

colors é uma sequencia de cores. Se usado *None* usa-se a sequencia padrão.

autopct Pode ser *None*, um string ou uma função. Diz para identificar a fatia com um número. Se for uma função ela será chamada. A *pctdistance* indica onde por este rótulo.

shadow é um booleano, indica se haverá sombra.

labeldistance o default é 1.1 e indica a distância radial onde os labels vão ser desenhados.

startangle onde começa a primeira fatia (a partir do eixo x).

radius O raio da pizza. Se *None* vale 1.

counterclock Booleano, indica o sentido antihorário ou não.

wedgeprops é um dicionário (como em `wedgeprops = {'linewidth': 3}` . Para saber mais veja a lista de argumentos do objeto `wedge`.

textprops é um dicionário com atributos de texto.

center a localização do centro da pizza (default 0,0)

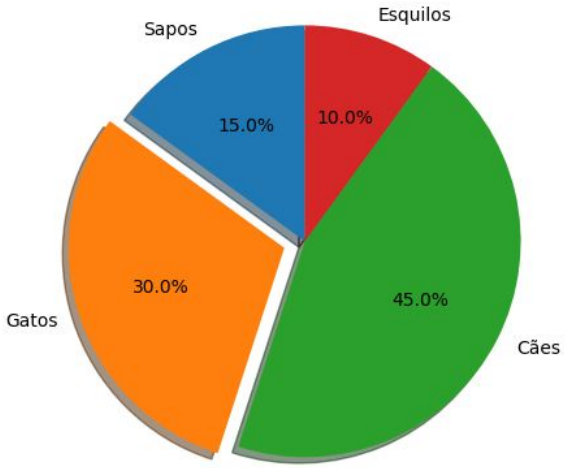
frame Booleano. Se true desenha um frame.

rotatelabels Booleano: rotaciona cada fatia no ângulo da fatia.

Retornos se chamada com *get* retorna uma lista de *wedges* outra de *texts* e de *autotexts* se autopct não é *None*.

startangle

```
import matplotlib.pyplot as plt
# Pie chart, where the slices will be
ordered and plotted counter-clockwise:
labels = 'Sapos', 'Gatos', 'Cães', 'Esquilos'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # only "explode" t
                    he 2nd slice (i.e. 'Gatos')
fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels,
    autopct='%1.1f%%',shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio
         #ensures that pie is drawn as a circle.
plt.show()
```



18.8 Histogramas

Desenha um histograma. Seu formato

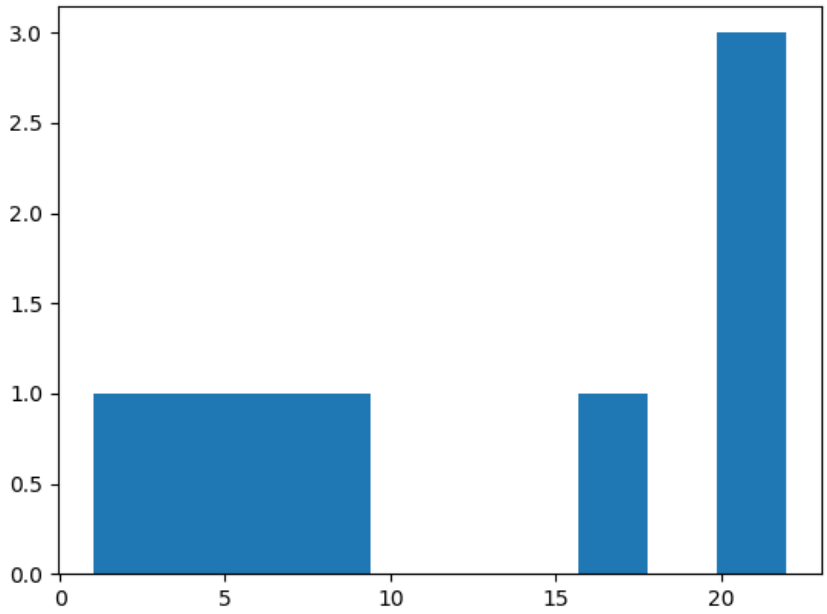
```
hist(x, bins=None, range=None,
density=None, weights=None,
cumulative=False, bottom=None,
histtype='bar', align='mid',
orientation='vertical', rwidth=None,
log=False, color=None, label=None,
stacked=False, normed=None, hold=None,
data=None, **kwargs)
```

x pode ser um array ou uma sequencia de arrays, que não precisam ter o mesmo comprimento.

Nota $**kwargs$ é aqui (como em todo o matplotlib) uma lista adicional (e opcional) de palavras chave e argumentos que podem controlar mais aspectos do que está sendo pedido. Veja a documentação, a respeito.

Veja um exemplo bem bobo:

```
import matplotlib.pyplot as plt
def hist1():
  x=[1,4,7,8,16,22,22,22]
  plt.hist(x)
  plt.show()
  plt.savefig('c:/p/python/hist2.png')
hist1()
```



18.9 Um exemplo

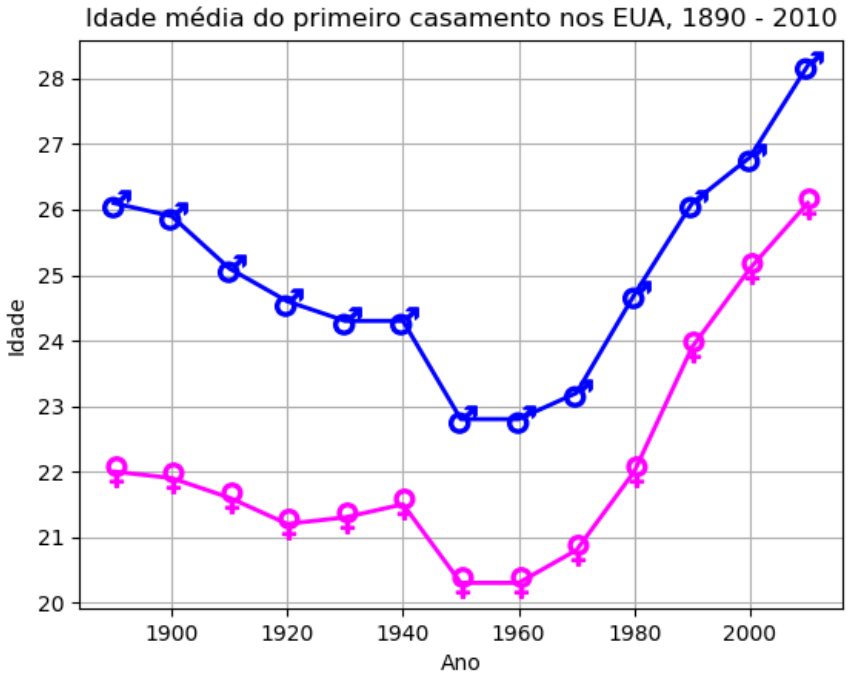
Eis o arquivo

```
Median age at First Marriage, 1890-2010
Source: U.S. Bureau of the Census, www.census.gov
Ano    Homens   Mulheres
1890     26.1     22.0
1900     25.9     21.9
1910     25.1     21.6
1920     24.6     21.2
1930     24.3     21.3
1940     24.3     21.5
1950     22.8     20.3
1960     22.8     20.3
1970     23.2     20.8
1980     24.7     22.0
1990     26.1     23.9
2000     26.8     25.1
2010     28.2     26.1
```

```
import numpy as np
import matplotlib.pyplot as plt
year, age_m, age_f = np.loadtxt
  ('c:/p/python/casamentos.txt',
  unpack=True,skiprows=3)
fig = plt.figure()
ax = fig.add_subplot(111)
# Plot ages with male or female symbols as markers
ax.plot(year, age_m, marker='$\u2642$',
        markersize=14, c='blue', lw=2,
        mfc='blue', mec='blue')
ax.plot(year, age_f, marker='$\u2640$',
        markersize=14, c='magenta', lw=2,
        mfc='magenta', mec='magenta')
ax.grid()
ax.set_xlabel('Ano')
ax.set_ylabel('Idade')
ax.set_title('Idade média do primeiro
```

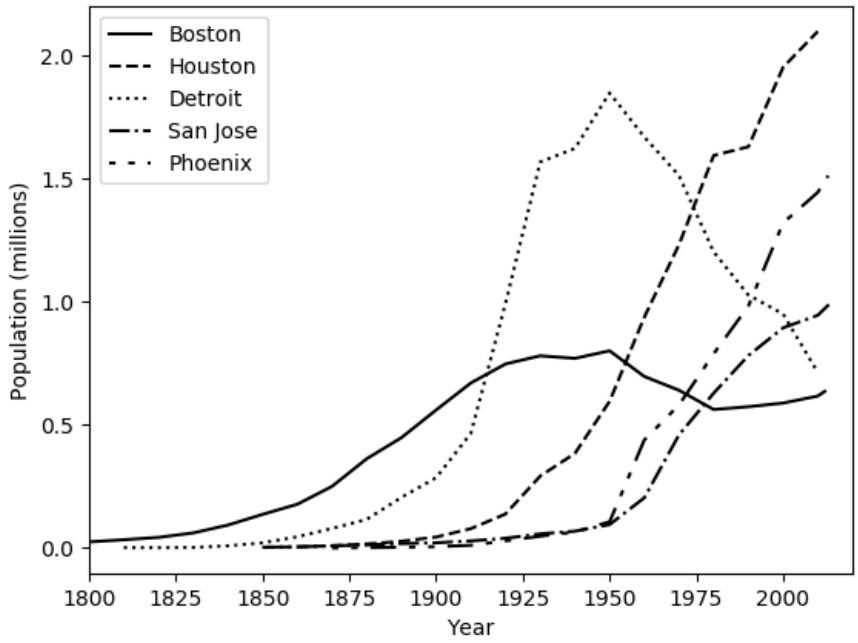


```
    casamento nos EUA, 1890 - 2010')
plt.savefig('c:/p/python/exe1.png')
plt.show()
```



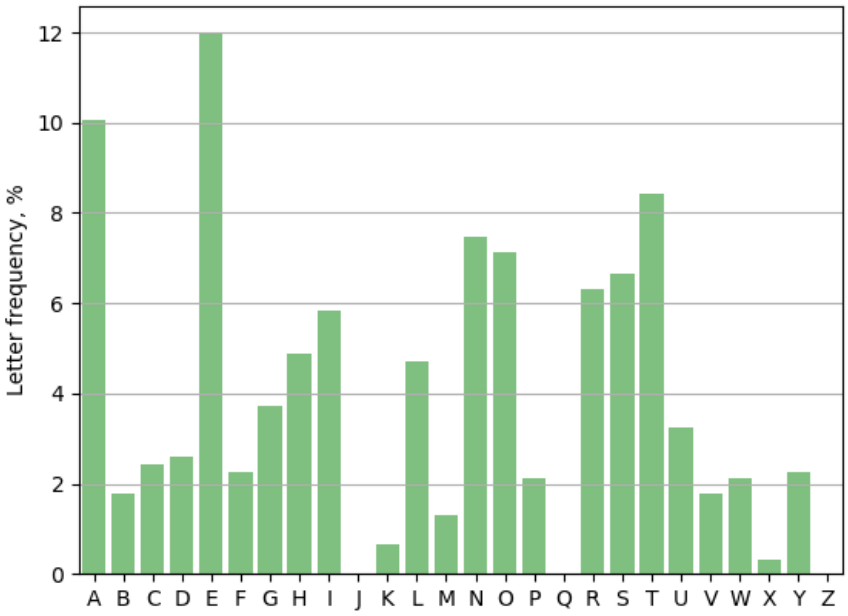
18.10 Exemplo

```
fig = plt.figure()
ax = fig.add_subplot(111)
cities = ['Boston', 'Houston',
  'Detroit', 'San Jose', 'Phoenix']
  linestyles = [{'ls': '-'}, {'ls': '--'},
  {'ls': ':'}, {'ls': '-.'},
  {'dashes': [2, 4, 2, 4, 8, 4]}]
for i, city in enumerate(cities):
    filename = '{}.tsv'.format(city.
lower()).replace(' ', '_')
    yr, pop = np.loadtxt(filename,
unpack=True)
    line, = ax.plot(yr, pop/1.e6,
label=city, color='k', **linestyles[i])
ax.legend(loc='upper left')
ax.set_xlim(1800, 2020)
ax.set_xlabel('Year')
ax.set_ylabel('Population (millions)')
plt.show()
```



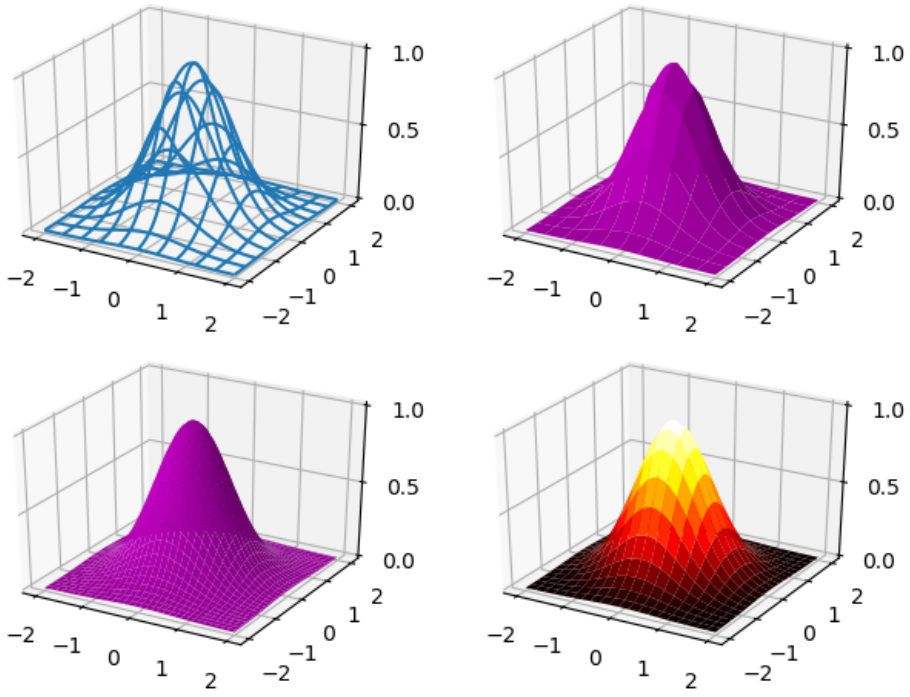
18.11 Exemplo

```
import numpy as np
import matplotlib.pyplot as plt
text_file = 'c:/p/python/mobydick.txt'
letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
lcount = dict([(l, 0) for l in letters])
for l in open(text_file).read():
    try:
        lcount[l.upper()] += 1
    except KeyError:
        pass
norm = sum(lcount.values())
fig = plt.figure()
ax = fig.add_subplot(111)
x = range(26)
ax.bar(x, [lcount[l]/norm * 100 for l in letters], width=0.8,
       color='g', alpha=0.5, align='center')
ax.set_xticks(x)
ax.set_xticklabels(letters)
ax.tick_params(axis='x', direction='out')
ax.set_xlim(-0.5, 25.5)
ax.yaxis.grid(True)
ax.set_ylabel('Letter frequency, %')
plt.show()
```



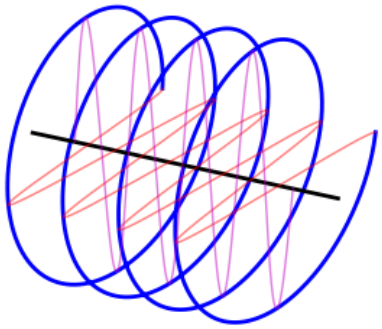
18.12 Exemplo

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm

L, n = 2, 400
x = np.linspace(-L, L, n)
y = x.copy()
X, Y = np.meshgrid(x, y)
Z = np.exp(-(X**2 + Y**2))

fig, ax = plt.subplots(nrows=2, ncols=2, subplot_kw=
   {'projection': '3d'})
ax[0,0].plot_wireframe(X, Y, Z, rstride=40, cstride=40)
ax[0,1].plot_surface(X, Y, Z, rstride=40, cstride=40, color='m')
ax[1,0].plot_surface(X, Y, Z, rstride=12, cstride=12, color='m')
ax[1,1].plot_surface(X, Y, Z, rstride=20, cstride=20, cmap=cm.hot)
for axes in ax.flatten():
    axes.set_xticks([-2, -1, 0, 1, 2])
    axes.set_yticks([-2, -1, 0, 1, 2])
    axes.set_zticks([0, 0.5, 1])
fig.tight_layout()
plt.show()
```



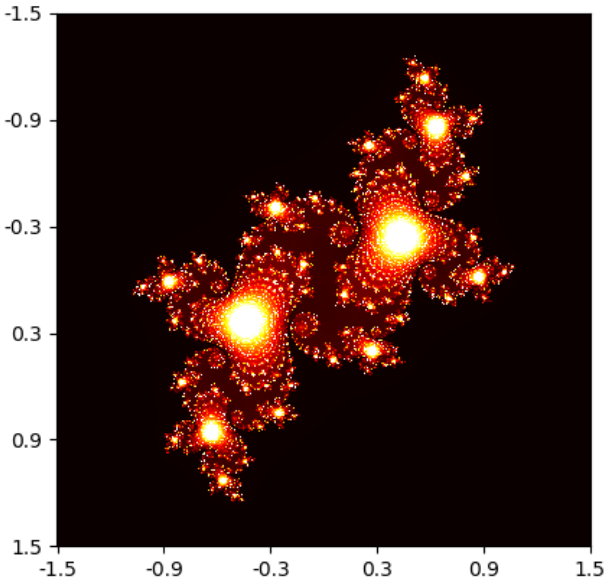
18.13 Exemplo

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

n = 1000
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot a helix along the x-axis
theta_max = 8 * np.pi
theta = np.linspace(0, theta_max, n)
x = theta
z =  np.sin(theta)
y =  np.cos(theta)
ax.plot(x, y, z, 'b', lw=2)

# An line through the centre of the helix
ax.plot((-theta_max*0.2, theta_max * 1.2), (0,0), (0,0), color='k', lw=2)
# sin/cos components of the helix (e.g. electric and magnetic field
# components of a circularly-polarized electromagnetic wave
ax.plot(x, y, 0, color='r', lw=1, alpha=0.5)
ax.plot(x, [0]*n, z, color='m', lw=1, alpha=0.5)

# Remove axis planes, ticks and labels
ax.set_axis_off()
plt.show()
```



18.14 Julia Set

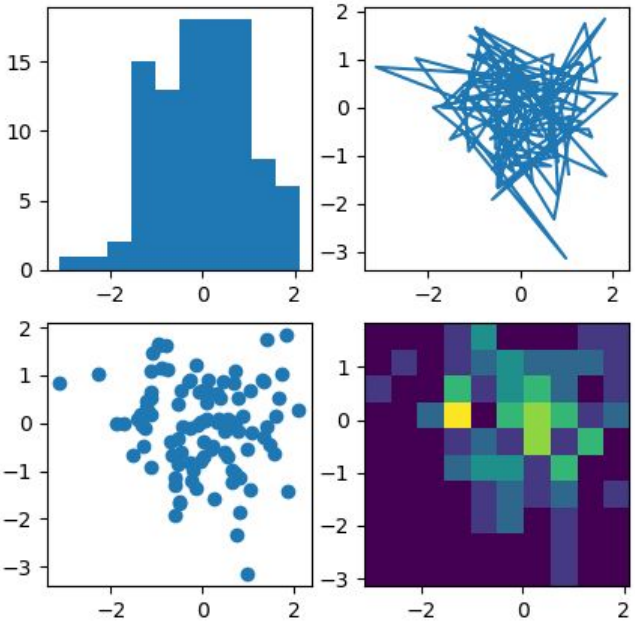
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# Image width and height; parameters for the plot
im_width, im_height = 500, 500
c = complex(-0.1, 0.65)
zabs_max = 10
nit_max = 1000
xmin, xmax = -1.5, 1.5
xwidth = xmax - xmin
ymin, ymax = -1.5, 1.5
yheight = ymax - ymin

julia = np.zeros((im_width, im_height))
for ix in range(im_width):
    for iy in range(im_height):
        nit = 0
        # Map pixel position to a point in the complex plane
        z = complex(ix / im_width * xwidth + xmin,
                    iy / im_height * yheight + ymin)
        # Do the iterations
        while abs(z) <= zabs_max and nit < nit_max:
            z = z**2 + c
            nit += 1
        shade = 1-np.sqrt(nit / nit_max)
        ratio = nit / nit_max
        julia[ix,iy] = ratio

fig, ax = plt.subplots()
ax.imshow(julia, interpolation='nearest', cmap=cm.hot)
# Set the tick labels to the coordinates of z0 in the complex plane
xtick_labels = np.linspace(xmin, xmax, xwidth / 0.5)
ax.set_xticks([(x-xmin) / xwidth * im_width for x in xtick_labels])
ax.set_xticklabels(['{:.1f}'.format(xtick) for xtick in xtick_labels])
ytick_labels = np.linspace(ymin, ymax, yheight / 0.5)
ax.set_yticks([(y-ymin) / yheight * im_height for y in ytick_labels])
ax.set_yticklabels(['{:.1f}'.format(ytick) for ytick in ytick_labels])
plt.show()
```



18.15 Um exemplo de subplot

Para escrever mais de um gráfico em um único espaço, usa-se subplot(). Veja a seguir um exemplo.

```
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(19680801)
data = np.random.randn(2, 100)
fig, axs = plt.subplots(2, 2, figsize=(5, 5))
axs[0, 0].hist(data[0])
axs[1, 0].scatter(data[0], data[1])
axs[0, 1].plot(data[0], data[1])
axs[1, 1].hist2d(data[0], data[1])
plt.show()
```

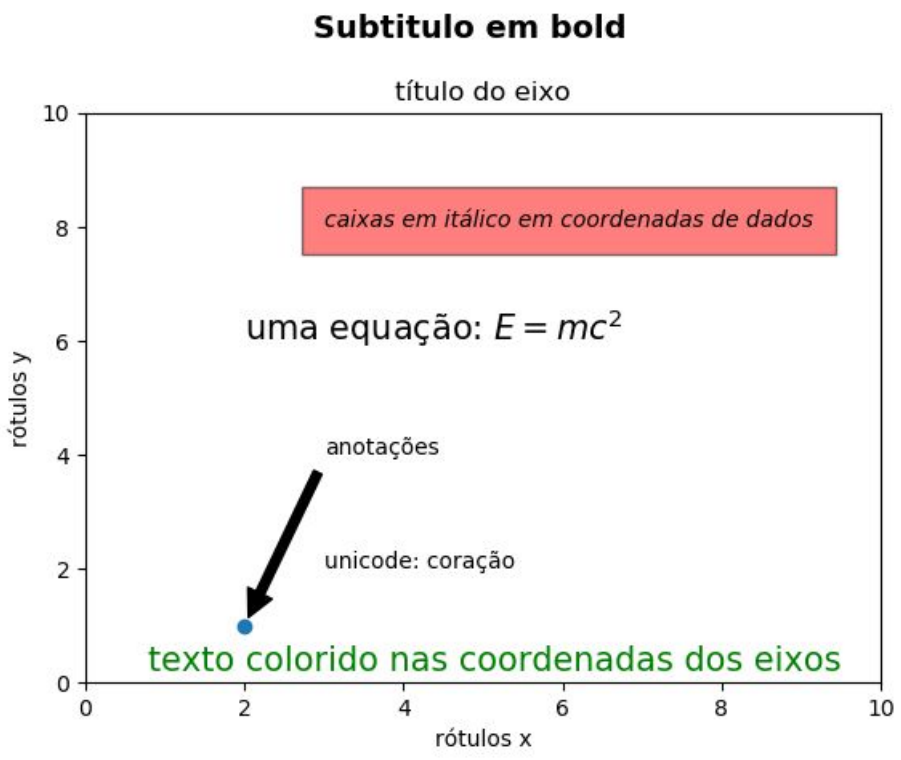


18.16 Interativo ou não

Fazendo `plt.ion()` tudo o que você manda fazer ele faz e mostra na janela. Fazendo `plt.ioff()` você faz tudo e no final tem que dar um `plt.show()` para ver o resultado.

18.17 Escrevendo texto

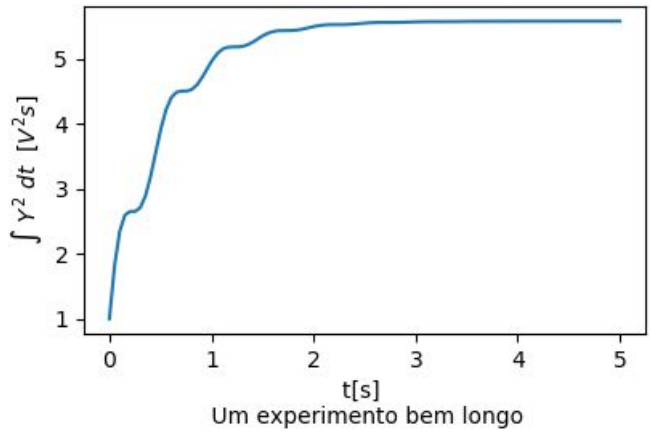
```
import matplotlib
import matplotlib.pyplot as plt
fig = plt.figure()
fig.suptitle('Subtitulo em bold', fontsize=14,
     fontweight='bold')
ax = fig.add_subplot(111)
fig.subplots_adjust(top=0.85)
ax.set_title('título do eixo')
ax.set_xlabel('rótulos x')
ax.set_ylabel('rótulos y')
ax.text(3, 8, 'caixas em itálico em coordenadas de
       dados', style='italic',
bbox={'facecolor': 'red', 'alpha': 0.5, 'pad': 10})
ax.text(2, 6, r'uma equação: $E=mc^2$', fontsize=15)
ax.text(3, 2, u'unicode: coração')
ax.text(0.95, 0.01, 'texto colorido nas coordenadas
       dos eixos',
verticalalignment='bottom',
horizontalalignment='right',
transform=ax.transAxes,
color='green', fontsize=15)
ax.plot([2], [1], 'o')
ax.annotate('anotação', xy=(2, 1), xytext=(3, 4),
arrowprops=dict(facecolor='black', shrink=0.05))
ax.axis([0, 10, 0, 10])
plt.show()
```



18.18 Escrevendo latex

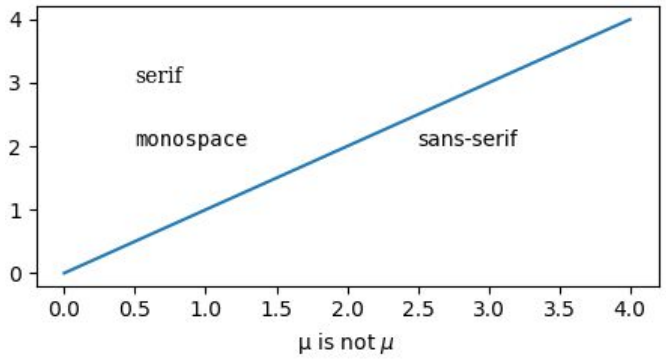
```
import matplotlib.pyplot as plt
import numpy as np
x1 = np.linspace(0.0, 5.0, 100)
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
fig, ax = plt.subplots(figsize=(5, 3))
fig.subplots_adjust(bottom=0.2, left=0.2)
```

```
ax.plot(x1, np.cumsum(y1**2))
ax.set_xlabel('t[s] \n Um experimento bem longo')
ax.set_ylabel(r'$\int\ Y^2\ dt\ \ [V^2 s]$')
plt.show()
```



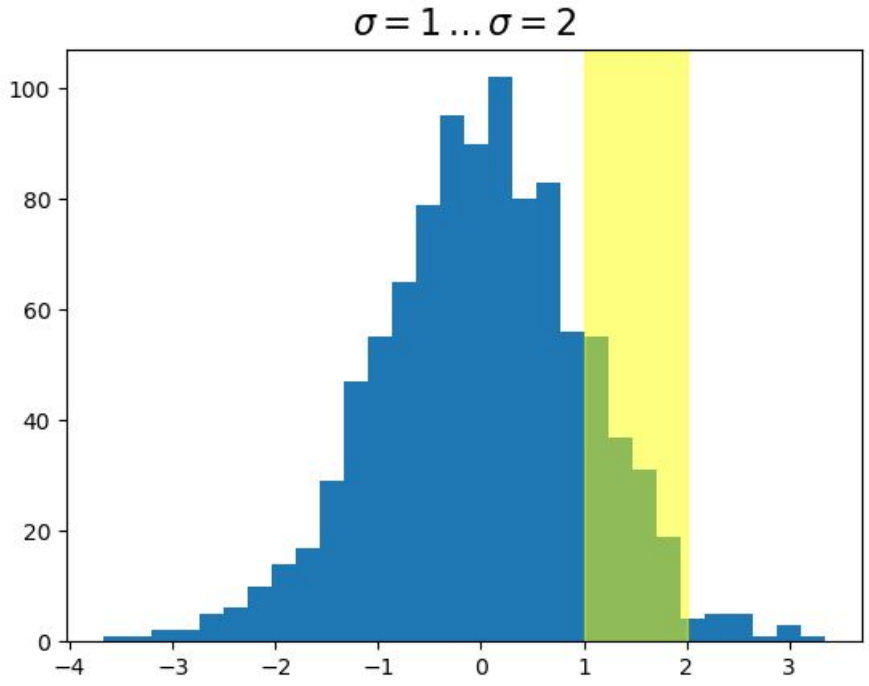
18.19 Mais texto

```
import matplotlib.pyplot as plt
plt.figure(figsize=(4.5, 2.5))
plt.plot(range(5))
plt.text(0.5, 3.,"serif", family="serif")
plt.text(0.5, 2.,"monospace",family="monospace")
plt.text(2.5, 2.,"sans-serif",family="sans-serif")
plt.xlabel(u"μ is not $\\mu$")
plt.tight_layout(.5)
plt.show()
```



18.20 mais

```
import matplotlib.transforms as transforms
import matplotlib.patches as patches
fig = plt.figure()
ax = fig.add_subplot(111)
x = np.random.randn(1000)
ax.hist(x, 30)
ax.set_title(r'$\sigma=1 \/ \dots \/ \sigma=2$',
fontsize=16)
trans = transforms.blended_transform_factory(
ax.transData, ax.transAxes)
rect = patches.Rectangle((1, 0), width=1, height=1,
transform=trans, color='yellow',
alpha=0.5)
ax.add_patch(rect)
plt.show()
```

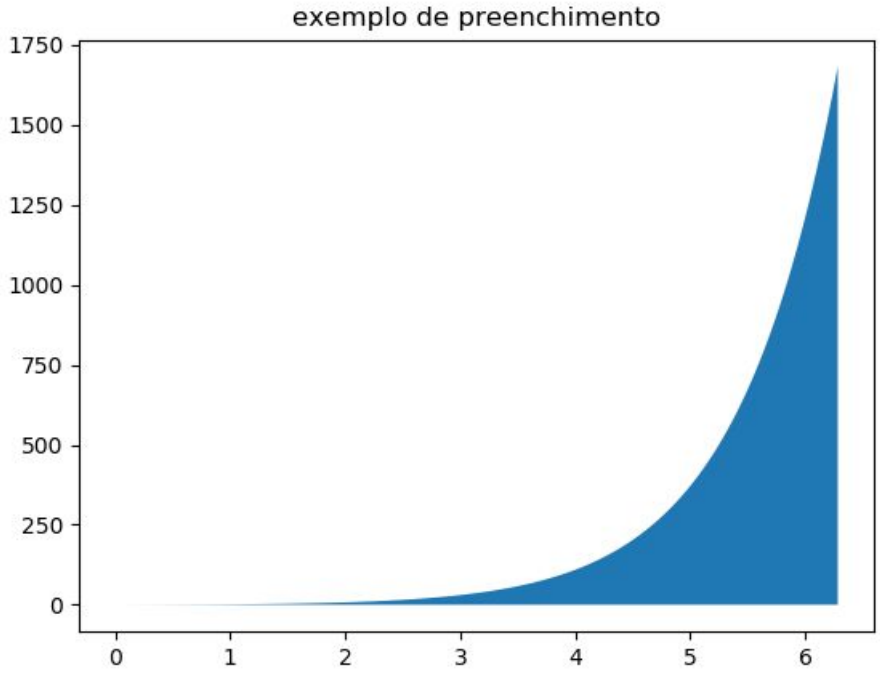
18.21 Preenchimento

Para fazer o preenchimento entre uma variável (aqui `ga` e o zero, faça:

```
axe.fill_between(range(12),ga,0)
```

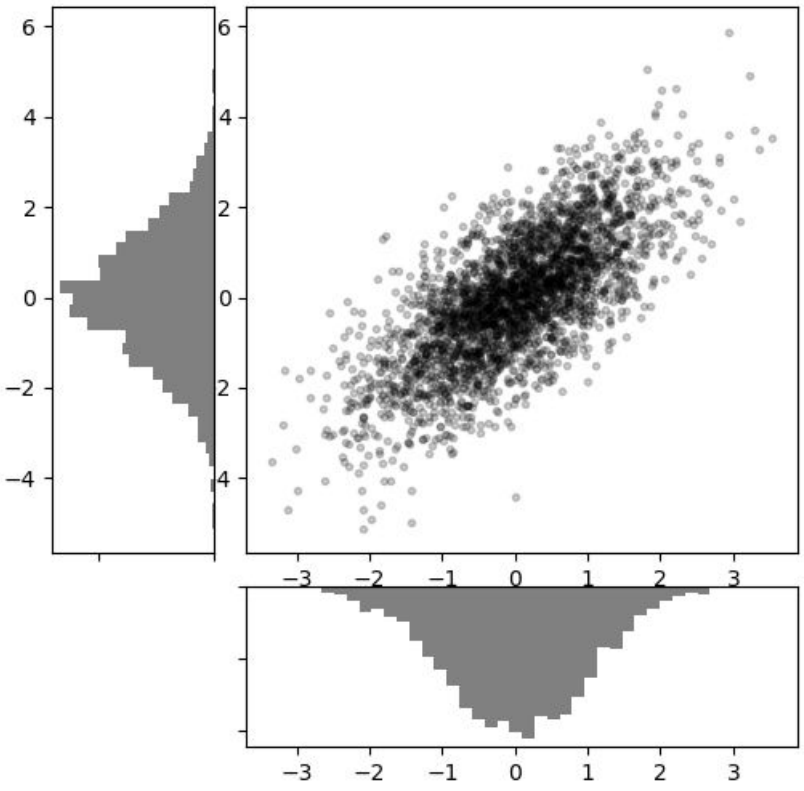
como no exemplo a seguir:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2*np.pi, 400)
y = np.exp(x)*(x/2)
yy=np.zeros((400))
fig, ax = plt.subplots()
plt.title('exemplo de preenchimento')
ax.fill_between(x,y,yy)
plt.show()
```



18.22 Exemplo com grid

```
import matplotlib.pyplot as plt
import numpy as np

# Create some normally distributed data
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean,
     cov, 3000).T
# Set up the axes with gridspec
fig = plt.figure(figsize=(6, 6))
grid=plt.GridSpec(4,4,hspace=0.2,wspace=0.2)
main_ax = fig.add_subplot(grid[:-1, 1:])
y_hist = fig.add_subplot(grid[:-1, 0],
     xticklabels=[], sharey=main_ax)
x_hist = fig.add_subplot(grid[-1, 1:],
     yticklabels=[], sharex=main_ax)
# scatter points on the main axes
main_ax.plot(x,y,'ok',markersize=3,alpha=0.2)
# histogram on the attached axes
x_hist.hist(x, 40, histtype='stepfilled',
      orientation='vertical', color='gray')
x_hist.invert_yaxis()
y_hist.hist(y, 40, histtype='stepfilled',
      orientation='horizontal', color='gray')
y_hist.invert_xaxis()
plt.show()
```



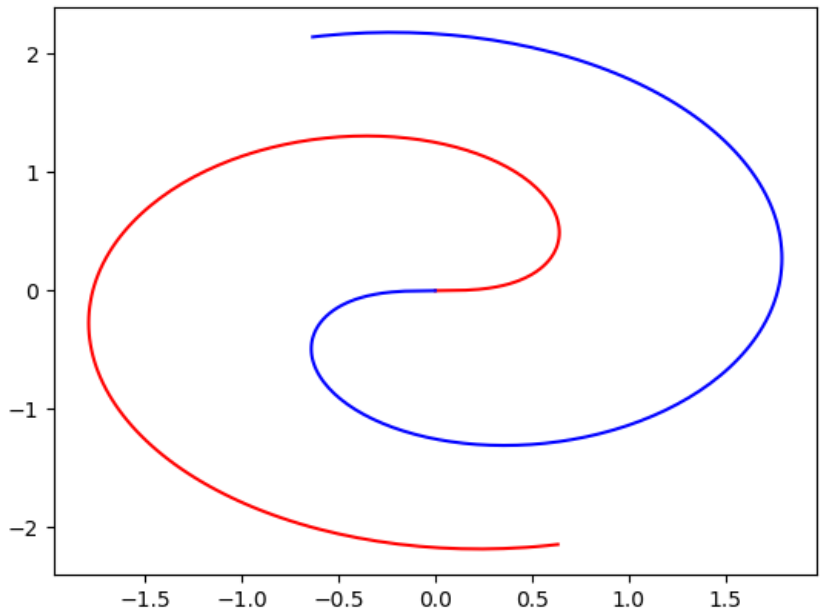
18.23 Espiral de Fermat

Em coordenadas polares:
$r = \theta^{1/2}$ ou ainda $r^2 = a^2 theta$

```
#fermaspi
import matplotlib.pyplot as plt
```



```
import numpy as np
def cart2pol(x, y):
    rho = np.sqrt(x**2 + y**2)
    phi = np.arctan2(y, x)
    return(rho, phi)

def pol2cart(rho, phi):
    x = rho * np.cos(phi)
    y = rho * np.sin(phi)
    return(x, y)
ta=200
t=np.linspace(0.0, 5, ta)
x=np.zeros((ta))
y=np.zeros((ta))
i=0
a=1
while(i<ta):
  (xa,ya)=pol2cart((a*t[i]**0.5),t[i])
  x[i]=xa
  y[i]=ya
  i=i+1
plt.plot(x,y,'r-')
i=0
a=-1
while(i<ta):
  (xa,ya)=pol2cart((a*t[i]**0.5),t[i])
  x[i]=xa
  y[i]=ya
  i=i+1
plt.plot(x,y,'b-')
plt.show()
```



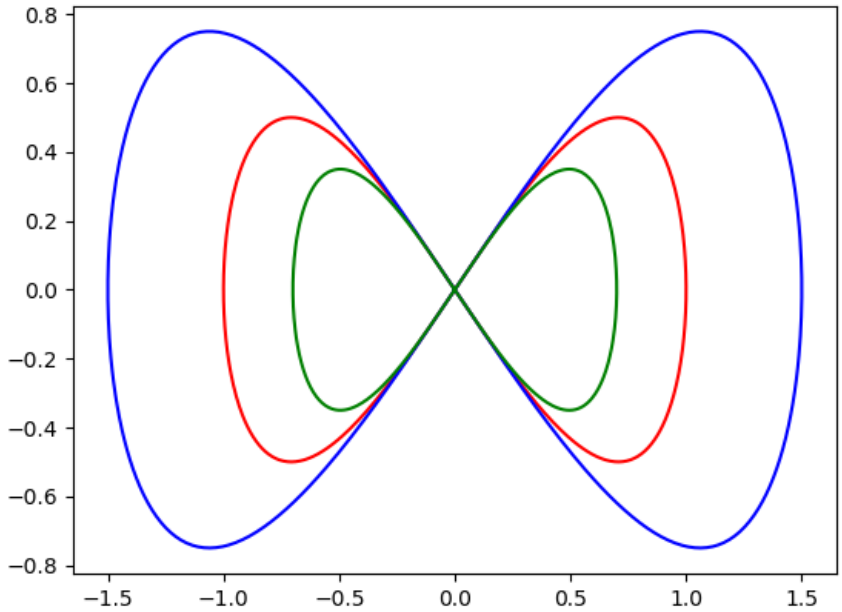
18.24 Lemniscata

Em coordenadas cartesianas:
$(x^2 + y^2)^2 = 2a^2(x^2 - y^2)$

Em coordenadas polares:

$r^2 = 2a^2 cos2\theta$

```
#lemniscata
import matplotlib.pyplot as plt
import numpy as np
ta=500
t=np.linspace(0.0, 6.28, ta)
x=np.zeros((ta))
y=np.zeros((ta))
i=0
a=1
while(i<ta):
  x[i]=a*np.sin(t[i])
  y[i]=a*np.sin(t[i])*np.cos(t[i])
  i=i+1
plt.plot(x,y,'r-')
i=0
a=1.5
while(i<ta):
  x[i]=a*np.sin(t[i])
  y[i]=a*np.sin(t[i])*np.cos(t[i])
  i=i+1
plt.plot(x,y,'b-')
i=0
a=0.7
while(i<ta):
  x[i]=a*np.sin(t[i])
  y[i]=a*np.sin(t[i])*np.cos(t[i])
  i=i+1
plt.plot(x,y,'g-')
plt.show()
```

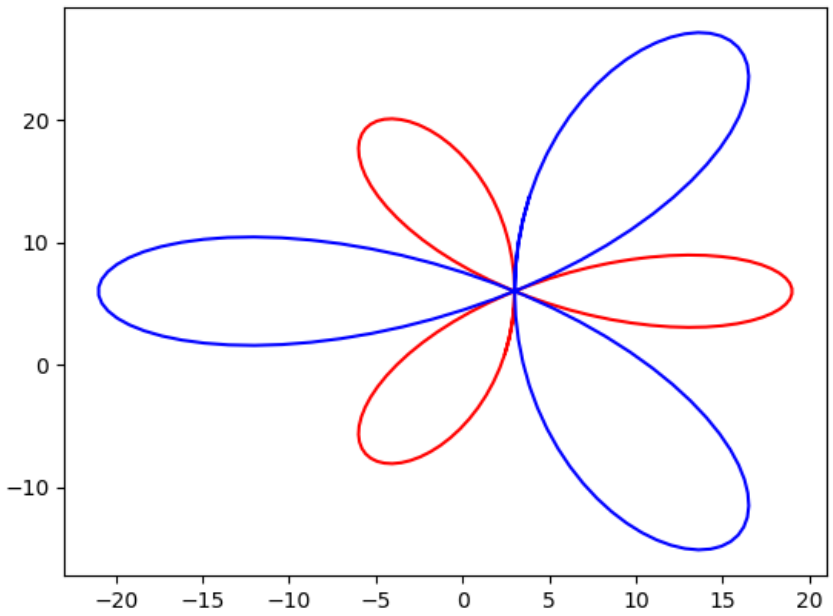


18.25 trifóide

```
#trifoide
import matplotlib.pyplot as plt
import numpy as np
fi=np.linspace(0.0, 6.5, 120)
x=np.zeros((120))
```



```
y=np.zeros((120))
i=0
a=-2
x0=3
y0=6
while(i<120):
  x[i]=x0+4*a*np.cos(fi[i])-4*a*np.cos(2*fi[i])
  y[i]=y0+4*a*np.sin(fi[i])+4*a*np.sin(2*fi[i])
  i=i+1
plt.plot(x,y,'r-')
i=0
a=3
x0=3
y0=6
while(i<120):
  x[i]=x0+4*a*np.cos(fi[i])-4*a*np.cos(2*fi[i])
  y[i]=y0+4*a*np.sin(fi[i])+4*a*np.sin(2*fi[i])
  i=i+1
plt.plot(x,y,'b-')
plt.show()
```



18.26 cardióide

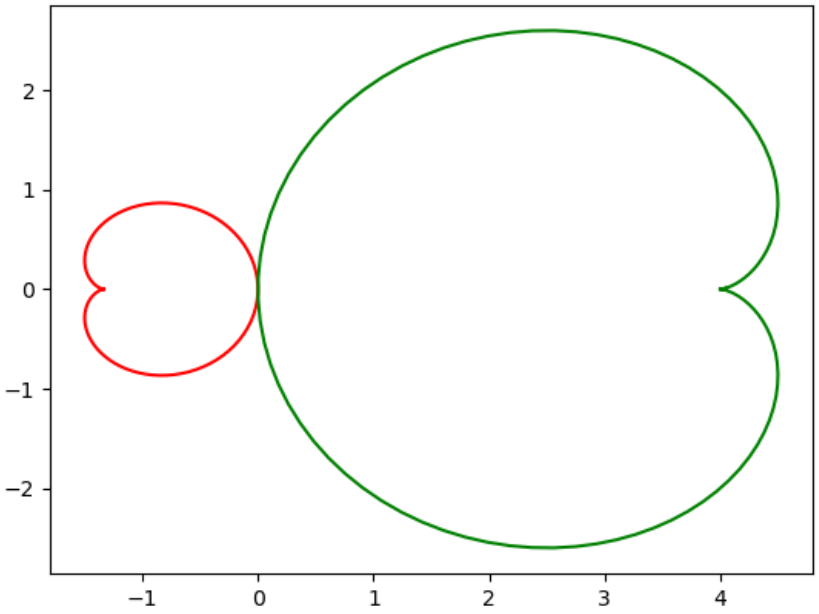
Representação paramétrica:
$x(\varphi) = 2a(1 - cos\varphi).cos\varphi$
$y(\varphi) = 2a(1 - cos\varphi).sin\varphi \qquad 0 \leq \varphi < 2\pi$

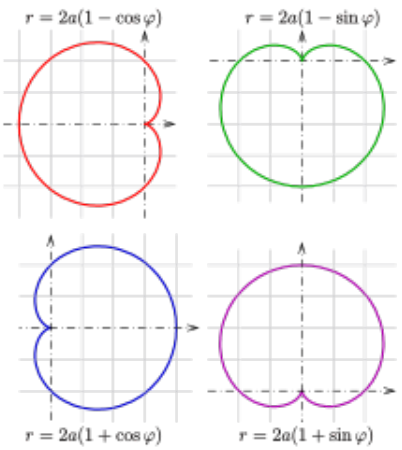
Em coordenadas polares:
$r(\varphi) = 2a(1 - cos\varphi)$

Em coordenadas cartesianas:
$(x^2 + y^2)^2 + 4ax(x^2 + y^2) - 4a^2y^2 = 0$

```
#cardioide
import matplotlib.pyplot as plt
import numpy as np
teta=np.linspace(0.0, 6.5, 120)
x=np.zeros((120))
```

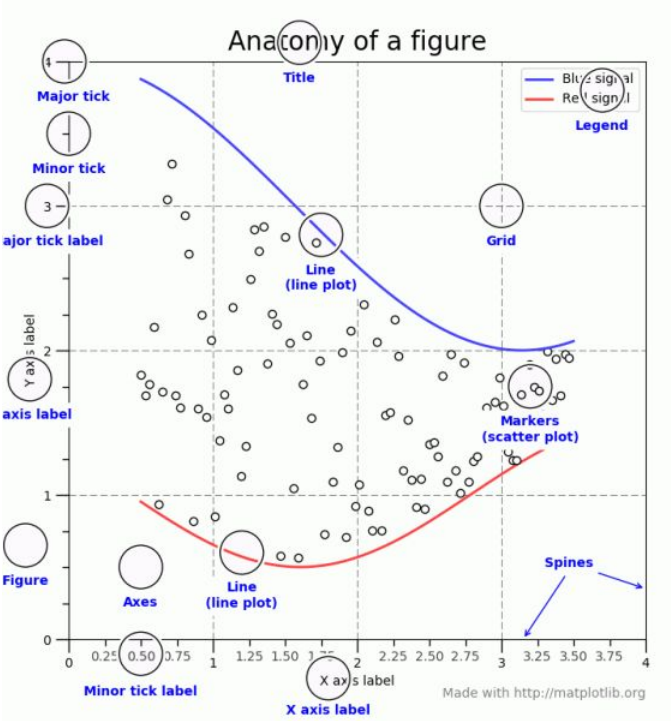
```
y=np.zeros((120))
i=0
a1=-2
while(i<120):
  x[i]=((2/3)*a1)+((1/3)*a1)*np.cos(teta[i])*(1-np.cos(teta[i]))
  y[i]=((1/3)*a1)*np.sin(teta[i])*(1-np.cos(teta[i]))
  i=i+1
plt.plot(x,y,'r-')
a2=6
i=0
while(i<120):
  x[i]=((2/3)*a2)+((1/3)*a2)*np.cos(teta[i])*(1-np.cos(teta[i]))
  y[i]=((1/3)*a2)*np.sin(teta[i])*(1-np.cos(teta[i]))
  i=i+1
plt.plot(x,y,'g-')
plt.show()
```





Para desenhar a cardioide em outras posições, use:

18.27 Anatomia de uma figura



Capítulo 19

Pacote: OpenCV

Definição

O pacote OCV (*Open Source Computer Vision*) agrega funções para processamento e análise de imagens. O pacote foi desenvolvido originalmente em C++ e rapidamente migrado para Python. Usa como pré-requisitos os seguintes pacotes

- Numpy: numerical python
- Matplotlib: gerador de gráficos no estilo Matlab

Não precisa se preocupar, se o instalador detectar que estes pacotes não estão presentes ele (o instalador) vai carregar e instalar estes dois pacotes.

A instalação do openCV se dá pelo comando

```
pip install opencv-python
```

emitido no diretório onde estiver instalado o pip.exe. Tipicamente é

```
cd\python\scripts
```

Depois de instalar o pacote, teste o funcionamento do mesmo dentro do python emitindo:

```
>>>import cv2
```

Se o python responder `>>>` sem nenhuma outra mensagem, é porque a instalação deu certo.

Algumas funções

A seguir uma lista de algumas funções do pacote

Função	o que faz	forma de chamar
Ler uma imagem	carrega para dentro do programa uma ou mais matrizes (vide canal)	`im=cv2.imread('arquivo.jpg',dica)`

`dica` é opcional e indica como a leitura deve ser feita. Seus valores:

1 leia o arquivo em true color

0 leia o arquivo em niveis de cinza

-1 leia o arquivo como ele foi criado

Após este comando, a variável `im` tem associada as seguintes informações:

variável	conteúdo
`im.shape[0]`	altura da imagem em pixels
`im.shape[1]`	largura da imagem em pixels
`im.shape[2]`	quantidade de canais da imagem

Função	o que faz	forma de chamar
Mostrar uma imagem	abre uma janela e escreve nela a imagem	`im=cv2.imshow('nome da janela',im)`
Gravar a imagem em disco	grava a imagem (eventualmente alterada)	`cv2.imwrite("arquivo.jpg",im)`

Se a imagem for branco e preto, haverá um único canal (uma matriz com o nome de `im`. Se a imagem for true color, haverá 3 matrizes de mesmo tamanho, a primeira sendo `blue` a segunda `green` e a terceira `red`. Acompanhe o programa abaixo

```
def ima1():
  import cv2
  imagem=cv2.imread('jovem.jpg')
  for i in range(0,imagem.shape[0],10):
    for j in range(0,imagem.shape[1],10):
#      (a,b,c)=imagem[i,j]
      imagem[i:i+5,j:j+5]=(0,255,255)
  cv2.imshow('olhe',imagem)
ima1()
```

Função	o que faz	forma de chamar
Desenha uma linha	desenha uma linha na imagem	cv2.line(im, (origem), (destino), cor)
Desenha um retângulo vazado	cria um retângulo se largura = -1, figura é cheia. Default é 1.	cv2.rectangle(im,(esqsup),(dirinf0, cor, largura-risco)
Desenha circulo	desenha um circulo	cv2.circle(im, (centro), raio, cor)

Acompanhe o programa

```
import numpy as np
import cv2
im = cv2.imread('oba.jpg')
vermelho = (0,0,255)
verde=(0,255,0)
azul=(255,0,0)
cv2.line(im,(5,5),(30,60), verde)
cv2.rectangle(im,(20,30),(100,120),azul,-1)
(X, Y) = (im.shape[1]//2, im.shape[0]//2)
cv2.circle(im,(X,Y),30,azul)
```

Para recortar (crop) uma imagem, faça:

```
import cv2
im=cv2.imread('oba.jpg')
recorte = im[100:200, 100:200]
cv2.imshow("Recorte",recorte)
cv2.imwrite("recorte.jpg",recorte)
```

Função	o que faz	forma de chamar
redimensionar	redimensiona uma imagem	imn=cv2.resize(im,(tamanho novo), interpolation=método)

o método pode ser:

cv2.INTER_AREA for shrinking

cv2.INTER_CUBIC (slow)

cv2.INTER_LINEAR for zooming.

Por default o método usado é `INTER_LINEAR`. Veja o código

```
import numpy as np
import cv2
im=cv2.imread('oba.jpg')
largura=im.shape[1]
altura=im.shape[0]
proporcao=float(altura/largura)
largura_nova=500
altura_nova=int(largura_nova*proporcao)
tamanho_novo=(largura_nova, altura_nova)
imredimens=cv2.resize(im,tamanho_novo,interpolation=cv2.INTER_AREA)
cv2.imshow('Resultado',imredimens)
cv2.waitKey(0)
```

A seguir, como fazer flip vertical e horizontal

```
import cv2
im = cv2.imread('oba.jpg')
flip_hor = im[::-1,:]
flip_vert = im[:,::-1]
flip_ambos = im[::-1,::-1]
... gravar, mostrar, etc
```

Para rotar uma imagem:

```
def ri(): #roda imagem
  import cv2
  im=cv2.imread("pedro_jovem.jpg")
  (alt,lar)=im.shape[:2]
  centro=(lar//2,alt//2)
  m=cv2.getRotationMatrix2D(centro,-10,1.0) #-10 graus
  imr=cv2.warpAffine(im,m,(lar,alt))
  cv2.imshow("rodada",imr)
  cv2.waitKey(0)
ri()
```


Capítulo 20

Pacote: TKInter

Este pacote permite criar software Python com interface GUI (cara de aplicativo windows). Eis aqui uma aplicação simples

```
#!/usr/bin/env python3
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        tk.Frame.__init__(self, master)
        self.grid()
        self.createWidgets()

    def createWidgets(self):
        self.quitButton = tk.Button(self, text='Tchal', command=self.quit)
        self.quitButton.grid()

app = Application()
app.master.title('Aplicação simples')
app.mainloop()
```

E uma segunda

```
from tkinter import *

class Application(Frame):
    def say_hi(self):
        print("hi there, everyone!")

    def createWidgets(self):
        self.QUIT = Button(self)
        self.QUIT["text"] = "QUIT"
        self.QUIT["fg"]   = "red"
        self.QUIT["command"] =  self.quit

        self.QUIT.pack({"side": "left"})

        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi

        self.hi_there.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

root = Tk()
app = Application(master=root)
```

```
app.mainloop()
root.destroy()
```

Capítulo 21

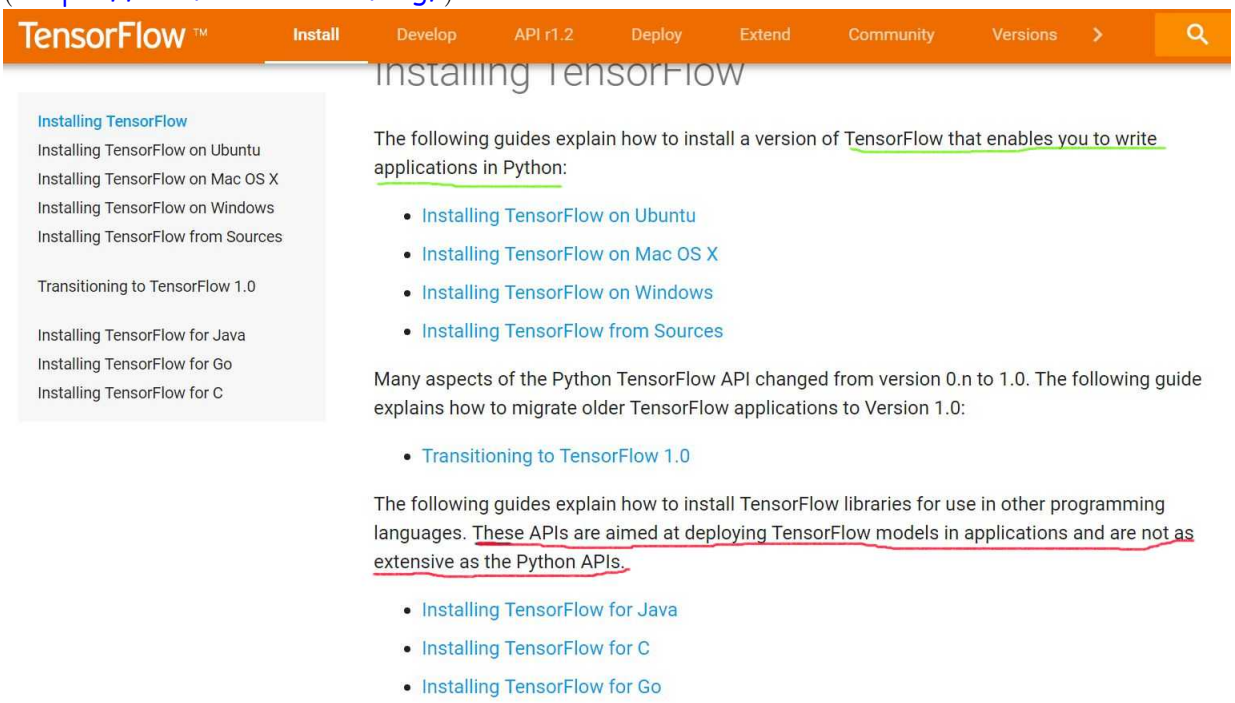
Pacote: TensorFlow

É um produto freeware (licença Apache) desenvolvido pela Google para aprendizado de máquina. Substituiu seu antecessor DistBelief que era proprietário. Além de ser um belo projeto de pesquisa ele é usado em produção em muitos produtos google (tradução, classificação de fotos, busca, reconhecimento da fala humana, etc). Sua primeira versão foi liberada em 9/11/15 e seu desenvolvimento começou em 2011.

Está em `www.tensorflow.org`. Embora tenha APIs para uso em C++, Java, Go seu uso mais completo e documentado se dá em Python, já que a ferramenta está desenvolvida neste ambiente. Uma ótima oportunidade para apreender e começar a usar esta linguagem.

Pode-se visualizar o vídeo em `https://www.youtube.com/watch?v=oZikw5k_2FM`. Se quiser, peça a tradução simultânea para o português. Verá o TS em ação.

Eis aqui uma demonstração do vigor do Python neste ambiente. É uma das telas do TensorFlow, lá no site dele (`https://www.tensorflow.org/`)



Vale lembrar a existência de um ambiente Python que pode ser executado avulso (a partir de um pendrive, por exemplo), o que é ótimo quando você não tem os direitos de administrador na máquina que está usando. Trata-se do chamado Python portátil e ele pode ser buscado em `https://winpython.github.io/`. Este sujeito já traz o NUMPY instalado.

Tensor Flow

Este produto só roda em máquinas 64 bits (lembrem-se, ele não economiza recursos) e estando nesse ambiente, a instalação é simples, basta fazer `pip install tensorflow`.

Depois de instalar, para verificar se deu tudo certo faça:

```
>>> import tensorflow as tf
>>> hello = tf.constant('Oi, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
```

Deve aparecer a mensagem `Oi, TensorFlow!` indicando que a instalação foi bem sucedida.

Tensor

O ponto central em TF é o tensor. Trata-se de um array de valores primitivos configurados em uma matriz (Python). O ranking de um tensor é o número de dimensões da matriz que o contém. Eis alguns exemplos:

```
3 # tensor de rank 0; escalar com forma []
[1. ,2., 3.] # tensor de rank 1;vetor forma [3]
[[1., 2., 3.], [4., 5., 6.]] # tensor rank 2;
#                      matriz com forma [2, 3]
[[[1., 2., 3.]], [[7., 8., 9.]]]
# tensor rank 3; tensor com forma [2, 1, 3]
```

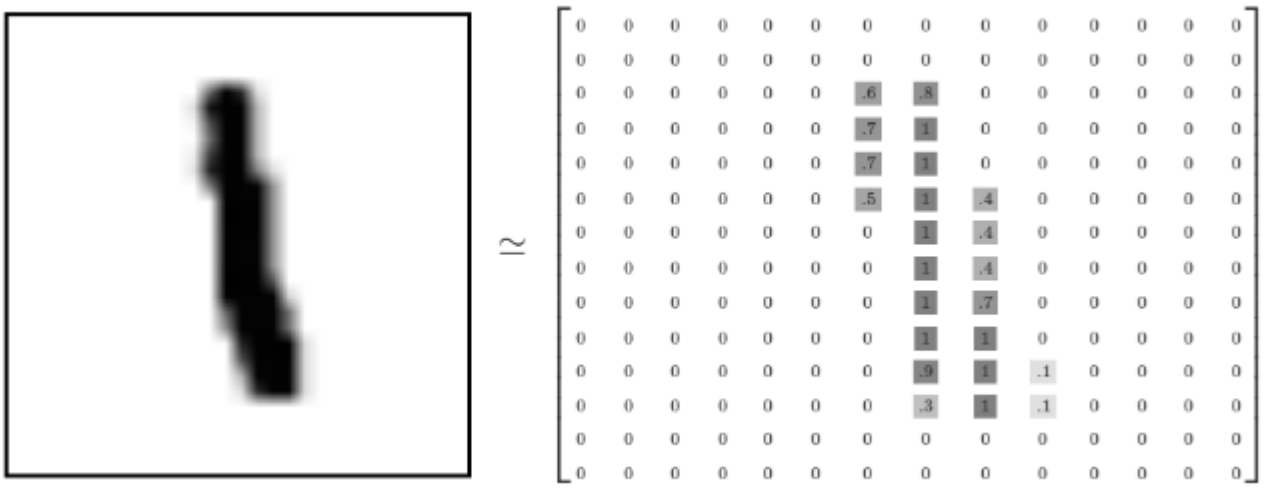
Rede

Uma rede neural é uma série de operações TensorFlow dispostas em um grafo de nodos. Cada nodo tem zero ou mais tensores de entrada e produz um tensor de saída.

Um exemplo completo está a seguir. Ele é um interpretador de dígitos escritos à mão, a partir dos dados do NIST.

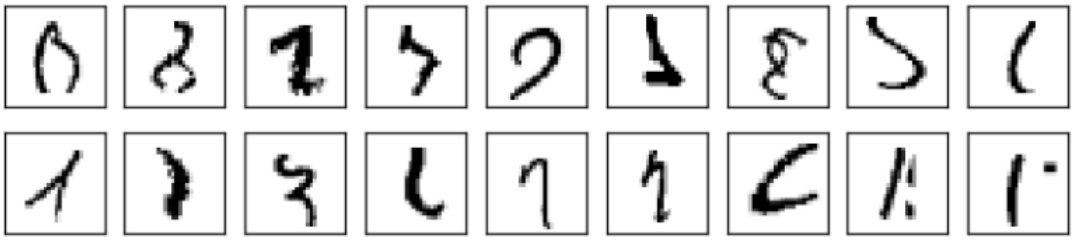
É um conjunto de 70.000 imagens de dígitos manuscritos capturadas como imagens de 28×28 pixels ($28 \times 28 = 784$) associados a um vetor [0..9] que indica qual o número certo. Os dados estão divididos em 55.000 (treinamento) 10.000 (teste) e 5.000 validação. Lembre-se que quem aprova não pode ser o que treinou.

Veja um exemplo de um número 1



Se você rodar o programa abaixo terá uma acurácia de 92% aproximadamente. Melhorando os operadores da rede neural chega-se facilmente a 97% e os melhores modelos chegam a 99.7 %.

Eis algumas imagens difíceis do conjunto



Uma explicação completa deste exemplo está em

```
\verb+https://www.tensorflow.org/get_started/mnist/beginners+
O módulo abaixo é mnist_softmax.py...

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import sys
```

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf #importa o tensorflow

FLAGS = None

def main(_):
  # Import data
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])  # dados de entrada
  W = tf.Variable(tf.zeros([784, 10]))         # a RN
  b = tf.Variable(tf.zeros([10]))              # os bias
  y = tf.matmul(x, W) + b                      # o resultado da RN
  # Define loss and optimizer
  y_ = tf.placeholder(tf.float32, [None, 10])
        # lugar onde vem a resposta certa
  cross_entropy = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
    # quero minimizar a diferença entre y (calculado) e y_ (certo)
  train_step =
  tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
  sess = tf.InteractiveSession()   # cria a sessao
  tf.global_variables_initializer().run()
    # inicializa variáveis
  # Train
  for _ in range(1000):
   # treinamento 1000 sessoes de 100 aleatorios
    batch_xs, batch_ys = mnist.train.next_batch(100)
    # coloca aqui os dados
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
    #roda a RN
  # Test trained model
  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
                  #y é igual a y_?
  accuracy =
   tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
     #acurácia
  print(sess.run(accuracy, feed_dict={x: mnist.test.images,
                                      y_: mnist.test.labels}))
     #imprime acurácia a partir das imagens de teste...
if __name__ == '__main__':
    # descobre se isto está sendo chamado ou importado
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir',
    type=str, default='/tmp/tensorflow/mnist/input_data',
              help='Directory for storing input data')
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

Se você rodar o programa acima terá uma rede neural funcional completa e obterá a acurácia de uma execução (tipicamente 92%).

Um exercício

Sua tarefa aqui é razoavelmente simples:

1. Descubra o mínimo necessário para entender uma rede neural artificial
2. Instale (ou descubra o uso de) Python 3 em uma máquina 64 bits
3. Instale o pacote tensorflow (`pip install tensorflow`)
4. Estude os tutoriais:
    - Getting started

- ... with Tensor Flow
- MNIST for ML beginners
- Deep MNIST for Experts

Se quiser, peça para o browser traduzir para o português, o resultado é bastante razoável (feito pelo TF...)

5. Digite ou copie o programa `mnist_softmax.py`
6. Rode-o.

Você terá uma Rede Neural satisfatória e contendo o estado da arte em Machine Learning.

Capítulo 22

Pacote: Itertools

O módulo itertools implementa iteradores inspirado em APL, Haskell e SML. Cada um deles foi reconstruído para se adaptar a Python eficiente.

Interadores infinitos Precisam ser interrompidos na marra

Iterador	Argumentos	Resultados	Exemplo
count()	start, [step]	start, start+step, start+2*step,	… count(10) –> 10 11 12 13 14 ...
cycle()	p	p0, p1, … plast, p0, p1, …	cycle('ABCD') –> A B C D A B C D ...
repeat()	elem [,n]	elem, elem, elem, … endlessly or up to n times	repeat(10, 3) –> 10 10 10

A seguir, iteradores que terminam quando acaba a lista mais curta:

Iterador	Argumentos	Resultados	Exemplo
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, …	accumulate([1,2,3,4,5]) –> 1 3 6 10 15
chain()	p, q, …	p0, p1, … plast, q0, q1, …	chain('ABC', 'DEF') –> A B C D E F
chain.from_iterable()	iterable	p0, p1, … plast, q0, q1, …	chain.from_iterable(['ABC', 'DEF']) –> A B C D E F
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	compress('ABCDEF', [1,0,1,0,1,1]) –> A C E F
filterfalse()	pred, seq	elements of seq where pred(elem) is false	filterfalse(lambda x: x%2, range(10)) –> 0 2 4 6 8
groupby()	iterable[, key]	sub-iterators grouped by value of key(v)	
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	islice('ABCDEFG', 2, None) –> C D E F G
starmap()	func, seq	func(*seq[0]), func(*seq[1]), …	starmap(pow, [(2,5), (3,2), (10,3)]) –> 32 9 1000
takewhile()	pred, seq	seq[0], seq[1], until pred fails	takewhile(lambda x: x<5, [1,4,6,4,1]) –> 1 4
tee()	it, n	it1, it2, … itn splits one iterator into n	
zip_longest()	p, q, …	(p[0], q[0]), (p[1], q[1]), …	zip_longest('ABCD', 'xy', fillvalue='-') –> Ax By C- D-

geradores combinatórios:

Iterador	Argumentos	Resultados
product()	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
permutations()	p[, r]	r-length tuples, all possible orderings, no repeated elements
combinations()	p, r	r-length tuples, in sorted order, no repeated elements
combinations_with_replacement()	p, r	r-length tuples, in sorted order, with repeated elements
product('ABCD', repeat=2)		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
permutations('ABCD', 2)		AB AC AD BA BC BD CA CB CD DA DB DC
combinations('ABCD', 2)		AB AC AD BC BD CD
combinations_with_replacement('ABCD', 2)		AA AB AC AD BB BC BD CC CD DD

22.1 Funções do Pacote Itertool

As funções a seguir retornam iteradores. Algumas devolvem fluxos de tamanho infinito, portanto este deve ser interrompido em algum momento.

itertools.chain(*iterables) Retorna cada um dos elementos interáveis. Veja

```
import itertools as it
def exe1():
  for a in it.chain('ABCDEF'):
    print(a)
exe1()
```

Este caso é exatamente igual a

```
import itertools as it
def exe1():
  for a in list('ABCDEF'):
    print(a)
exe1()
  dá como resposta A, B, C, D, E e F
```

itertools.combinations(iterable, r) Devolve as combinações r a r. Devolve subsequencias de comprimento r a partir da entrada. As combinações são emitidas em ordem lexicográfica. Se a entrada já está ordenada, a saída também estará ordenada. Os elementos são tratados unicamente por sua posição, não por seu valor. Se os elementos na entrada são únicos, não haverá repetição na saída.

O número de itens retornados é

$$C(n,r) = \frac{n!}{r!(n-r)!} = \frac{A(n,r)}{r!}$$

Veja o exemplo

```
import itertools as it
def exe1():
  for a in it.combinations('ABCDEF',3):
    print(a)
exe1()

Dá como respostas
('A', 'B', 'C')
('A', 'B', 'D')
('A', 'B', 'E')
('A', 'B', 'F')
('A', 'C', 'D')
('A', 'C', 'E')
('A', 'C', 'F')
('A', 'D', 'E')
```

```
...
('D', 'E', 'F')

Ou
it.combinations(range(5),3)
dará (0, 1, 2), (0, 1, 3), (0, 1, 4), (0, 2, 3),
(0, 2, 4), (0, 3, 4), (1, 2, 3), (1, 2, 4),
(1, 3, 4) e (2, 3, 4)
```

itertools.combinations_with_replacement(iterable, r) Retorna subsequencias de comprimento r a partir da entrada, permitindo que os elementos da entrada sejam repetidos mais de uma vez.

O número de itens retornados é

$$(n + r - 1)!/r!/(n - 1)!$$

quando $n > 0$.

veja o exemplo:

```
it.combinations_with_replacement('ABC',2)
dá
('A', 'A'), ('A', 'B'), ('A', 'C'),
('B', 'B'), ('B', 'C') e ('C', 'C')
```

itertools.compress(data, selectors) Retorna apenas os iteradores que são selecionados como True. Veja:

```
it.compress('ABCDE',[1,0,1,0,1]):
Retorna A, C e E
```

itertools.count(start=0, step=1) Retorna valores espaçados em *step* começando em *start*.

itertools.cycle(iterable) Retorna os itens da entrada de maneira cíclica ao infinito. Precisa ser interrompido.

itertools.dropwhile(predicate, iterable) Enquanto o predicado for falso, ele não devolve. Após o primeiro Verdadeiro, volta tudo.

```
it.dropwhile(lambda x: x.isupper(),'AcBgChDE'):
é cBgChDE
```

itertools.groupby(iterable[, key]) Devolve duas respostas, veja:

```
import itertools as it
data='aaaaaaabbbbbbbbccccccde'
groups = []
uniquekeys = []
for k, g in it.groupby(data):
    groups.append(list(g))      # Armazena como uma lista
    uniquekeys.append(k)
print(groups)
print(uniquekeys)

Retorna
print(k)
[['a', 'a', 'a', 'a', 'a', 'a', 'a'], ['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['c', 'c', 'c

print(g)
['a', 'b', 'c', 'd', 'e']
```

filterfalse Substitui as ifilter e ifilterfalse do Python2 que não existem mais no 3.

```
import itertools
def is_lt_100(x):
  return x < 100

a = [1,2,3,4,1000,66,1201]
b = itertools.filterfalse(is_lt_100, a)
```

```
for i in b:
  print (i)

deu 1000 e 1201
```

map A rigor este cara não é do itertools, mas vamos lá

```
for a in map(pow,(2,3,5),(10,11,12)):
  print(a)
deu como respostas  1024, 177147 e 244140625
```

itertools.starmap(function, *iterables) Compare com a anterior

```
import itertools as it
for a in it.starmap(pow,[(2,10),(3,11),(5,12)]):
  print(a)
```

itertools.permutations(iterable, r=None)

```
\begin{verbatim}
import itertools as it
for a in it.permutations('ABC',2):
  print(a)

deu
('A', 'B'), ('A', 'C'), ('B', 'A'),
('B', 'C'), ('C', 'A') e ('C', 'B')
```

itertools.product(iterables, repeat=1) Realiza o produto cartesiano. O product(A,B) faz o mesmo que ((x,y) for x in A for y in B).

Para comutar o produto de um iterável com ele mesmo use a palavra repeat. Assim, product(A,repeat=4) faz o mesmo que product(A,A,A,A).

```
import itertools as it
for a in it.product('ABC','123'):
  print(a)

  deu
('A', '1'), ('A', '2'), ('A', '3'),
('B', '1'), ('B', '2'), ('B', '3'),
('C', '1'), ('C', '2') e  ('C', '3')
```

itertools.repeat(objeto[,vezes]) Repete indefinidamente um iterador. Veja em

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.takewhile(predicate, iterable) Retorna até que o predicado seja falso

```
   # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
```

Capítulo 23

UTF-8

A informática convive – desde seu início – com uma dificuldade e tanto: como padronizar a utilização de códigos binários. O objetivo é nobre: permitir a troca de objetos binários e a sua correta interpretação por parte de todos os parceiros em uma comunicação qualquer. A necessidade é nova, a informática só apareceu na face da terra na década de 50 do século passado. No início, cada fabricante usava seus códigos. A IBM, por exemplo, construiu toda a sua família de equipamentos usando o código EBCDIC. Isto evoluiu para o código ASCII (início de 1960, comite ANSI-X3.2) que acabou sendo adotado como padrão. Ele privilegiou o idioma inglês, e padronizou seus caracteres nas primeiras 128 posições do código ASCII. Aos demais idiomas do planeta ficou disponível a segunda parte do código ASCII (mais 128 posições). É aqui que o código BRASCII (NBR-9614:1986 e NBR-9611:1991 da ABNT) colocou os caracteres acentuados do português. Assim como o Brasil, muitos outros países fizeram o mesmo. Isto apresentou pelo menos 2 problemas:

- Um arquivo composto no Brasil, quando impresso numa impressora fabricada na França, aparecia como uma série de caracteres malucos.
- Idiomas com muitos caracteres (katakana, hiragana, hangul, tagalog,...) simplesmente não tinham como ser representados no código ASCII. Não havia espaço.

23.1 Unicode

Em meados da década de 80, a academia e a indústria começaram a discutir propostas de resolver os 2 problemas acima. Diversos padrões foram sugeridos e eles começaram a ser usados, na base do "cada um por si". Uma primeira proposta sugeria usar 16 bits (2 bytes) o que permitiria $2^{16} = 65536$ caracteres. Parecia muito, mas vale citar Peter Norton, que disse há mais de 30 anos: *na informática, não importa quanto você tem. Não é o suficiente.*

Em 1991, criou-se o Consórcio Unicode e em outubro de 1991 o primeiro volume do padrão foi publicado. Em 1996, introduziu-se um mecanismo de codificação e com ele, a barreira dos 2 bytes foi ultrapassada. Agora, o espaço de endereçamento do Unicode ultrapassou o milhão de caracteres, o que permitiu incluir idiomas históricos (hieróglifos egípcios) ou mesmo idiomas muito pouco usados (Linear B por exemplo), além de algumas brincadeiras como o idioma Klingom. O padrão atual consagra 1.114.112 códigos de ponto (*codepoints*), correspondendo ao intervalo hexadecimal de 00.00.00 a 10.FF.FF. Note que nem todo codepoint indica um caracter, já que alguns deles são reservados a outras finalidades.

Há alguns codepoints privados, que não têm significação no Unicode. Eles ficam para uma negociação entre remetente e destinatário e podem ser usados para qualquer coisa (são eles: área privada: U+E000..U+F8FF com 6,400 caracteres; área suplementar A: U+F0000..U+FFFFD com 65,534 caracteres e área suplementar B: U+100000..U+10FFFD com 65,534 caracteres).

O Consórcio Unicode trabalha junto com a ISO (International Organization for Standardization) já que o padrão ISO/IEC 10646 é o próprio código Unicode. Um conceito importante é o de script. Trata-se de um conjunto de letras, sinais e regras de escrita que agregam partes do Unicode e que são usados em conjunto de *writing systems*. Na versão 7 do Unicode, são 123 os scripts, e há uma lista de candidatos a serem incluídos.

23.2 UTF=Unicode Transformation Formats

Diversos mecanismos foram sugeridos para implementar o código Unicode, com a preocupação sempre presente de compatibilidade reversa, isto é: um novo esquema de codificação deveria preservar – tanto quanto possível – o legado infindável de dados já coletados e guardados. Foram propostos os UTF-1 (nunca chegou a ser muito usado, pois tinha problemas de performance), UTF-16 (usava 1 ou 2 palavras de 16 bits) , UTF-32 (cada caracter usava exatos 32 bits), mas todas elas acabaram migrando para a UTF-8 que se firmou como padrão de fato, principalmente porque muito cedo foi adotado como padrão pelo consórcio IMC (Internet Mail Consortium) e pelo W3C. O Google reporta que desde 2008, a maioria dos conteúdos HTML trocados na web usa a codificação UTF-8. O esquema todo foi escrito durante um jantar em 2 de setembro de 1992 por Ken Thompson e Rob Pike. Suas grandes qualidades são:

- Compatibilidade reversa, já que caracteres da primeira parte do ASCII não sofrem modificação (1^o bit=0). Desta maneira qualquer arquivo ASCII que não use a segunda parte já é um arquivo UTF-8.
- Clara distinção entre caracteres de byte único (1^o bit=0) e caracteres multi-bytes. Estes contém um primeiro byte heading e um ou mais bytes de continuação. O heading tem 2 ou mais bits 1 seguidos por um zero enquanto os bytes de continuação sempre começam por 10.
- Auto-sincronização. Bytes únicos, header e continuação não compartilham valores, indicando que o início do caracteres está a no máximo 3 bytes para trás.
- Indicação do comprimento: O número de uns na alta ordem do byte heading indica a quantidade de bytes do caractere, sem ser necessário examinar os seus bytes.

Veja-se um resumo do exposto

bits	inicio	fim	tamanho	byte1	byte2	byte3	byte4
7	U+0000	U+007F	1	0xxxxxxx			
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx		
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx	
21	U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Veja-se agora alguns exemplos de caracteres unicode de 1, 2, 3 e 4 bytes

Character		Binary code point
$	U+0024	0100100
¢	U+00A2	000 10100010
€	U+20AC	00100000 10101100
𤭢	U+24B62	00010 01001011 01100010

Binary UTF-8	Hexadecimal UTF-8
00100100	24
11000010 10100010	C2 A2
11100010 10000010 10101100	E2 82 AC
11110000 10100100 10101101 10100010	F0 A4 AD A2

Uma tabela de conversão binário hexadecimal para ajudar:

hex	bin	hex	bin	hex	bin	hex	bin
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

23.3 Exemplos

1. Seja o seguinte conjunto de 5 caracteres UNICODE codificados em UTF-8.

```
F2B1B88DF4BAA8A7E192A9EF92A5F4A584BF
```

Seus codepoints em hexadecimal

```
0B1E0D 13AA27 14A9 F4A5 12513F
```

E em decimal

```
728589 1288743 5289 62629 1200447
```

2. Seja o seguinte conjunto de 5 caracteres UNICODE codificados em UTF-8.

```
39F6B2B6A503E68FAFF59A84B1
```

Seus codepoints em hexadecimal

```
39 1B2DA5 03 63EF 15A131
```

E em decimal

```
57 1781157 3 25583 1417521
```

23.4 Exercício

A seguir um conjunto de 5 caracteres UNICODE codificados em UTF-8. Você deve examiná-los e descobrir quais os 5 codepoints primeiro escritos em hexadecimal e depois em decimal (é só uma simples conversão).

```
E89A8DC4A9F69C82A1F4ABBABBCA8F
```

23.4.1 Resposta

```
1 : 868D 0129 19C0A1 12BEBB 028F          34445 297 1687713 1228475 655
```


Capítulo 24

Um estudo de otimização

24.1 Otimização de algoritmos em Python

Seja o problema 12 do magnífico site de matemática e programação projecteuler.net. Ele diz: A sequência dos números triangulares é gerada pela adição dos números naturais. Assim, o 7^o número triangular será 1+2+3+4+5+6+7 = 28. Os primeiros 10 números triangulares são

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...$$

Vai-se listar aqui os divisores dos primeiros sete números triangulares:

1: 1

3: 1,3

6: 1,2,3,6

10: 1,2,5,10

15: 1,3,5,15

21: 1,3,7,21

28: 1,2,4,7,14,28

Pode-se ver que 28 é o primeiro número triangular que tem mais do que 5 divisores. Qual é o valor do primeiro número triangular que tem mais do que 500 divisores ?

24.1.1 Primeira estratégia

Sem pensar muito, pode-se resolver a questão usando a força bruta. Aqui, geram-se os números triangulares como descrito e depois acham-se todos seus divisores. A busca acaba quando se acha o primeiro com mais do que 500 divisores. (Uma informação: inteiros de 32 bits são suficientes para este problema). Uma solução em python:

```
t=1
a=1
cnt=0
while cnt <= 500:
   cnt=0
   a=a+1
   t=t+a
   i=1
   while i<=t:
      if t%i == 0:
         cnt=cnt+1
      i=i+1
print(t)
```

A implementação acima é muito demorada (estimei em umas 5h em uma CPU rápida)

24.1.2 Segunda estratégia

Uma melhora evidente é interromper a busca quando o divisor alcança a raiz quadrada do número triangular: Para cada divisor abaixo da raiz quadrada existe um acima dela. O código fica:

```
t=1
a=1
cnt=0
while cnt <= 500:
   cnt=0
   a=a+1
   t=t+a
   i=1
   ttx = sqrt(t)
   while i<=ttx:
      if t%i == 0:
         cnt=cnt+2
   if t=ttx*ttx:
      cnt=cnt-1 # correção quadrado perfeito
print(t)
```

Ainda assim, esta segunda estratégia continua muito demorada.

24.1.3 Terceira estratégia

Vamos nos socorrer de um pouco de matemática. Sabe-se que qualquer inteiro N pode ser expresso de maneira única como

$$N = p_1^{a1} \times p_2^{a2} \times p_3^{a3} \times ...$$

onde p_i é um número primo e a_n seu expoente. Por exemplo, $28 = 2^2 \times 7^1$. Daqui, o número de divisores de N, $D(N)$ de qualquer inteiro pode ser computado como

$$D(N) = (a_1 + 1) \times (a_2 + 1) \times ...$$

onde os a_n são os expoentes dos fatores primos que compõe o número N. No exemplo, os divisores de 28 são $D(28) = (2 + 1) \times (1 + 1) = 3 \times 2 = 6$. Uma tabela de primos é necessária para aplicar esta relação. Uma estratégia para criá-la pode ser (até 65500):

```
def primo(n):
# devolve True se n primo e False se n composto
  import math
  if n==1:
    return False
  if n<4:
    return True
  if n%2==0:
    return False
  if n<9:
    return True
  if n%3==0:
    return False
  r=math.floor(math.sqrt(n))
  f=5
  while f<=r:
    if n%f==0:
      return False
    if (n%(f+2))==0:
      return False
    f=f+6
  return True
def criatabprimos():
    pri=[2]
    i=1
    j=3
    while j<65500:
        if primo(j):
```

```
            pri.append(j)
            i=i+1
        j=j+2
    f=open("f:/p/.../primos.lis","w")
    f.write(str(a)[1:-1])
    f.close()
```

Depois, para ler essa tabela, usar:

```
def leiapri():
    f=open("f:/p/.../primos.lis","r")
    a=f.read()
    pri=list(map(int,a.split(", ")))
    print("feito")
```

Agora, o processamento

```
t=1
a=1
cnt=0
pri # lista de primos (acima...)
while cnt <= 500:
   cnt=1
   a=a+1
   t=t+a
   tt=t
   i=0
   while i < len(pri):
      if pri[i]*pri[i]>tt:
         cnt=cnt*2
         break
      expoente = 1
      while tt%pri[i]==0:
         expoente=expoente+1
         tt=tt//pri[i]
      if expoente>1:
         cnt=cnt*expoente
      if tt==1:
         break
      i=i+1
print(t)
```

Quarta estratégia

Ainda se pode melhorar mais, sabendo que os números triangulares também podem ser obtidos de acordo com

$$t = \frac{n \times (n+1)}{2}$$

onde os componentes n e $n+1$ são necessariamente co-primos (isto é, não têm qualquer fator primo comum nem divisor comum). Aqui, o número de divisores $D(t)$ pode ser obtido

$$D(t) = D(n/2) \times D(n+1) \; se \; n \; par$$

ou

$$D(t) = D(n) \times D((n+1)/2) \; se \; n+1 \; par$$

Cada componente é muito menor do que o número triangular. A tabela de primos também é muito menor (contém apenas primos menores do que 1000) Mais, o resultado do componente $n+1$ pode ser reaproveitado como n no próximo número triangular, sem ser necessário recalculá-lo. Fica:

```
n=3
Dn=2
cnt=0
f=open("f:/p/.../primos.lis","r")
a=f.read()
```



```
prix=list(map(int,a.split(", ")))
pri=prix[0:168] # lista primos até 1000
while cnt <= 500:
  n = n+1
  n1 = n
  if n1 % 2 == 0:
    n1 = n1//2
  Dn1 = 1
  i=0
  while i<len(pri):
    if pri[i]*pri[i] > n1:
      Dn1=2*Dn1
      break
    exponent=1
    while n1 % pri[i] == 0:
       exponent=exponent+1
       n1 = n1//pri[i]
    if exponent > 1:
       Dn1=Dn1*exponent
    if n1 == 1:
       break
    i=i+1
  cnt = Dn*Dn1
  Dn = Dn1
print (int((n*(n-1)/2)))
```

Esta implementação demorou menos de $\frac{1}{10}$ de segundo na mesma CPU lá do começo. Veja-se como vale a pena estudar algoritmos.

24.1.4 Exercício

O objetivo deste exercício é treinar a implementação de algoritmos em python e também meditar sobre como modificações na implementação de um mesmo problema a partir de conhecimentos matemáticos (Santa Matemática !) podem reduzir radicalmente os tempos de processamento.

Para ajudar nesta meditação, implemente a alternativa que quiser e informe qual o número inteiro triangular que tem mais do que

```
168
```

divisores.

Resposta

1385280

Capítulo 25

Pacote: Django

É um framework para construir aplicações WEB. Foi criado originalmente como sistema para gerenciar um site jornalístico na cidade de Lawrence, no Kansas. Virou depois um projeto de código aberto.

25.1 Passo a Passo: tutorial 1 da documentação

25.1.1 Obter e instalar os pacotes

Instale o pacote Django

25.1.2 Path

Você precisa fazer o python acessível de qualquer lugar. Assim, sua variável path deve refletir este fato. Para isso:

1. Painel de controle
2. Aba system
3. Aba advanded system settings
4. Procure a variável PATH e acrescente a ela `;c:\python3` ou o nome onde o python tiver sido instalado
5. Salve e feche tudo

25.1.3 Criar o diretório do projeto Django

Vá em algum lugar (diretório) e crie uma pasta para o projeto. Aqui será

```
cd\
mkdir lixopro
cd lixopro
python c:\python3\scripts\django-admin.py startproject mysite
```

O resultado disso é a criação de um diretório abaixo de lixopro chamado mysite onde o django vai guardar coisas. Eis o que foi criado

```
mysite/
   manage.py
   mysite/
      __init__.py
  settings.py
  urls.py
  wsgi.py
```

cada um desses arquivos é:

- O mysite externo é um container para o seu projeto. O nome dele não interessa.
- `manage.py` é um utilitário de linha de comando que permite a você interagir com o projeto django. Mais em `https://docs.djangoproject.com/en/2.0/ref/django-admin/`
- o mysite interno é o diretório do projeto. Você usará este nome para importar coisas dentro dele (por exemplo `mysite.urls`).

- `mysite/__init__.py` é um arquivo vazio que diz ao python que este diretório deve ser um pacote python.
- `mysite/setting.py` configuração para o projeto django.
- `mysite/urls.py` As declarações de url. É uma tabela de conteúdo.
- `mysite/wsgi.py` Um entry=point para web-servers compatíveis com WSGI.

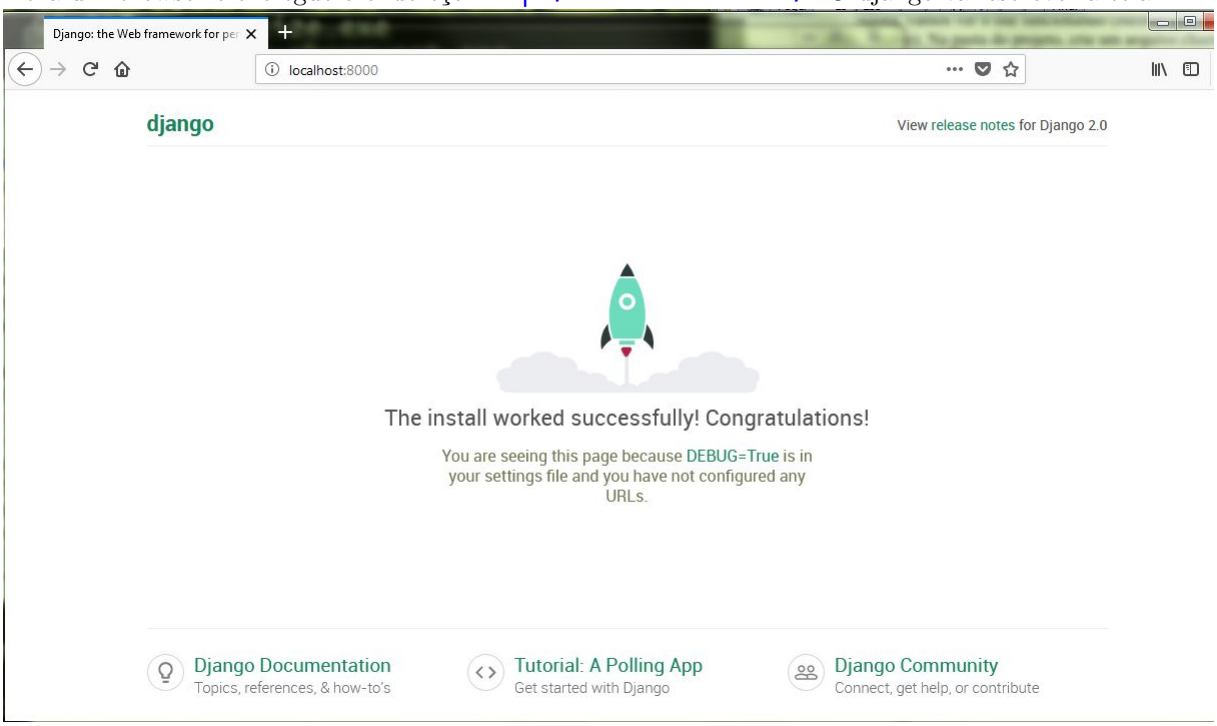
25.1.4 Começando

Vá nesse diretório e execute

```
> python manage.py runserver
```

25.1.5 Testar com o browser

Abra um browser e entregue o endereço `http:/localhost:8000/`. O django vai escrever a tela



25.1.6 A qualquer momento

Em qualquer janela dos, faça

```
> python -m django --version
```

e deve obter algo como `2.0.7` indicando a versão do django que está instalada.

25.1.7 Criando sua primeira aplicação

Trata-se de escrever uma APP. A propósito, qual a diferença entre projeto e APP ? Uma APP é uma aplicação web que faz alguma coisa. Um projeto é uma coleção de configurações e APPs para um website particular. Um projeto pode conter múltiplas APPs e uma APP pod ser usada em múltiplos projetos.

Para criar uma app vá para o diretório onde está `manage.py` e digite o comando

```
>python manage.py startapp polls
```

Isto vai criar o diretório polls que deverá ter

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

25.1.8 Escrevendo a primeira coisa

Abra o arquivo `polls/views.py` e coloque algum código python nele

```
from django.http import HttpResponse
def index(request):
   return HttpResponse("Ola mundo cruel")
```

Agora precisamos configurar uma URL Dentro do diretório polls crie um arquivo chamado `urls.py`. Agora seu diretório deve estar assim

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    urls.py          ---> recém criado
    views.py
```

O arquivo `polls/urls.py` deve incluir o seguinte código

```
from django.urls import path
from . import views
urlpatterns=[
   path('', views.index, name='index'),
]
```

O próximo passo é atualizar a url no módulo `mysite/urls.py` Adicione um import para `django.urls.include` e um insert na lista de `urlpatterns`. O arquivo `mysite\urls.py` deve ficar assim

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

25.1.9 Para testar

Entrar no diretório `\lixopro\mysite`

```
> python manage.py runserver
```

no browser `//localhost:8000/polls` e deve aparecer a mensagem

```
alo mundo cruel
```

Isto encerra o equivalente ao tutorial 1 da documentação do Django

25.2 Passo a passo: tutorial 2

25.2.1 Estabelecendo o banco de dados

Deve-se manipular `mysite/settings.py`. É um módulo normal python com variáveis controle do Django. Por default, a configuração usa SQLite. Este já está instalado no Python, assim, você não precisa instalar mais nada.

Se você quiser usar outro banco de dados (mysql ?), instale ele e depois altere os seguintes parâmetros no item DATABASES 'default' de acordo com o banco. Este cara é um dicionário contendo as coisas para interligar com um banco de dados. Eis a definição para SQLite:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'mydatabase',
    }
}
```

Eis um exemplo para postgres

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

A primeira coisa para criar um banco de dados é alterar o módulo

```
/pro/mysite/polls/models.py   (módulo models.py em polls)

from django.db import models

class Question(models.Model):
question_text = models.CharField(max_length=200)
pub_date = models.DateTimeField('date published')

class Choice(models.Model):
question = models.ForeignKey(Question, on_delete=models.CASCADE)
choice_text = models.CharField(max_length=200)
votes = models.IntegerField(default=0)
```

Deve-se notar que cada entidade deve ser uma classe. Os tipos de campo, (entre outros) podem ser: AutoField, BinaryField, BooleanField, CharField, DateField, DateTimeField, DecimalField, DurationField, EmailField, FileField, FloatField, ImageField, IntegerField, URLField

Os relacionamentos são indicados usando ForeignKey, ManyToManuField, OneToOneField.

Depois de montar seu banco de dados, é necessário dizer ao Django que a aplicação (polls no caso) está instalada. Então em

```
/pro/mysite/mysite/settings.py

INSTALLED_APPS = [
   'polls.apps.PollsConfig',
   ...
   ]
```

Agora deve-se rodar uma execução de manage para informar ao Django que foram feitas modificações no banco de dados e ele deve obedecê-las.

```
/pro/mysite

> python manage.py makemigrations polls
```

Se quiser ver o que este cara fez, execute `python manage.py sqlmigrate polls 0001` Ele vai listar o que irá fazer.

Para fazer a implementação real no banco:

```
/pro/mysite

> python manage.py migrate
```

Em resumo, para manutenção no banco de dados, são 3 coisas:

1. Mude seus modelos em `/pro/mysite/polls/models.py`
2. Rode em `/pro/mysite` python manage.py makemigrations
3. No mesmo local, rode python manage.py migrate

Para olhar o ambiente e brincar com ele, deve fazer

```
/pro/mysite

> python manage.py shell
```

Este cara vai abrir um Python interativo que vai permitir que você brinque com o ambiente. lembrando que a primeira coisa que precisa fazer é importar as classes (entidades) que você definiu no seu modelo. Então:

```
>>> from polls.models import Choice, Question
```

Para sair deste shell, faça `quit()`.

Agora, vai-se lidar com a administração do site. Para criar um superusuário, faça

```
/pro/mysite

> python manage.py createsuperuser
```

Agora ele vai pedir o nome, email e senha (2 vezes) do sujeito.

Para iniciar o servidor de desenvolvimento faça

```
/pro/mysite

> python manage.py runserver
```

Depois, você deve abrir um browser e escrever `127.0.0.1:8000/admin/` Deve aparecer uma tela de administração.

Para incluir o novo sistema desenvolvido na tela de administração:

```
/pro/mysite/polls/admin.py

from django.contrib import admin
from .models import Question
admin.site.register(Question)
```

25.3 Passo a Passo: Tutorial 3

Agora vão ser criadas as interfaces do sistema, as views. Uma view é uma página web que faz algo e tem um template específico.

Para escrever views,

```
em pro/mysite/polls/views.py

def detail(request, question_id):
   return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id):
   response = "You're looking at the results of question %s."
   return HttpResponse(response % question_id)

def vote(request, question_id):
   return HttpResponse("You're voting on question %s." % question_id)
```

Agora vá no browser e peça `/polls/1/results/`

Note que este tutorial vai ensinando a criar views e templates, mas não me interessa muito. Vou direto para

25.4 Passo a passo: Tutorial 4

Criando um formulário simples

```
polls/templates/polls/detail.html

<h1>{{ question.question_text }}</h1>
{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
<input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
<label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
```

```
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

Agora uma view que manipula o voto

25.5 Resumão

1. Baixar Python e Django. Anotar os diretórios onde estão. Criar um path que acesse o python.
2. Criar um diretório (md pro) entrar nele e executar `python c:\python\scripts\django-admin startproject mys` Este cara vai criar uma árvore de diretórios.
3. Para pôr o servidor em ação, vá em cd/pro/mysite e execute `python manage.py runserver`. A saída é com `quit()`.
4. Para criar um aplicativo em /pro/mysite `python manage.py startapp nome`
5. Altere a view em /pro/mysite/nome/views.py incluindo

   ```
   from django.http import HttpResponse
   def index(request):
     return HttpResponse("Alo Mundo")
   ```

6. Em /pro/mysite/nome crie `urls.py` e nele inclua

   ```
   from django.urls import path
   from . import views
   urlpatterns = [
      path('', views.index, name='index'),
   ]
   ```

7. Agora, em mysite/urls.py, acrescente

   ```
   from django.contrib import admin
   from django.urls import include, path
   urlpatterns = [
      path('nome/', include('nome.urls')),
      path('admin/',admin.site.urls),
   ]
   ```

8. Se tudo funcionou, ao chamar o servidor (runserver), depois um browser com `127.0.0.1:8000/nome/` deverá aparecer o alo mundo.
9. Para configurar o banco de dados, atualize `mysite/settings.py`. Se for usar SQLite, nada a fazer.
10. Em nome/models.py defina seu banco de dados, a saber

    ```
    from django.db import models
    class ENTIDADE(models.Model):
       campo = models.tipo(opções)
       ... = ...
    ```

    Principais tipos: CharField, dateField, DecimalField, FloatField, IntegerField... Relacionamentos: ForeignKey, ManyToManyField e OndeToOneField.

11. Em mysite/settings.py, altere INSTALLED_APPS, como em

    ```
    INSTALLED_APPS = [
       'nome.apps.NomeConfig',
       'django.contrib.admin',
       ...
    ]
    ```

12. Em /pro/mysite, execute `python manage.py makemigrations nome`.

13. No mesmo lugar, faça `python manage.py migrate`

14. Para testar e brincar com seus arquivos, em /pro/mysite execute `python manage.py shell`. Você abrirá uma sessão python interativa com acesso a seus arquivos.

15. Para incluir registros:

```
x = Entidade(campo = '....', campo=234...)
x.save()
```

16. Em /nome/models.py, atualize as classes de cada entidade, incluindo nelas

```
def __str__(self):
   return self.campo descritivo
```

17. Criando um usuário administrador. Vá em /pro/meusite e execute `python manage.py createsuperuser`. Depois crie um usuário admin, dê um email e uma senha. A seguir chame o servidor via runserver, e via browser vá para `127.0.0.1:8000/admin/` e entre no usuário admin.

18. Torne a aplicação visível ao admin: Vá em pro/mysite/nome e crie um arquivo chamado `admin.py` e nele escreva

```
from django.contrib import admin
from .models import Entidade
admin.site.register(Entidade)
```

    Agora, reexecutando o passo anterior, você vai ser sua entidade no admin

19. Em /pro/mysite/nome/urls.py faça a modificação

```
from django.urls import path
from . import views
app_name = 'nome'
urlpatterns = [
   path('', views.IndexView.as_view(), name='index'),
   path('<int:pk>/', views.DetailView.as_view(), name='detail'),
   ...
]
```

20. Em /pro/mysite/nome/views.py, altere

```
from django.http import HttpResponseDirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse
from django.views import generic
from .models import Entidade, Entidade
class IndexView(generic.ListView):
   template_name = 'nome/index.html'
   context_object_name = ...
   def get_queryset(self):
      return ...
class DetailView(generic.DetailView):
   model=Entidade
   template_name = 'nome/detail.html'
...
```

21. Crie formulários. Em pro/mysite/nome/templates/nome crie um `detail.html` contendo

```
<h1>{{ question.question_text }}</h1>
{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
<input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}"
<label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```


22. Crie arquivos estáticos. Em /pro/mysite/nome/static/nome/ crie o arquivo **style.css** e nele inclua o estilo desejado para o site.

    E daí, em /pro/mysite/nome/template/nome/index.html faça a referência ao estilo criado escrevendo

```
{% load static %}
<link rel="stylesheet" type ="text/css" href="{% static 'nome/style.css' %}" />
{% if latest_question_list %}
   <ul>
   ...
```

23. Se quiser criar imagens... Em /pro/mysite/nome/static/nome/images/ crie uma imagem chamada background.gif

    E depois, no estilo acima, faça

```
li a {
color: green;
}
body {
background: white url("images/background.gif") no-repeat;
}
```

Capítulo 26

Bibliografia

26.1 Python

para saber mais sobre Python

BEA13 BEAZLEY, David e JONES, Brian K. **Python Cookbook**. São Paulo, Novatec, 2013. 713p.

RAM15 RAMALHO, Luciano. **Python Fluente**. São Paulo, Novatec, 2015. 797p.

MEN10 MENEZES, Nilo Ney Coutinho. **Introdução à Programação com Python**. São Paulo, Novatec, 2010. 327p.

MCK18 MCKINNEY, Wes. **Python para Análise de Dados**. São Paulo, Novatec, 2018. 615p.

MUE16 MUELLER, John Paul, **Começando a programar em Python para leigos** Rio de Janeiro, Alta Books, 2016. 379p.

KIN15 KINSLEY, Harrison e McGUGAN, Will. **Introdução ao desenvolvimento de Jogos em Python com PyGame**. São Paulo, Novatec, 2015.

PYTxx Comunidade Python Mundial. Site https://www.python.org.

PYTxx Comunidade Python Brasil. Site https://python.org.br.

KAN18 KANTEK, Pedro Luis Garcia Navarro. **Python**. Curitiba, UFPR, 2018. Disponível em www.pkantek.com.br/atividades.

DOW16 DOWNEY, Allen B. **Pense em Python**. São Paulo, Novatec, 2016.

26.2 Algoritmos e programação

Livros e materiais sobre programação.

DEL12 DELGADO, Armando Luiz. **Linguagem C++**. Curitiba, UFPR, 2012.

DEW89 DEWDNEY, A. K. **The (new) Turing Omnibus**. New York, W. H. Freeman, 1989. 460p.

COR14 CORMEN, Thomas H. **Algoritmos**. Rio de Janeiro, Elsevier, 2014. 188p.

BHA17 BHARGAVA, Aditya. **Entendendo Algoritmos**. São Paulo, Novatec, 2017. 259p.

SKI97 SKIENA, Steven S. - **The Algorithm Design Manual**. New York, Springer. 1997. 486p.

SKI03 SKIENA, Steven S e REVILLA, Miguel A. **Programming Challenges**. New York, Springer. 2003. 361p.

SKI12 SKIENA, Steven S e REVILLA, Miguel A. **Desafios de Programación**. New York, Springer. 2012. 343p. Tradução do inglês ao espanhol.

FOR05 FORBELLONE, A. L. V. e EBERSPÄCHER, H. F. - **Lógica de Programação**. São Paulo. 2005. 214p.

FOR72 FORSYTHE, Alexandra I. et all. **Ciência de Computadores** - 1. curso. Volume 1. Rio de Janeiro, Ao Livro Técnico. 1972. 234p.

FOR72b FORSYTHE, Alexandra I. et all. **Ciência de Computadores** - 1. curso. Volume 2. Rio de Janeiro, Ao Livro Técnico. 1972. 560p.

KNU71 KNUTH, Donald E. **The Art of Computer Programming**. Volumes 1 e 2. Reading, Massachussets, Addison Wesley Publishing Company, 1971, 624p.

WIR89 WIRTH, Niklaus. **Algoritmos e estruturas de dados**. Rio de Janeiro, Prentice-Hall do Brasil, 1989, 245p.

KAO08 KAO, Ming-Yang. (ed.) **Encyclopedia of Algorithms**. New York, Springer, 2008. 1200p.

26.3 Métodos Numéricos

SAN07 SANCHES, Ionildo José e FURLAN, Diogenes. **Métodos Numéricos**. Curitiba, UFPR, 2007.

FRA07 FRANCO, Neide Bertoldi. **Cálculo Numérico**. São Paulo, Pearson, 2007.

JUS18 JUSTO, Dagoberto Adriano et alli. **Cálculo Numérico, versão Python**. Porto Alegre, UFRGS, 2018.

BAR87 BARROSO, Leônidas Conceição et alli. **Cálculo Numérico (com aplicações)**. São Paulo, Harbra, 1987.

RUG88 RUGIERO, Márcia e LOPES, Vera Lúcia. **Cálculo Numérico - aspectos teóricos e computacionais**. São Paulo, Pearson, 1988.

PAC78 PACITTI, Tércio e ATKINSON, Cyril. **Programação e Métodos Computacionais**. vol 1 e 2. Rio de Janeiro, LTC, 1979.

CAM14 CAMPOS Filho, Frederico Ferreira. **Algoritmos Numéricos**. Rio de Janeiro, LTC, 2014.

Índice Remissivo

Python 3 Cheat Sheet

Latest version on :
https://perso.limsi.fr/pointal/python:memento

Base Types

integer, float, boolean, string, bytes

```
int 783  0 -192  0b010 0o642 0xF3
          null   binary octal hexa
float 9.23 0.0  -1.7e-6
                        ×10^-6
bool True False
str "One\nTwo"        Multiline string:
     escaped new line  """X\tY\tZ
    'I\'m'             1\t2\t3"""
     escaped '          escaped tab
bytes b"toto\xfe\775"
       hexadecimal octal
```

immutables

Container Types

- **ordered sequences**, fast index access, repeatable values

```
list [1,5,9]   ["x",11,8.9]   ["mot"]   []
tuple (1,5,9)  11,"y",7.4     ("mot",)  ()
```

Non modifiable values (immutables) — *expression with just comas →* `tuple`

```
str bytes  (ordered sequences of chars / bytes)   ""  b""
```

- **key containers**, no *a priori* order, fast key acces, each key is unique

```
dictionary dict {"key":"value"}   dict(a=3,b=4,k="v")   {}
(key/value associations) {1:"one",3:"three",2:"two",3.14:"π"}
collection set {"key1","key2"}    {1,9,3,0}            set()
```

keys=hashable values (base types, immutables…) `frozenset` *immutable set* *empty*

Identifiers

for variables, functions, modules, classes… names

`a…zA…Z_` followed by `a…zA…Z_0…9`

- diacritics allowed but should be avoided
- language keywords forbidden
- lower/UPPER case discrimination

☺ `a toto x7 y_max BigOne`
☹ ~~`8y and for`~~

Variables assignment

`=`

assignment ⇔ **binding** of a *name* with a *value*

1) *evaluation of right side expression value*
2) *assignment in order with left side names*

```
x=1.2+8+sin(y)
a=b=c=0          assignment to same value
y,z,r=9.2,-7.6,0 multiple assignments
a,b=b,a          values swap
a,*b=seq         unpacking of sequence in
*a,b=seq         item and list
x+=3   increment ⇔ x=x+3   and *= /= %= …
x-=2   decrement ⇔ x=x-2
x=None « undefined » constant value
del x  remove name x
```

Conversions

`type(`*expression*`)`

```
int("15") → 15
int("3f",16) → 63        can specify integer number base in 2nd parameter
int(15.56) → 15          truncate decimal part
float("-11.24e8") → -1124000000.0
round(15.56,1)→15.6      rounding to 1 decimal (0 decimal → integer number)
bool(x)  False for null x, empty container x , None or False x ; True for other x
str(x)→"…"               representation string of x for display (cf. formating on the back)
chr(64)→'@'  ord('@')→64   code ↔ char
repr(x)→"…"              literal representation string of x
bytes([72,9,64]) → b'H\t@'
list("abc") → ['a','b','c']
dict([(3,"three"),(1,"one")]) → {1:'one',3:'three'}
set(["one","two"]) → {'one','two'}
```

separator `str` and *sequence of* `str` → *assembled* `str`

```
':'.join(['toto','12','pswd']) → 'toto:12:pswd'
```

`str` *splitted on whitespaces* → `list` of `str`

```
"words with  spaces".split() → ['words','with','spaces']
```

`str` *splitted on separator* `str` → `list` *of* `str`

```
"1,4,8,2".split(",") → ['1','4','8','2']
```

sequence of one type → `list` of another type *(via comprehension list)*

```
[int(x) for x in ('1','29','-3')] → [1,29,-3]
```

Sequence Containers Indexing

for lists, tuples, strings, bytes…

```
negative index   -5  -4  -3  -2  -1
positive index    0   1   2   3   4
lst=[10, 20, 30, 40, 50]
positive slice    0   1   2   3   4   5
negative slice   -5  -4  -3  -2  -1
```

Items count

`len(lst)→5`

index from 0
(here from 0 to 4)

Individual access to **items** via `lst[`*index*`]`

```
lst[0]→10   ⇒ first one     lst[1]→20
lst[-1]→50  ⇒ last one      lst[-2]→40
```

On mutable sequences (`list`), remove with `del lst[3]` *and modify with assignment* `lst[4]=25`

Access to **sub-sequences** via `lst[`*start slice*`:`*end slice*`:`*step*`]`

```
lst[:-1]→[10,20,30,40]   lst[::-1]→[50,40,30,20,10]   lst[1:3]→[20,30]   lst[:3]→[10,20,30]
lst[1:-1]→[20,30,40]     lst[::-2]→[50,30,10]         lst[-3:-1]→[30,40]  lst[3:]→[40,50]
lst[::2]→[10,30,50]      lst[:]→[10,20,30,40,50]  shallow copy of sequence
```

Missing slice indication → from start / up to end.
On mutable sequences (`list`), remove with `del lst[3:5]` *and modify with assignment* `lst[1:4]=[15,25]`

Boolean Logic

Comparators: `<` `>` `<=` `>=` `==` `!=` (*boolean results*) ≤ ≥ = ≠

```
a and b   logical and   both simulta-neously
a or b    logical or    one or other or both
```

pitfall : `and` *and* `or` *return* *value* *of* `a` *or of* `b` *(under shortcut evaluation).*
⇒ *ensure that* `a` *and* `b` *are booleans.*

```
not a     logical not
True
False     True and False constants
```

Statements Blocks

parent statement:
indentation !
statement block 1…
parent statement:
statement block2…
next statement after block 1

configure editor to insert 4 spaces in place of an indentation tab.

Modules/Names Imports

module `truc` ⇔ *file* `truc.py`

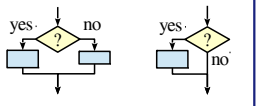
```
from monmod import nom1,nom2 as fct
                →direct acces to names, renaming with as
import monmod  →acces via monmod.nom1 …
```

modules and packages searched in *python path* (cf `sys.path`)

Conditional Statement

statement block executed only ***if*** *a condition is true*

```
if logical condition:
    statements block
```



Can go with several *elif*, *elif*... and only one final *else*. Only the block of first true condition is executed.

```
if age<=18:
    state="Kid"
elif age>65:
    state="Retired"
else:
    state="Active"
```

with a var `x`*:*
`if bool(x)==True:` ⇔ `if x:`
`if bool(x)==False:` ⇔ `if not x:`

Maths

floating numbers… approximated values

Operators: `+` `-` `*` `/` `//` `%` `**`
Priority `(…)` × ÷ ↑ ↑ a^b
integer ÷ ÷ remainder

`@` → matrix × *python3.5+*`numpy`

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
pow(4,3)→64.0
```

usual priorities

angles in radians

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
sqrt(81)→9.0      √
log(e**2)→2.0
ceil(12.5)→13
floor(12.5)→12
```

modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, *etc. (cf. doc)*

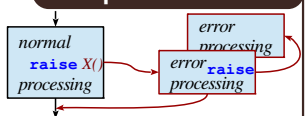
Exceptions on Errors

Signaling an error:

```
raise ExcClass(…)
```

Errors processing:

```
try:
    normal procesising block
except Exception as e:
    error processing block
```



`finally` *block for final processing in all cases.*

Conditional Loop Statement

*statements block executed **as long as** condition is true*

```
while logical condition:
    statements block
```


```
s = 0          # initializations before the loop
i = 1          # condition with a least one variable value (here i)
while i <= 100:
    s = s + i**2
    i = i + 1  # make condition variable change !
print("sum:",s)
```

Algo: $s = \sum_{i=1}^{i=100} i^2$

beware of infinite loops!

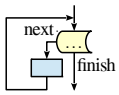
Loop Control

`break` *immediate exit*
`continue` *next iteration*
`else` *block for **normal** loop exit.*

Iterative Loop Statement

*statements block executed **for each** item of a container or iterator*

```
for var in sequence:
    statements block
```



Go over sequence's **values**

```
s = "Some text"   # initializations before the loop
cnt = 0
for c in s:       # loop variable, assignment managed by for statement
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

Algo: count number of `e` in the string.

loop on dict/set ⇔ loop on keys sequences
use *slices* to loop on a subset of a sequence

Go over sequence's **index**

- modify item at index
- access items around index (before / after)

```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```

Algo: limit values greater than 15, memorizing of lost values.

good habit : don't modify loop variable

Go simultaneously on sequence's **index** and **values**:

```
for idx,val in enumerate(lst):
```

Display

```
print("v=",3,"cm :",x,",",y+4)
```

items to display : literal values, variables, expressions

`print` options:

- `sep=" "` items separator, default space
- `end="\n"` end of print, default new line
- `file=sys.stdout` print to file, default standard output

Input

```
s = input("Instructions:")
```

`input` always returns a **string**, convert it to required type (cf. boxed *Conversions* on the other side).

Generic Operations on Containers

`len(c)` → items count
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → `list` sorted *copy*
`val in c` → boolean, membership operator `in` (absence `not in`)
`enumerate(c)` → *iterator* on (index, value)
`zip(c1,c2…)` → *iterator* on tuples containing c_i items at same index
`all(c)` → `True` if **all** `c` items evaluated to true, else `False`
`any(c)` → `True` if **at least one** item of `c` evaluated true, else `False`

*Note: For dictionaries and sets, these operations use **keys**.*

*Specific to **ordered sequences containers** (lists, tuples, strings, bytes…)*

`reversed(c)` → *inversed iterator* `c*5`→ duplicate `c+c2`→ concatenate
`c.index(val)` → *position* `c.count(val)` → *events count*

```
import copy
copy.copy(c)     → shallow copy of container
copy.deepcopy(c) → deep copy of container
```

Operations on Lists

modify original list

`lst.append(val)`	add item at end
`lst.extend(seq)`	add sequence of items at end
`lst.insert(idx,val)`	insert item at index
`lst.remove(val)`	remove first item with value *val*
`lst.pop([idx])` →*value*	remove & return item at index *idx* (default last)
`lst.sort()` `lst.reverse()`	sort / reverse liste *in place*

Operations on Dictionaries

```
d[key]=value      d.clear()
d[key] → value    del d[key]
d.update(d2)      update/add associations
d.keys()
d.values()        →iterable views on keys/values/associations
d.items()
d.pop(key[,default]) → value
d.popitem() → (key,value)
d.get(key[,default]) → value
d.setdefault(key[,default]) →value
```

Operations on Sets

Operators:
`|` → union (vertical bar char)
`&` → intersection
`-` `^` → difference/symetric diff.
`<` `<=` `>` `>=` → inclusion relations
Operators also exist as methods.

```
s.update(s2)  s.copy()
s.add(key)    s.remove(key)
s.discard(key) s.clear()
s.pop()
```

Files

storing data on disk, and reading it back

```
f = open("file.txt","w",encoding="utf8")
```

file **variable** for operations
name of file on disk (+path…)
opening **mode**
- `'r'` read
- `'w'` write
- `'a'` append
- …`'+'` `'x'` `'b'` `'t'`

encoding of chars for *text files*: `utf8` `ascii` `latin1` …

cf. modules `os`, `os.path` and `pathlib`

writing

```
f.write("coucou")
f.writelines(list of lines)
```

reading

read empty string if end of file

```
f.read([n])      → next chars
                   if n not specified, read up to end !
f.readlines([n]) → list of next lines
f.readline()     → next line
```

*text mode `t` by default (read/write `str`), possible binary mode `b` (read/write `bytes`). **Convert from/to required type** !*

`f.close()` dont forget to **close the file** after use !

`f.flush()` write cache
`f.truncate([taille])` resize

reading/writing progress sequentially in the file, modifiable with:

`f.tell()` →*position*
`f.seek(position[,origin])`

Very common: opening with a guarded block (automatic closing) and reading loop on lines of a text file:

```
with open(…) as f:
    for line in f :
        # processing of line
```

Integers Sequences

`range([start,] end [,step])`

start default 0, *fin* not included in sequence, *pas* signed default 1

`range(5)` → 0 1 2 3 4
`range(3,8)` → 3 4 5 6 7
`range(2,12,3)` → 2 5 8 11
`range(20,5,-5)` → 20 15 10
`range(len(seq))` → *sequence of index of values in seq*

range provides an immutable sequence of int constructed as needed

Function Definition

function name (identifier)
named parameters

```
def fct(x,y,z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

`fct`

result value of the call, if no computed result to return: `return None`

parameters and all variables of this block exist only *in* the block and *during* the function call (think of a "black box")

Advanced: `def fct(x,y,z,*args,a=3,b=5,**kwargs):`
**args variable positional arguments (→`tuple`), default values,*
***kwargs variable named arguments (→`dict`)*

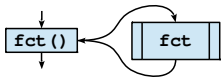
Function Call

```
r = fct(3,i+2,2*i)
```

storage/use of returned value
one argument per parameter

this is the use of function name *with parenthesis* which does the call

*Advanced: *sequence **dict*



Operations on Strings

```
s.startswith(prefix[,start[,end]])   s.strip([chars])
s.endswith(suffix[,start[,end]])
s.count(sub[,start[,end]])           s.partition(sep) → (before,sep,after)
s.index(sub[,start[,end]])           s.find(sub[,start[,end]])
s.is…()  tests on chars categories (ex. s.isalpha())
s.upper()  s.lower()  s.title()  s.swapcase()
s.casefold()  s.capitalize()  s.center([width,fill])
s.ljust([width,fill])  s.rjust([width,fill])  s.zfill([width])
s.encode(encoding)  s.split([sep])  s.join(seq)
```

Formating

formating directives — values to format

```
"modele{} {} {}".format(x,y,r) → str
"{selection:formating!conversion}"
```

- **Selection** :

```
2
nom
0.nom
4[key]
0[2]
```

Examples:

```
"{:+2.3f}".format(45.72793)
→'+45.728'
"{1:>10s}".format(8,"toto")
→'      toto'
"{x!r}".format(x="I'm")
→'"I\'m"'
```

- **Formating** :

fill char *alignment* *sign* *mini width*.*precision~maxwidth* *type*

`< > ^ =` `+ - space` `0` at start for filling with 0

integer: `b` binary, `c` char, `d` decimal (default), `o` octal, `x` or `X` hexa…
float: `e` or `E` exponential, `f` or `F` fixed point, `g` or `G` appropriate (default),
string: `s` … `%` percent

- **Conversion** : `s` (readable texte) or `r` (literal representation)