Pensar Python

Pensar Como um Informático

Versão 1.0

Junho 8

Pensar Python

Pensar Como um Informático

Versão 1.0

Junho 8

Diogo Gomes
João Paulo Barraca

baseado no livro "Think Python"

de Allen B. Downey

Green Tea Press
9 Washburn Ave
Needham MA 02492

Prefácio

O porque deste livro

Em 2013 a Universidade de Aveiro criou a disciplina de Laboratórios de Informática, onde para explorar o mundo da informática recorreu à linguagem Python como ferramenta por excelência. No ano seguinte, em 2014 criou o curso de Licenciatura Engenharia Informática em que a disciplina introdutória à programação (Fundamentos de Programação) pressupõe igualmente o ensino da linguagem Python.

Desafiados os autores a serem regentes de disciplinas focadas nesta linguagem e com base na experiência obtida em anos anteriores a leccionar Python e Java, a alunos do primeiro ano de vários cursos, decidimos produzir um livro de introdução à programação, escrito em Português, que tivesse por base a linguagem Python.

Ao invés de começar um livro do zero, optou-se por fazer uma tradução do excelente livro "Think Python" de Allen B. Downey, disponível segundo a licença Creative Commons. Ao iniciar a tradução, rapidamente concluímos que o mais adequado seria um meio termo, uma adaptação. Procurámos assim manter o espírito do livro original, mas adicionando alguns conteúdos que consideramos importantes para a realidade do ensino da programação em Português. Este livro pretende ser uma ferramenta importante para a aprendizagem da linguagem Python, ao mesmo tempo que fornece bases sólidas para a construção de programas corretos.

Esperamos pois, que com este livro, os leitores possam adquirir os princípios base da programação de computadores, e adquiram as bases para que mais facilmente possa vir a compreender tópicos mais avançados de programação.

Conteúdo

Capítulo 1

Como Programar

O objetivo deste livro é ensinar a pensar como um informático. Esta maneira de pensar combina alguns dos melhores aspetos da matemática, engenharia e ciências naturais. Como matemático, um informático usa linguagens formais para denotar ideias (mais especificamente computações). Como engenheiro projeta soluções, integra componentes em sistemas e avalia o impacto das diversas alternativas. Como cientista, observa o comportamento de um sistema complexo, formula hipóteses e testa as suas teses.

A competência mais importante de um informático é a sua capacidade para **resolver problemas**. Saber resolver um problema significa ser capaz de formular o problema, pensar de forma criativa possíveis soluções, e expressar a solução de forma clara e precisa. O processo de aprender a programar não é mais que uma oportunidade para praticar esta competência. É por esta razão que este capitulo tem o título de "Como Programar".

Num primeiro patamar, o leitor vai aprender a programar, uma competência essencial para um informático. Mas num patamar superior, vai fazer uso da programação como um meio para atingir um fim. À medida que vamos progredindo, este propósito tornar-se-á mais claro.

1.1 A Linguagem de programação Python

A linguagem de programação que vai aprender é a Python. A linguagem Python é um exemplo de uma **linguagem de alto nível**; poderá ainda ter ouvido falar de outras linguagens de alto nível, tais como as linguagens C, C++, PHP ou Java.

Também existem **linguagens de baixo nível**, por vezes apelidadas de "linguagens máquina" ou "linguagens *assembly*". Sem entrar em muitos detalhes, um computador só pode executar programas escritos em linguagens de baixo nível. Tal implica que um programa escrito numa linguagem de alto nível, tenha de ser processado (e convertido para uma linguagem de mais baixo nível) antes que possa ser executado. Este processo leva algum tempo, o que se torna uma desvantagem das linguagens de alto nível.

Por outro lado, as vantagens são imensas. Em primeiro lugar, é muito mais fácil programar numa linguagem de alto nível. Um programa escrito numa linguagem de alto nível leva menos tempo a ser escrito, é mais curto e fácil de ler, sendo maior a probabilidade de ser

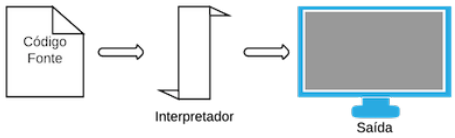

Figura 1.1: Um interpretador processa o programa passo a passo, realizando computações e lendo linhas alternadamente

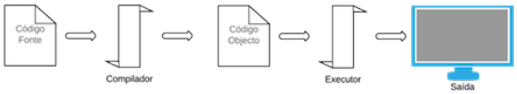

Figura 1.2: Um compilador traduz o código fonte em código objeto, que por sua vez é executado pelo executor de hardware

correto. Em segundo lugar, as linguagens de alto nível são **portáveis**, o que significa que podem ser executadas em computadores diferentes com poucas ou nenhumas alterações. Uma programa escrito numa linguagem de baixo nível apenas executa num dado tipo de computador, caso seja necessário executar o mesmo programa num computador diferente, o mesmo tem de ser re-escrito.

Devido a estas vantagens, quase todos os programas modernos são escritos numa linguagem de alto nível. Ficando o uso das linguagens de baixo nível vedado a utilizações muito especializadas e específicas.

Dois tipos de programas processam linguagens de alto nível em linguagens de baixo nível: **interpretadores** e **compiladores**. Um interpretador lê o programa de alto nível e executa-o, o mesmo que dizer que desempenha as instruções pedidas pelo programa. O interpretador processa o programa aos poucos, progressivamente lê uma linha ou pedaço de código e executa a computação presente na mesma. A Figure 1.1 apresenta a estrutura de um interpretador.

Um compilador por sua vez lê a totalidade do programa e traduz o mesmo, antes que qualquer computação seja realizada. Neste contexto a linguagem de alto nível é apelidada de **código fonte**, e à sua tradução **código objeto** ou **executável**. A Figure 1.2 apresenta a estrutura de um compilador.

A linguagem Python é considerada uma linguagem interpretada, uma vez que um programa escrito em Python é executado por um interpretador. Existem duas formas de usar o interpretador: **modo interactivo** e **modo *scripting***. O modo interativo utiliza-se executando o comando que invoca o interpretador, que tipicamente é **python** (poderá também incluir a versão e ser **python2.7** ou **python3.5**). À medida que vai escrevendo código em Python, o interpretador apresenta o resultado de cada instrução:

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

O símbolo, »>, é a ***prompt*** utilizada pelo interpretador para indicar que se encontra disponível a processar instruções. Se escrever 1 + 1, o interpretador responde 2.

Em alternativa, pode armazenar o código num ficheiro e usar o interpretador para executar os conteúdos do ficheiro, a que chamamos de guião (em inglês: *script*). Por convenção, os *scripts* Python têm nomes que acabam em `.py`. O sufixo do nome é também apelidado de extensão.

Para executar o *script*, é necessário indicar ao interpretador o nome do ficheiro. Se tiver um *script* chamado `xpto.py` e estiver num ambiente de linha de comandos UNIX[1], pode escrever `python xpto.py`. Noutros ambientes de desenvolvimento, os detalhes para a execução de um script são diferentes. Pode encontrar instruções para o seu ambiente de desenvolvimento no sitio da *Web* da linguagem Python em `http://python.org`.

Trabalhar no modo interativo é conveniente para testar pequenos trechos de código, uma vez que os mesmos executam imediatamente. Mas para tudo mais que meia dúzia de linhas, deverá guardar o seu código num *script* para que possa mais tarde modificar e executar o programa.

1.2 O que é um Programa?

Um **Programa** é uma sequência de instruções que especificam como uma computação deve ser desempenhada. A computação pode ser matemática, tal como a resolução de um sistema de equações ou o cálculo das raízes de um polinómio, mas também pode ser simbólica, tal como a pesquisa e substituição de uma palavra num documento ou a compilação de um programa.

Os detalhes são diferentes de linguagem para linguagem, mas algumas instruções base são partilhadas por todas as mesmas:

entrada: Adquirir dados do teclado, de um ficheiro, ou de outra fonte.

saída: Apresentação de dados no ecrã, envio para um ficheiro ou outro destino.

matemáticas: Desempenham operações aritméticas básicas tais como a adição e a multiplicação.

execução condicional: Verificam certas condições e executam o código apropriado.

repetição: Desempenham uma ação de forma repetitiva, normalmente com uma pequena variação.

Acredite ou não, estas são todas as instruções que precisa de conhecer. Todos os programas que já alguma vez utilizou, não importa o quão complexos, são constituídos por instruções destas. Assim é possível pensar a programação como o processo de decomposição de uma tarefa grande e complexa em sub-tarefas e sub-sub-tarefas cada vez mais pequenas e simples, até que as mesmas possam ser descritas apenas pelas instruções básicas anteriores.

Este conceito, apresenta-se neste momento como algo vago. No entanto regressaremos a este assunto quando abordarmos o que são **algoritmos**.

[1] Este ambiente existe na maioria dos sistemas, sendo nativo em Linux, BSD e OS X

1.3 O que é a Depuração?

A programação é uma atividade sujeita ao erro. Por razões históricas, os erros em programação apelidam-se de ***bugs*** e o processo para os encontrar chama-se ***debugging*** (traduzido para português como: depuração).

Podem existir três tipos de erros num programa: erros de sintaxe, erros em tempo de execução e erros semânticos. É útil fazer a distinção entre os mesmos para permitir uma mais rápida resolução destes.

1.3.1 Erros de sintaxe

O interpretador de Python apenas executa um programa se a sintaxe do mesmo estiver correta; caso contrário, o interpretador apresenta uma mensagem de erro. Por **sintaxe** referimos-nos à estrutura do programa e às regras dessa estrutura. Por exemplo, os parênteses existem sempre aos pares, tal que `(1 + 2)` está correto, e `8)` apresenta um **erro sintático**.

Em Português um leitor pode tolerar a maior parte dos erros de sintaxe, razão pela qual conseguimos ler a maioria das mensagens escritas nas redes sociais, sem desistir ao encontrar o primeiro erro sintatico. Já as linguagens de programação não são tão condescendentes. Se existir um qualquer erro de sintaxe no programa, o interpretador de Python irá sinalizar o mesmo através de uma mensagem de erro, e não será possível executar o programa. Se este for o seu primeiro contacto com uma linguagem de programação, provavelmente muito do tempo será despendido nas tarefas de encontrar e corrigir erros sintáticos. À medida que a experiência for aumentando, fará menos erros, e os que fizer serão encontrados mais rapidamente.

1.3.2 Erros em tempo de execução

O segundo tipo de erros são os de tempo de execução (em inglês: *runtime error*), assim denominados porque o mesmo só surge durante a execução do programa. Estes erros, também chamados de **exceções** (em inglês: *exceptions*) surgem como o próprio nome indica de forma excecional. Ocorrem quando, estando o programa construído com uma sintaxe correta, tenta executar operações que são inválidas para a informação em causa, ou quando existe um evento externo à programação (ex.: a memória de um computador ficou cheia).

Os erros de tempo de execução são raros nos programas mais simples que irá desenvolver nos próximos capítulos, pelo que poderá levar algum tempo até se deparar com o primeiro.

Um exemplo simples de um erro de execução acontece se tentar realizar uma divisão em que o quociente é 0. A instrução seria representada em Python na forma `a / b` o que é perfeitamente válido do ponto de vista sintático. No entanto, se `b` for igual a 0, não será possível realizar o cálculo e o resultado é um erro em tempo de execução.

1.3.3 Erros semânticos

O terceiro tipo de erros são os semânticos. Se existir um erro semântico no seu programa, o programa executará normalmente no sentido que o computador não irá comunicar o erro

com mensagens específicas, mas também não irá fazer o que seria esperado. Fará qualquer outra coisa, mais concretamente, exatamente aquilo que instruiu no seu programa.

O problema é que o programa que escreveu não é o programa que pretendia ter escrito. Tal deve-se ao facto do significado do programa (semântica) estar errado. Identificar um erro semântico pode não ser trivial, pois requer um processo de retrospetiva e análise em que, com base no *output* do programa, se procuram as razões para o mesmo.

Se considerar que quer realizar a soma de dois valores `a` e `b`, esta soma pode ser representada em Python através da instrução `a + b`. Irá ocorrer um erro semântico se em alternativa a instrução for `a - b`. Neste caso não existem qualquer erro sintático ou em tempo de execução. No entanto o resultado será a diferença entre os valores e não a sua soma, o que constitui um erro semântico.

1.4 Depuração

Uma das competências mais importantes que irá adquirir, é a de ser capaz de fazer depuração. Embora possa parecer muitas das vezes uma tarefa frustrante e útil apenas para os iniciantes, a depuração é das componentes mais intelectualmente desafiantes, interessantes e ricas intelectualmente da programação.

De certa maneira, a depuração é um trabalho de detetive. É confrontado com um conjunto de pistas, e é lhe pedido que deduza o processo e os eventos que conduziram às evidências com que se depara.

A depuração é também como as ciências experimentais. Assim que se tem uma hipótese sobre o que está a acontecer de errado, modifica-se o programa e tenta-se de novo. Se a hipótese estiver correta, consegue prever o resultado da modificação, e o programa aproxima-se de um programa correto. Caso a hipótese esteja errada, pode criar uma nova hipótese. Tal como disse *Sherlock Holmes*, "Quando eliminar o impossível, o que restar, por mais improvável, deverá ser a verdade". (A. Conan Doyle, "O sinal dos quatro")

De forma alguma deve-se considerar que a depuração é uma tarefa apenas para iniciantes, pois só estes cometem erros. Isto é completamente falso, sendo que a depuração é uma componente omnipresente da programação independentemente da experiência do programador. Para algumas pessoas, programar e depurar são a mesma coisa. Isto é, programar é um processo de depuração gradual até que o programa desempenhe o pretendido. A ideia subjacente é que deve criar programas que façam *qualquer coisa* e que aos poucos vá realizando **pequenas alterações** e fazendo depuração, de tal modo que exista sempre qualquer coisa a funcionar.

Por exemplo, o Linux é um sistema operativo que contem milhões de linhas de código, mas começou como um programa muito simples desenvolvido por Linus Torvalds para explorar o processador 80368 da Intel. De acordo com Larry Greenfield, "Um dos projetos iniciais do Linus era criar um programa que alternaria entre escrever AAAA e BBBB. Mais tarde evoluiu para o Linux." (*The Linux User's Guide* Beta Version 1)

Nos próximos capítulos serão feitas mais sugestões sobre como fazer depuração e serão apresentadas outras técnicas para melhorar a correção dos programas desenvolvidos.

A programação, e especialmente a depuração, desperta por vezes emoções fortes. Quando se depara com um *bug* difícil, pode se sentir zangado, impotente ou até mesmo embaraço.

Existem provas evidenciais que as pessoas respondem naturalmente a um computador como se ele trata-se de uma pessoa. Quando eles trabalham bem, olhamos para eles como um colega, e quando eles são obstinados ou rudes, respondemos de volta da mesma maneira que respondemos a alguém que seja obstinada ou rude para conosco. (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*)

Prepararmo-nos para estas reacções, ajuda-nos a melhor lidar com as mesmas. Uma forma de lidar com estes sentimentos é ver o computador como um empregado, com certas qualidades tais como velocidade e precisão, e com defeitos tais como a falta de empatia e a incapacidade para perceber o contexto.

A sua tarefa é ser um bom gestor: encontrar maneiras de tirar partido das qualidades e mitigar os defeitos. Acima de tudo, usar as nossas emoções para lidar com o problema, sem permitir que as reações interfiram com a nossa capacidade de trabalhar efetivamente.

Aprender a fazer depuração pode ser extremamente frustrante, mas é uma capacidade essencial em inúmeras atividades para lá da programação. Em cada capitulo haverá uma secção como esta em que partilhamos algumas ideias/conceitos de como fazer depuração

1.5 Linguagens Formais e Naturais

As **linguagens naturais** são as linguagens que as humanos falam, tal como o Português, Inglês e o Francês. Não foram desenhadas por um comité (embora por vezes surja um a tentar impor regras sobre a mesma), evoluíram aos longo do tempo de forma natural.

As **linguagens formais** são linguagens que foram desenhadas por pessoas e/ou comités para aplicações específicas. Por exemplo, a notação utilizada pelos matemáticos é uma linguagem formal que é particularmente boa a transmitir a relação entre números e símbolos. Os químicos usam uma linguagem formal para representar a estrutura química de uma molécula. E mais importante:

> **As Linguagens de Programação são linguagens formais que foram desenhadas para exprimir computações.**

As linguagens formais tendem a ter um conjunto de regras rígidas em relação à sintaxe de forma a evitar ambiguidades. Por exemplo, `3 + 3 = 6` é uma afirmação matemática sintaticamente correta, mas `3+ = 3$6` não o é. H_2O é uma fórmula química sintaticamente correta, já $_2Zz$ não o é.

As regras de sintaxe estão tipificadas em duas: as que dizem respeito aos símbolos (em inglês: *tokens*) e as que dizem respeito à estrutura. Os símbolos são os elementos básicos da linguagem, tais como palavras, números e elementos químicos. Um dos problemas com `3+ = 3$6` é que $ não é um um símbolo válido em matemática (tanto quanto os autores sabem). Similarmente, $_2Zz$ não é válido porque não existe um elemento com a abreviação Zz.

Já o segundo tipo de erro de sintaxe refere-se à estrutura da instrução; isto é, à maneira como os símbolos estão organizados. A instrução `3 + = 3` é inválida porque muito embora os símbolos + e = sejam válidos, não podem existir um após o outro. De forma semelhante, numa formula química o subscrito vem após o nome do elemento e nunca antes (tal como o 2 em O_2).

Quando lê uma frase em Português or uma instrução numa linguagem formal, necessita de perceber a estrutura da frase. Muito embora no Português o faça de forma subconsciente, é sempre necessário interpretar a estrutura. A este processo chamamos de análise (em inglês: ***parsing***).

Por exemplo, quando ouve a frase, "O António fez uma grande maratona", percebe de imediato que "O António" é o sujeito e que "fez uma grande maratona" é o predicado. Assim que analisou a frase, pode perceber o que ela significa, ou seja a semântica da frase. Assumindo que sabe o que é uma maratona e o seu tamanho, compreenderá também a implicação da frase.

Embora as linguagens formais e naturais tenham muitos aspetos em comum tais como símbolos, estrutura, sintaxe, e semântica, existem algumas diferenças.

Ambiguidade: As linguagens naturais estão repletas de ambiguidade. As pessoas lidam com esta ambiguidade através do contexto em que as mesmas se encontram e partindo de outras fontes de informação. No exemplo anterior não é claro que o António tenha realmente feito algum exercício físico, visto que pode ter realizado algo com um nível esforço mais intelectual. Por exemplo, pode ter estado uma semana a estudar para um exame ou a visionar uma série de TV. Já as linguagens formais são desenhadas para serem o mais completamente desprovidas de ambiguidade, o que significa que uma afirmação tem exatamente um significado, independentemente do contexto.

Redundância: De maneira a ultrapassar a ambiguidade e reduzir mal entendidos, as linguagens naturais empregam enumeras redundâncias. Como resultado, são pouco concisas. No caso deste exemplo, é assumido que uma maratona é algo extenso podendo ser considerado redundante a utilização da palavra "grande". As linguagens formais são menos redundantes e mais concisas.

Literalidade: As linguagens naturais estão repletas de idiomáticas e metáforas. Se disser, "O António fez uma grande maratona", tal como já identificado há uma probabilidade elevada de o António não ter feito nenhuma ação com esforço físico (esta idiomática refere-se à execução de algo com uma dificuldade intrínseca). As linguagens formais dizem sempre literalmente o que pretendem.

Alguém que tenha crescido a falar uma linguagem natural (todos nós) tem normalmente alguma dificuldade em se ajustar a uma linguagem formal. De certa maneira, a diferença entre uma linguagem formal e uma linguagem natural assemelha-se à diferença entre poesia e prosa, com acréscimos:

Poesia: As palavras são usadas pelo seu som assim como pelo seu significado, e o poema no seu todo cria efeitos ou respostas emocionais. A ambiguidade não só é comum como deliberada, podendo existir igualmente redundâncias.

Prosa: O significado literal das palavras é mais importante, e a estrutura contribui para o significado. A prosa é mais fácil de ser analisada que a poesia, mas contém ainda algumas ambiguidades.

Programas: O significado de um programa de computador é inequívoco e literal, e pode ser compreendido inteiramente através da análise dos seus símbolos e estrutura.

Podemos desde já apresentar algumas sugestões para ler melhor programas (e outras linguagens formais). Primeiro, considere que uma linguagem formal é muito

mais densa que uma linguagem natural (não existe redundância), pelo que leva mais tempo a ler. Mais, a sua estrutura é muito importante, pelo que não é apropriado ler de cima a baixo, da esquerda para a direita. Deve-se sim, analisar o programa mentalmente, identificando os símbolos e interpretando a sua estrutura linha a linha e bloco a bloco. Por fim, os detalhes são importantes. Pequenos erros ortográficos e de pontuação, que podem passar despercebidos numa linguagem natural, fazem grandes diferenças numa linguagem formal.

1.6 O primeiro programa

Tradicionalmente, o primeiro programa escrito numa nova linguagem de programação é chamado de "Hello, World!" (em português: "Olá, Mundo!") uma vez que a única coisa que faz é escrever as palavras "Hello, World!". Em Python, instruimos o computador assim:

```
print("Hello, World!")
```

Isto é um exemplo de uma **instrução de impressão**, que na realidade não imprime nada em papel. Apresenta sim um valor no ecrã. Neste caso, o resultado são as palavras

```
Hello, World!
```

As aspas no programa delimitam o inicio e fim do texto que queremos ver apresentado no ecrã, no entanto não fazem parte do que é apresentado.

Os parêntesis indicam que `print` é uma função (tal como as funções matemáticas). Voltaremos às funções no chapter 3.

É uma boa ideia ler este livro em frente a um computador por forma a experimentar de imediato os inúmeros exemplos com que se irá deparar. Poderá executar a maioria destes no modo interativo, mas caso os coloque em scripts será mais fácil experimentar variações dos mesmos.

Sempre que estiver a experimentar uma nova funcionalidade, deve procurar fazer alguns erros de forma propositada. Por exemplo, no programa "Hello, World!", o que será que acontece se não colocar uma das aspas? E se omitir ambas? O que acontece se `print` estiver mal escrito (p.ex, `prinl`?).

Este tipo de experiências ajuda-o a lembrar-se do que leu; ajuda também ao desenvolvimento das suas capacidades de análise, já que desta maneira percebe melhor o que as mensagens de erro do interpretador significam. É melhor cometer erros agora e propositadamente, pois sabe exatamente o que foi alterado, do que no futuro acidentalmente.

Capítulo 2

Variáveis, Expressões e Declarações

2.1 Valores e Tipos

Um **valor** é uma das coisas mais básicas com que um programa opera, pode ser uma letra ou um número. Os valores com que nos deparámos até ao momento foram o `1`, `2` e `"Hello World"`.

Estes valores são de tipos diferentes: `2` é um número inteiro, e `"Hello World"` é uma ***string*** (termo inglês que denota uma sequência de carateres alfanuméricos). Tanto o leitor como o interpretador identificam uma *string* pelo facto da mesma estar entre plicas ou aspas. Por convenção usamos plicas quando temos uma palavra e aspas quando temos mais que uma palavra.

Se não estiver certo do tipo de um valor, o interpretador pode auxiliar, indicando o tipo de qualquer variável:

```
>>> type("Hello, World!")
<class 'str'>
>>> type(17)
<class 'int'>
```

Como esperado, as *strings* são do tipo `str` e os números inteiros pertencem ao tipo `int`. Menos óbvio, os números racionais pertencem a um tipo chamado de `float`, uma vez que os números são representados num formato denominado de vírgula-flutuante (em inglês **floating-point**).

```
>>> type(3.2)
<class 'float'>
```

É importante referir que na cultura anglo-saxónica é usado o ponto (`.`) para separar a parte inteira da decimal. Enquanto na cultura Portuguesa é utilizada a vírgula (`,`). O Python segue a nomenclatura anglo-saxónica.

De que tipo serão valores como `'17'"" e \verb`'3.2'!? Embora aparentem ser números, estão entre plicas como as *strings*.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

São pois *strings*.

Quando escreve um numero inteiro de grande dimensão, existe a tentação de separar os milhares com o ponto (.), como em 1.000,00. Embora o interpretador de Python não interprete este número como um `float`, continua a ser um tipo válido.

```
>>> 1.000,00
(1.0, 0)
```

Por ventura inesperado para leitor, o Python interpreta 1.000,00 como uma sequência de valores, separados pela vírgula. Na **??** iremos abordar o que são estas sequências. Se considerar que 1.000,00 é um valor, a situação exemplifica um erro semântico: o código está correto e não produz mensagem alguma de erro, mas infelizmente não produz o efeito desejado.

2.2 Variáveis

Uma das características mais poderosas das linguagens de programação é a sua capacidade para manipular **variáveis**. Uma variável é um nome pelo qual nos referimos a um valor.

Uma instrução de atribuição cria novas variáveis e atribui-lhes valores:

```
>>> mensagem = "Falam, falam e não os vejo a fazer nada"
>>> n = 17
>>> pi = 3.1415926535897932
```

Este exemplo cria três atribuições. A primeira atribui uma *string* a uma variável chamada `mensagem`; a segunda atribui o numero inteiro 17 a `n`; a terceira atribui o valor (aproximado) de π a `pi`.

Uma forma comum de representar variáveis em papel é escrever o nome destas com uma seta a apontar para o valor da variável. Este tipo de figura é chamado de **diagrama de estado** uma vez que representa em que estado se encontra cada uma das variáveis (como se tratasse do estado mental da variável). A Figure 2.1 apresenta o resultado do exemplo anterior.

O tipo de uma variável é o tipo do valor ao qual a mesma se refere.

```
>>> type(messagem)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

Exercício 2.1. *Se escrever um `int` com um 0 no início, pode obter um erro confuso:*

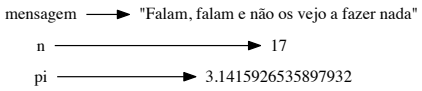

Figura 2.1: Diagrama de estado

```
>>> num = 02492
                ^
SyntaxError: invalid token
```

No entanto, números começados por 0 e uma letra como o `x` *ou o* `b` *são válidos, embora apresentem um comportamento "estranho":*

```
>>> num = 0x2132
>>> num
1114
>>> num = 0x10
>>> num
16
>>> num = 0x1
>>> num
1
```

Consegue perceber o que está a ocorrer? Pista: experimente também os números: `0b01`*,* `0b010`*,* `0b0100` *e* `0b01000`*.*

2.3 Nomes de variáveis e palavras reservadas

Um programador escolhe sempre um nome com significado para as variáveis - desta forma documenta-se o uso dado à variável. Deve evitar-se sempre utilizar letras únicas para variáveis, pois pouco dizem sobre o seu uso. Considere que a variável tem o nome `a`, o que diz este nome sobre o seu uso? É bem mais fácil de entender um programa se ela se chamar `"altura"`.

O nome de uma variável pode ser arbitrariamente extenso. Pode conter letras e números, mas precisa de começar por uma letra. É válido usar letras maiúsculas, mas é uma boa ideia começar o nome da variável com uma letra minúscula (a razão será abordada mais tarde).

O caráter *underscore*, `_`, pode aparecer no nome. É muito comum usar em nomes que tenham múltiplas palavras, tais como `"meu_nome"` ou `"velocidade_do_vento"`.

Se atribuir um nome inválido a uma variável, obterá um erro de sintaxe:

```
>>> 10ovos = 'dezena de ovos'
SyntaxError: invalid syntax
```

```
>>> muitos@ = 1000000
SyntaxError: invalid syntax
>>> class = 'A'
SyntaxError: invalid syntax
```

`10ovos` é inválido porque não começa por uma letra. `muitos@` é inválido porque contém o caráter `@`. Mas o que dizer de `class`?

Acontece que `class` é uma das palavras reservadas (em inglês: **keywords**) da linguagem Python. O interpretador usa as palavras reservadas para reconhecer a estrutura do programa, como tal não podem ser utilizadas como nome de variáveis.

A linguagem Python, na sua versão 3, possui 33 palavras reservadas:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Convém ter presente esta lista. Se o interpretador sinalizar um erro com uma variável e não conseguir perceber o porquê, verifique se o nome da variável não faz parte desta lista. Se utilizar um editor preparado para a linguagem Python, este irá assinalar as palavras reservadas com uma cor diferente. Portanto torna-se simples evitar a sua utilização ao nomear variáveis.

2.4 Operadores e Operandos

Operadores são símbolos especiais que representam computações, tais como a adição ou multiplicação. Os valores aos quais são aplicados os operadores são chamados de **operandos**.

Os operadores `+`, `-`, `*`, `/` e `**` permitem fazer adições, subtrações, multiplicações, divisões e mesmo calcular exponenciais, como nos exemplos seguintes:

```
20+32
hora - 1
hora * 60 + minuto
minuto / 60
5 ** 2
(5 + 9) * (15 - 7)
```

Em algumas linguagens, o caráter `^` é utilizado para calcular exponenciais, mas em Python este caráter representa o operador binário de *ou exclusivo* também conhecido por `XOR`. As operações binárias não são abordadas neste livro, mas pode ler mais sobre as mesmas em `http://wiki.python.org/moin/BitwiseOperators`.

Como já mencionado, os numeros podem ser do tipo `int` ou `float`. Conforme o tipo a operação aritmética dara origem um resultado do tipo `int` ou `float`. Operações `+`, `-` ou

`*` entre `int`'s dão origem a `int` e sempre que um dos operandos for do tipo `float` dará origem a `float`.

A excepção encontra-se no operador `/`, o resultado desta divisão é um real (`float`) mesmo que o dividendo e o divisor sejam inteiros (`int`). O operador `//` é que força o cálculo da divisão inteira.

Como complemento à divisão inteira o operador `%` devolve o resto da divisão inteira.

```
>>> minuto = 90
>>> minuto / 60.0
1.5
>>> minuto // 60
1
>>> minuto % 60
30
```

2.5 Expressões e Instruções

Uma **expressão** é a combinação de valores, variáveis e operadores. Um valor só por si é considerado uma expressão, assim como uma variável, pelo que as seguintes linhas são todas válidas (assumindo que foi atribuído um valor à variável `x`):

```
17
x
x + 17
```

Uma **instrução** é uma unidade de código que o interpretador de Python consegue executar. Já trabalhámos com dois tipos de instruções: impressão e atribuição.

Tecnicamente uma expressão é também uma instrução, mas é provavelmente mais simples pensar nas duas como sendo coisas distintas. A diferença mais importante é que uma expressão tem um valor, podendo ser reutilizada, enquanto uma instrução não.

2.6 Modo Interactivo e Modo *scripting*

Um dos benefícios de trabalhar com uma linguagem interpretada é que podemos experimentar pequenos pedaços de código em modo interativo, isto mesmo antes de colocar o código num programa (*script*). Mas existem algumas diferenças entre o modo interativo e o modo *scripting* que podem gerar alguma confusão.

Por exemplo, se utilizar o interpretador de Python como uma calculadora, escreve algo como:

```
>>> km = 42.195
>>> km * 0.621371
26.2188
```

A primeira linha atribui um valor à variável `km`, mas não possui um efeito visível (possui um efeito claro: a variável `km` passará a ter o valor 42,195). Já a segunda linha é uma expressão, pelo que o interpretador avalia-a e apresenta o resultado. Assim aprendemos que uma maratona, ou seja 42,195 quilómetros, correspondem a 26.2188 milhas.

Mas se escrever o mesmo código num *script* e o executar, não vai obter qualquer resultado para o ecrã. Em modo de *scripting* uma expressão, quando sozinha, não tem qualquer efeito visível. O Python na realidade avalia a expressão, mas não apresenta o seu valor a menos que isto seja explicitamente pedido:

```
km = 42.195
print(km * 0.621371)
```

Este comportamento pode ser confuso para um iniciante na linguagem.

Um *script* contém uma sequência de instruções. Se existir mais do que uma instrução, os resultados aparecem um de cada vez à medida que as instruções vão sendo executadas.

Por exemplo, o *script*:

```
print(1)
x = 2
print(x)
```

apresenta no ecrã:

```
1
2
```

A instrução `x = 2` não produz qualquer manifestação externa.
Exercício 2.2. *Para verificar isto, escreva as seguintes instruções no interpretador de Python e repare no que elas apresentam:*

```
5
x = 5
x + 1
```

Agora coloque as mesmas instruções num script e execute-o. O que vê no ecrã? Altere o script transformando cada expressão numa instrução de impressão e execute-o de novo.

2.7 Ordem das operações

Quando surge mais do que um operador numa expressão, a ordem pela qual a mesma é avaliada depende de **regras de precedência**. Para os operadores matemáticos, a linguagem Python segue a convenção matemática. O acrónimo **PEMDAS** é útil para nos recordar destas regras:

- **P**arênteses têm o mais alto nível de precedência e podem ser utilizados para forçar uma expressão a ser avaliada pela ordem desejada. Uma vez que as expressões entre parênteses são as primeiras a ser avaliadas, `2 * (3 - 1)` é 4, e `(1 + 1) ** (5 - 2)` é 8. Também pode utilizar parênteses para tornar a leitura de uma expressão mais simples, como em `(minuto * 100) / 60`, mesmo que isto não altere o resultado.

- **E**xponenciação é o nível mais elevado seguinte, de maneira a que `2 ** 1 + 1` seja 3, não 4, e `3 * 1 ** 3` seja 3, não 27.
- **M**ultiplicação e **D**ivisão têm a mesma precedência, que é maior que a da **A**dição e **S**ubstração, que também partilham a mesma precedência. Assim`2 * 3 - 1` é 5, não 4, e `6 + 4 / 2` é 8, não 5.
- Operadores com a mesma precedência são avaliados da esquerda para a direita (exceto a exponenciação). Assim na expressão `graus / 2 * pi`, a divisão acontece antes da multiplicação. Para dividir por 2π, é necessário utilizar parênteses (`graus / (2 * pi)` ou escrever `graus / 2 / pi`.

Na prática não é necessário ter presente as regras de precedência para outros operadores, uma vez que sempre que existir dúvida, pode-se recorrer ao uso de parênteses por forma a tornar as coisas claras.

2.8 Operações com *Strings*

De uma maneira geral não são permitidas operações matemáticas com *strings*, mesmo que a *string* se pareça com um número. Assim as instruções seguintes são inválidas:

```
>>> '2'-'1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>> 'ovo'/'simples'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>> 'quatro' * 'cinco'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

O operador `+`, que não é apresentado no exemplo anterior, funciona com *strings*, mas produz um efeito aparentemente inesperado: produz a **concatenação**, que não é mais do que a junção das *strings* extremo a extremo. Um exemplo:

```
primeira = 'para'
segunda = 'quedas'
print(primeira + segunda)
```

O resultado deste programa será `paraquedas`.

O operador `*` também funciona com *strings*, produzindo repetição. Por exemplo, `'eh' * 3` produz `'eheheh'`. Se um dos operandos for uma *string*, o outro necessita ser um inteiro.

Esta utilização do `+` e do `*` faz sentido se fizermos uma analogia com a adição e multiplicação. Tal como `4 * 3` é equivalente a `4 + 4 + 4`, é expectável que `'eh' * 3` seja o mesmo que `'eh' + 'eh' + 'eh'`, e de facto assim é. Por outro lado, a concatenação de *strings* e a repetição podem ser muito diferentes da adição e multiplicação. Pense numa propriedade da adição que a concatenação não partilha[1].

[1] Comutatividade

2.9 Comentários

À medida que os programas vão crescendo em tamanho e complexidade, tornam-se também mais difíceis de ler e compreender. As linguagens formais são densas, e é muitas vezes difícil olhar para um pedaço de código e perceber o seu papel, ou o porquê de cada linha.

Um primeiro **passo importante** para melhorar a compreensão de um programa é o de dar às variáveis nomes com significado. No entanto, é frequentemente necessário adicionar mais informação para auxiliar a sua compreensão.

Por esta razão, é uma boa ideia acrescentar notas ao programa de forma a explicar em linguagem natural o que os programas estão a fazer. A estas notas dá-se o nome de **comentários**, e começam sempre pelo símbolo `#`.

Ao contrário de outras linguagens, em Python não existem blocos de comentários sobre várias linhas. Cada linha individual de comentário tem obrigatoriamente de se iniciar com o símbolo `#`.

```
# calcula a percentagem de tempo que já passou numa hora
percentagem = (minuto * 100) / 60
```

Neste caso, o comentário aparece sozinho na linha anterior, mas pode também colocar o comentário no final da linha:

```
percentagem = (minuto * 100) / 60 # percentagem de tempo que já passou numa hora
```

Tudo o que se segue ao `#` até ao final da linha é ignorado — não produz qualquer efeito no programa.

Os comentários são extremamente úteis quando documentam propriedades ou funcionalidades menos óbvias do código. É razoável assumir que um leitor consegue perceber o que um dado código *faz*, mas difícil assumir que percebe a *utilidade* ou *propósito* do mesmo. Por outro lado é frequente assinalar com notas potenciais problemas ou limitações, ou mesmo aspetos a melhorar num algoritmo.

No entanto, nem todos os comentários são úteis. Um comentário como este é redundante e despropositado:

```
velocidade = 5      # atribui 5 à variável velocidade
```

Já este comentário contém informação útil, não presente no código

```
velocidade = 5      # velocidade em metros/segundo
```

2.10 Depuração

Neste momento o erro de sintaxe mais provável de ter cometido é a utilização de um nome inválido para uma variável, como por exemplo `global` que é uma *keyword*, ou `mãe` e `€uros`, que contêm caracteres inválidos. A versão 3 da linguagem Python aceita a palavra `mãe` como nome de uma variável. No entanto a utilização de carateres acentuados é altamente desaconselhada.

Se colocar um espaço no nome de uma variável, o interpretador de Python assume tratarem-se de dois operandos sem qualquer operador:

```
>>> nome familia = "Silva"
SyntaxError: invalid syntax
```

No caso dos erros de sintaxe, as mensagens de erro não ajudam muito. As mensagens mais comuns são `SyntaxError: invalid syntax` e `SyntaxError: invalid token`, nenhuma das quais com muitos detalhes. Isto acontece porque o interpretador perdeu a capacidade de processar o programa e portanto não consegue dizer muito mais.

O erro de *runtime* mais comum deverá ser o "`use before def;`", isto é, tentativa de uso de uma variável antes da à mesma ter sido atribuído um valor. É muito comum este erro ser causado por uma variável mal escrita (`montante`/`momtante`):

```
>>> montante = 327.68
>>> juros = momtante * taxa
NameError: name 'momtante' is not defined
```

Os nomes das variáveis são sensíveis ao uso de maiúsculas e minúsculas, pelo que `montante` não é o mesmo que `Montante`.

O erro semântico mais comum deverá ser a criação de expressões em que o nível de precedência não foi bem compreendido. Por exemplo, para avaliar $\frac{1}{2\pi}$, podemos ser tentados a escrever:

```
>>> 1.0 / 2.0 * pi
```

No entanto, como a divisão ocorre primeiro, o resultado obtido seria $\pi/2$, o que obviamente não é a mesma coisa! Não existe um meio do interpretador de Python saber o que pretendia escrever, pelo que o interpretador não lança nenhuma mensagem de erro; simplesmente obterá um resultado errado.

Capítulo 3

Funções

3.1 Invocação de funções

No contexto da programação em Python, **função** é nome que se atribui a um bloco de declarações que realizam uma qualquer computação fazendo corresponder um resultado. Quando se define uma função, é necessário especificar o nome e indicar que declarações fazem parte da função. Noutros locais do programa pode "invocar" este bloco de declarações (a função) pelo nome atribuído a esta. Anteriormente já foi demonstrado um exemplo de uma **invocação de uma função** quando se apresentou o programa "Hello, World!". Muitas outras podem ser utilizadas:

```
>>> type(32)
<class 'int'>
```

O nome da função é `type`. A expressão em parênteses possui o nome de **argumento** da função. O resultado, para esta função em particular, é o tipo do argumento '32' que é um número inteiro ('`int`').

É comum referir que uma função "aceita" ou "tem" um argumento e "devolve" ou "retorna" um resultado. Este resultado é frequentemente chamado de "resultado de retorno" e não tem a ver com o que a função imprime, mas sim com um valor que ela efetivamente devolve. Isto é muito semelhante à matemática, onde a expressão `y = sin(x)` aceita "x" como argumento e devolve o valor do seno de "x" para a variável "y".

3.2 Funções para conversão de tipos

O Python fornece nativamente várias funções que convertem valores entre tipos de dados. A função `int()` aceita qualquer valor e converte-o para um inteiro. Se não conseguir efetuar a conversão, é sinalizada uma situação de erro:

```
>>> int('32')
32
>>> int('Computador')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Computador'
```

A função `int` permite converter números reais para inteiros mas não irá aplicar nenhum arredondamento; simplesmente irá ignorar a parte fracionária:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

A função `float` converte inteiros e mesmo *strings* em números reais:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, a função `str` converte o argumento para uma *string*:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.3 Funções matemáticas

O Python possui módulos que implementam a maioria das funções matemáticas normalmente utilizadas. Um **módulo** é um ficheiro que contém uma coleção de funções de alguma forma relacionadas entre si. Neste caso, o módulo matemático possui diversas funções que implementam operações matemáticas comuns.

Antes que seja possível utilizar um módulo, é necessário que este seja **importado** para o programa:

```
>>> import math
```

Esta declaração cria um **objeto do tipo módulo** chamado ***m*ath**. Se tentar imprimir o objeto, será possível obter alguma informação sobre ele:

```
>>> print(math)
<module 'math' (built-in)>
```

Como foi dito, o módulo contém funções e variáveis dentro de si. É possível aceder a essas funções (ou variáveis), especificando o nome do módulo e da função, separando estes por um ponto (“.”). A este formato é dado o nome de **notação do ponto**.

```
>>> racio = potencia_sinal / potencia_ruido
>>> decibeis = 10 * math.log10(racio)

>>> radianos = 0.7
>>> altura = math.sin(radianos)
```

O primeiro exemplo utiliza `log10` para calcular o rácio entre o sinal e o ruído em decibéis (assumindo que os valores das variáveis `potencia_sinal` e `potencia_ruido` foram definidas anteriormente). O módulo *math* também possui a função `log` que calcula logaritmos utilizando a base `e`.

O segundo exemplo obtém o seno da variável `radianos`. O nome da variável é uma pista que a função `sin` e outras funções trigonométricas (`cos`, `tan`, etc.) consideram que o argumento se encontra em radianos. Para converter radianos em graus, divide-se por 360 e multiplica-se por 2π:

```
>>> graus = 45
>>> radianos = graus / 360.0 * 2 * math.pi
>>> math.sin(radianos)
0.7071067811865475
```

A expressão `math.pi` obtém a variável `pi` do módulo `math`. O seu valor não é o valor completo de `pi` mas sim uma aproximação com cerca de 15 casas decimais, o que é suficiente para a maioria dos cálculos.

Se ainda se lembrar do que aprendeu de trigonometria, pode verificar o valor do seno comparando-o à raiz quadrada de dois dividida por dois:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

3.4 Composição

Até agora olhou-se para os elementos de um programa — variáveis, expressões e declarações — de forma isolada, sem foco na combinação destes elementos.

Uma das funcionalidades mais úteis das linguagens de programação é a sua habilidade em permitir ao programador pegar em pequenos blocos de código e **compô-los**. Por exemplo, o argumento de uma função pode ser qualquer tipo de expressão, incluindo operadores aritméticos:

```
>>> graus = 45
>>> x = math.sin(graus / 360.0 * 2 * math.pi)
```

Pode mesmo utilizar invocações de funções como argumento de outra função:

```
>>> x = math.exp(math.log(x+1))
```

Em qualquer sítio que seja possível colocar um valor, é possível colocar uma expressão arbitrária, por mais complexa que seja, com uma exceção: o lado esquerdo de uma declaração de atribuição tem de corresponder ao nome de uma variável. Qualquer outra expressão colocada do lado esquerdo resulta num erro de sintaxe (existem algumas exceções que serão descritas mais tarde).

```
>>> minutos = horas * 60                    # correto
>>> horas * 60 = minutos                    # errado!
  File "<stdin>", line 1
SyntaxError: can't assign to operator
```

3.5 Criar funções novas

Até agora foram sempre utilizadas funções já existentes na linguagem Python, ou num módulo interno. No entanto, também é possível adicionar funções novas. Uma **definição de função** especifica o nome da função e a sequência de declarações que são executadas quando a função é invocada.

Um exemplo:

```
def imprime_letra():
    print("Eu vi um sapo")
    print("A encher o papo")
```

À palavra `def` dá-se o nome de **palavra reservada** (ou palavra chave), que indica que esta declaração define uma função. O nome da função é `imprime_letra`. As regras para os nomes das funções são as mesmas que para as variáveis: letras, números e mesmo alguma pontuação são permitidas. Não é possível utilizar uma palavra reservada como nome de uma função. O nome da função deve indicar o que a função efetua de forma clara e não devem ser dados os mesmo nomes a variáveis e funções. Imagine o que seria se a função `math.exp(x)` afinal não calculasse e^x, ou se a função se chamasse apenas `math.e` (colide com a variável `math.e`).

Como a função `imprime_letra` não possui nada entre os parênteses, ela não admite qualquer argumento.

À primeira linha da definição da função é dado o nome de **cabeçalho**, enquanto ao resto é dado o nome de **corpo**. É obrigatório que o cabeçalho seja terminado com o sinal de dois pontos ":", e o corpo tem de ser indentado. É esta indentação que define que um conjunto de declarações pertence à função que se está a definir. Por convenção, a indentação é de quatro espaços (ver Seção **??**). O corpo de uma função pode conter um qualquer número de declarações.

As *strings* na declaração `print` estão rodeadas por aspas. A regra de boas práticas é a utilização de aspas quando a *string* tem mais de uma palavra e plicas quando se trata de apenas uma palavra.

Se introduzir a definição desta função no modo interativo, o interpretador irá imprimir reticências (...) a indicar que a definição da função não se encontra completa.

```
>>> def imprime_letra():
...     print("Eu vi um sapo")
...     print("A encher o papo")
...
```

Para terminar a definição da função é apenas necessário introduzir uma linha em branco. Quando se define uma função num *script* isto não é necessário. À medida que os nossos exemplos se tornam mais complexos, iremos recorrer maior numero de vezes à escrita de código num ficheiro próprio com extensão `.py`. Relembramos que pode utilizar um qualquer editor de texto simples, e que pode executar o seu código através da linha de comandos:

```
$ python3 ficheiro_de_codigo.py
Eu vi um sapo
A encher o papo
```

A criação de uma função cria automaticamente uma variável com o mesmo nome.

```
>>> print(imprime_letra)
<function imprime_letra at 0xb7e99e9c>
>>> type(imprime_letra)
<class 'function'>
```

O valor da variável `imprime_letra` é um **objeto de uma função**, que possui o tipo `'function'`.

A sintaxe para invocar a nova função criada é a mesma utilizada para as funções internas à linguagem Python:

```
>>> imprime_letra()
Eu vi um sapo
A encher o papo
```

Uma função que esteja definida pode ser invocada dentro de outras funções. Por exemplo, para imprimir a letra da música duas vezes, poderia escrever:

```
def repete_letra():
    imprime_letra()
    imprime_letra()
```

De seguida pode invocar `repete_letra`:

```
>>> repete_letra()
Eu vi um sapo
A encher o papo
Eu vi um sapo
A encher o papo
```

3.6 Definição e utilização

Juntando os fragmentos de código da secção anterior, o programa completo fica com o seguinte aspeto:

```
def imprime_letra():
    print("Eu vi um sapo")
    print("A encher o papo")

def repete_letra():
    imprime_letra()
    imprime_letra()

repete_letra()
```

Este programa contém duas definições de funções: `imprime_letra` e `repete_letra`. Quando o interpretador processa o ficheiro, as definições de funções são interpretadas normalmente, no entanto, o seu efeito é criar representações de funções. As instruções dentro de cada função não são executadas até que a função seja chamada pelo seu nome, e a definição de uma função também não gera qualquer resultado para o ecrã.

Como seria esperado, é necessário definir as funções antes que possam ser invocadas (executadas). Por outras palavras, a definição da função tem de ser interpretada antes que seja invocada pela primeira vez.

Exercício 3.1. *Mova a última linha do programa para o topo, de forma a que a invocação da função ocorra antes da definição. Execute o programa e interprete a mensagem que lhe é apresentada.*

Exercício 3.2. *Mova a invocação da função de novo para o fundo do programa. Mova a definição da função* `imprime_letra` *para depois da definição da função* `repete_letra`*. O que acontece quando se executa o programa?*

3.7 Fluxo de execução

De forma a garantir que uma função é definida antes da sua primeira utilização, é necessário que se saiba qual a ordem pela qual as instruções são executadas. A isto dá-se o nome de **fluxo de execução**.

A execução tem início sempre na primeira instrução de um programa. As instruções são executadas, uma de cada vez, desde o topo até ao final.

A definição de funções não altera o fluxo de execução do programa, mas as instruções dentro de uma função não são executadas na sua definição, apenas quando a função é invocada.

Uma invocação de uma função é como um desvio no fluxo de execução. Em vez de se executar a próxima instrução, o fluxo de execução salta para o corpo da função, executa todas as instruções que a compõem, voltando depois para o ponto de execução anterior.

Isto parece bastante simples, até ao momento que recorde que uma função pode invocar outra. Quando no meio de uma função, o interpretador de Python poderá ter de executar instruções noutra função. Mas quando se encontra nesta nova função, o interpretador poderá ter de executar ainda outra função!

Felizmente, a linguagem Python consegue lidar com este aspeto de uma forma bastante simples: sempre que uma função termina de ser executada, a execução retoma no ponto da função pai que a chamou. Quando se atinge o final do programa, a execução é terminada.

Qual o motivo desta explicação com muitas voltas? Quando se lê um programa, não é suposto que este seja lido de cima para baixo. Faz bastante mais sentido ler de acordo com o fluxo de execução.

3.8 Parâmetros e argumentos

Algumas funções nativas da linguagem Python que vimos necessitam de argumentos. Por exemplo, quando invoca `math.sin` é fornecido um número como argumento. Algumas funções possuem dois argumentos: `math.pow` necessita de dois, a base e o expoente.

Dentro das funções os argumentos são atribuídos a variáveis chamadas de **parâmetros**. Veja o código seguinte, que consiste numa função definida pelo utilizador que aceita um argumento:

```
def imprime_duplicado(nome):
    print(nome)
    print(nome)
```

Esta função necessita de um argumento que é atribuído a um parâmetro chamado `nome`. Quando a função é invocada, ela imprime duas vezes para o ecrã o valor que o parâmetro possui (qualquer que ele seja).

Esta função imprime em duplicado qualquer coisa que seja fornecido como argumento.

```
>>> imprime_duplicado('Programar')
Programar
Programar
>>> imprime_duplicado(17)
17
17
>>> imprime_duplicado(math.pi)
3.14159265359
3.14159265359
```

As mesmas regras de composição que se aplicam às funções nativas da linguagem Python, também se aplicam a funções criadas por si. Desta forma, pode ser utilizada qualquer expressão como argumento da função `imprime_duplicado`.

```
>>> imprime_duplicado('Programar' * 4)
Programar Programar Programar Programar
Programar Programar Programar Programar
>>> imprime_duplicado( math.cos(math.pi) )
-1.0
-1.0
```

O argumento fornecido a uma função é avaliado (calculado o seu valor), antes que a função seja invocada, sendo passado como argumento o resultado final. Desta forma, `'Programar'*4` e `math.cosmath.pi` não são avaliados duas vezes, mas sim apenas uma vez.

Também é possível usarem-se variáveis como argumentos de funções:

```
>>> frase = 'quinje a jéro!!'
>>> imprime_duplicado( frase )
quinje a jéro!!
quinje a jéro!!
```

O nome da variável que é passada como argumento (`frase`) não tem nada a ver com o nome do parâmetro (`nome`). Não interessa qual o nome dado à variável passada como argumento (na função que invoca); dentro da função `imprime_duplicado`, todos os valores são referidos utilizando `nome`.

3.9 Variáveis e parâmetros são locais

Quando se cria uma variável dentro de uma função, diz-se que ela é **local**, o que significa que só existe dentro da função. Por exemplo:

```
def imprime_nome(primeiro, ultimo):
    pessoa = primeiro + ' ' + ultimo
    imprime_duplicado( pessoa )
```

Esta função aceita dois argumentos, concatena os argumentos juntamente com um espaço, e imprime o resultado duas vezes. O resultado da concatenação é armazenado na variável `pessoa`. A função pode ser utilizada da seguinte forma:

```
>>> nome1 = 'Zacarias'
>>> nome2 = 'Fonseca'
>>> imprime_nome(nome1, nome2)
Zacarias Fonseca
Zacarias Fonseca
```

Quando a função `imprime_nome` termina a sua execução, a variável `pessoa` é destruída. Se tentar imprimir o valor da variável, certamente encontrará um erro:

```
>>> print(pessoa)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pessoa' is not defined
```

3.10 Diagramas da *Stack*

Para manter a lista das variáveis que são utilizadas e onde elas se encontram, por vezes é útil desenhar um **diagrama da pilha** de funções (inglês: ***s*tack diagram**). Tal como os diagramas de estado, os diagramas da *stack* de funções, mostram o valor de cada variável, mas também mostram a que função cada variável pertence.

Nestes diagramas, cada variável é representada por um **quadro** (inglês: *frame*). Uma *frame* é uma caixa, com o nome da função ao lado da caixa, e as variáveis e parâmetros da função dentro da caixa. O diagrama da **stack** do exemplo anterior é apresentado na Figura 3.1.

Os quadros, ou **frames** estão organizados numa pilha, indicando a ordem pela qual as funções foram invocadas e quem invocou quem. Neste exemplo, a função `imprime_duplicado` foi invocada pela função `imprime_nome`, e a função `imprime_nome` foi invocado pela função `__main__`. Esta função é especial pois existe em todos os programas, sempre no início da pilha de funções. Qualquer instrução ou variável que seja declarada fora de uma função, na realidade pertence à função `__main__`.

Cada parâmetro da função `imprime_nome` partilha o seu valor com o argumento correspondente (posicionalmente). Portanto, `primeiro` possui o mesmo valor que `nome1`, `segundo` possui o mesmo valor que `nome2`, e `frase` possui o mesmo valor que `pessoa`.

Se existir um erro durante a invocação de uma função, o interpretador Python irá imprimir o nome da função, o nome da função que a invocou, o nome da função que invocou esta última e assim sucivamente, até à função `__main__`.

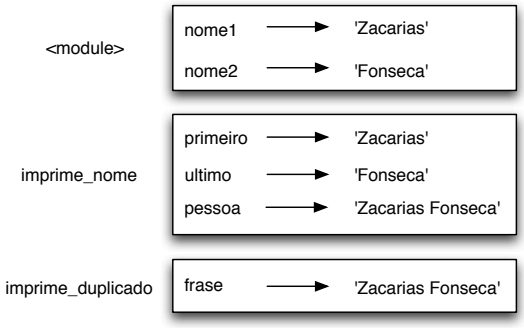

Figura 3.1: Diagrama da *Stack*

Por exemplo, se tentar aceder à variável `pessoa` dentro da função `imprime_duplicado`, o resultado será um erro do tipo `NameError`:

```
Traceback (most recent call last):
  File "test.py", line 13, in <module>
    imprime_nome(nome1, nome2)
  File "test.py", line 9, in imprime_nome
    imprime_duplicado( pessoa )
  File "test.py", line 4, in imprime_duplicado
    print pessoa
NameError: global name 'pessoa' is not defined
```

A esta lista de funções é dado o nome de **conteúdo da pilha** (inglês: ***s*tacktrace** ou ***t*raceback**). Ela indica em que ficheiro aconteceu o erro (`test.py`), em que linha (4), e que funções foram invocadas até se atingir o erro (`imprime_duplicado` e `imprime_nome`). Também apresenta qual o erro em concreto (`NameError: global name 'pessoa' is not defined`).

Como se pode verificar, a ordem das funções no *stacktrace* é a mesma que encontramos no diagrama da *stack*. A função que foi invocada em último lugar é a indicada no fundo da lista.

3.11 Funções e Procedimentos

Algumas funções que utilizámos, tal como as funções matemáticas, devolvem um resultado; e continuaremos a chamá-las de **funções**. Outras, como por exemplo a `imprime_nome` realizam várias ações mas não devolvem qualquer valor. Na realidade é comum chamá-las de **procedimentos**.

Quando se invoca uma função, pretendemos quase sempre fazer algo com o resultado; por exemplo, utilizar o resultado como parte de uma expressão:

```
x = math.cos(radianos)
dourado = (math.sqrt(5) + 1) / 2
```

Quando utiliza o modo interativo e executa uma função, o interpretador de Python irá apresentar o resultado:

```
>>> math.sqrt(5)
2.2360679774997898
```

No modo de *script*, a invocação de uma função não apresenta qualquer resultado e este é perdido para sempre.

```
math.sqrt(5)
```

Este programa calcula o valor da raiz quadrada de 5, mas como o valor não é guardado, ou apresentado, o programa não é muito útil.

O exemplo seguinte armazena o resultado numa variável chamada `a`, que poderia ser utilizada mais tarde:

```
a = math.sqrt(5)
```

Os procedimentos podem realizar cálculos, apresentar informação para o ecrã, ou fazer qualquer outra operação, mas não devolvem qualquer valor. Se tentar atribuir o resultado de um procedimento a uma variável, esta variável irá ser de um tipo especial: `None`.

```
>>> resultado = imprime_duplicado('Bingo')
Bingo
Bingo
>>> print(resultado)
None
```

O valor `None` não é uma *string*. `'None'` é um valor especial de um tipo interno próprio. Pode utilizar a função `type` para inspecionar o tipo de qualquer coisa, tal como a variável `resultado`:

```
>>> resultado = imprime_duplicado('Bingo')
Bingo
Bingo
>>> print(type(resultado))
<class 'NoneType'>
```

Todas as funções que foram definidas até agora, na realidade são procedimentos. Iremos definir funções nos próximos capítulos.

3.12 Porque existem funções?

Pode não parecer claro porque se deve dar ao trabalho de dividir um programa em funções. No entanto garantimos que este conceito é vital para a computação. Existem várias razões a favor das funções:

- Definir uma nova função permite criar um grupo de instruções, o que torna o programa mais simples de analisar e depurar.

- As funções podem tornar um programa mais pequeno pois eliminam a necessidade de repetir instruções. Considere que cria uma função que calcula a área de um quadrado a partir da altura e largura. Sempre que quiser calcular a área, basta invocar a função com a altura e largura. Se quiser modificar o algoritmo, também só precisa de o fazer num unico sítio (dentro da função).
- Dividir um programa em funções permite que se depure o programa por blocos, o que facilita a localização e correção de eventuais problemas. Depois, os blocos individuais podem ser combinados de forma a criar um programa final.
- As funções são por vezes úteis para vários programas. Depois de criada uma função e devidamente validada, pode utilizá-la noutros programas com a certeza que irá funcionar corretamente.

3.13 Importação com `from`

Na Secção 3.3 testemunhou que a linguagem Python permite a importação de módulos externos por um programa. Estes módulos contém funções que ficam depois disponíveis para utilização nas instruções do programa atual. O exemplo que vimos foi:

```
>>> import math
>>> print(math)
<module 'math' (built-in)>
>>> print(math.pi)
3.14159265359
```

Quando se importa o módulo `math`, é criado um objeto chamado `math`. Este objeto possui constantes como `pi` ou `e` e funções como `sin` ou `exp`. Para aceder a estes elementos é necessário utilizar o módulo (exemplo: `math.e`), pois o acesso direto não é possível, resultando num erro:

```
>>> import math
>>> print(pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
```

Existe um método de importação alternativo que permite importar não o módulo mas sim elementos individuais do módulo, tal como o `pi`:

```
>>> from math import pi
```

Este método de importação pode ser descrito como: do módulo `math` importa a variável (ou função) `pi`. Passa a ser possível aceder ao elemento de forma direta, sem ser necessário recorrer à notação de ponto:

```
>>> print(pi)
3.14159265359
```

Em alternativa pode importar todos os elementos do módulo se utilizar o símbolo asterisco (`*`):

```
>>> from math import *
>>> cos(pi)
-1.0
```

A vantagem de se importar tudo é que torna o código produzido mais conciso. Deixa de ser necessário utilizar a notação de ponto (`math.pi`) para aceder a cada elemento. A desvantagem, que pode ser grave quando se importam muitos módulos, é que cria conflitos entre os nomes definidos entre os diferentes módulos, ou as variáveis definidas no próprio programa.

O código seguinte exemplifica o que acontece quando uma constante importada de um módulo colide com uma variável do programa:

```
>>> e = 1
>>> from math import *
>>> print e
2.71828182846
>>> e = 0
>>> print e
0
```

Como apresentado, a sobreposição não resulta num erro, apenas numa alteração dos valores das variáveis (ou na definição de funções), o que pode levar a erros de execução ou semânticos.

3.14 Depuração

Se estiver a utilizar um editor de texto para escrever os seus programas, é bastante provável que já se tenha deparado com problemas relacionados com a indentação. Em especial no número de espaços ou tabulações, ou a troca entre espaços e tabulações. A melhor maneira para evitar estes problemas é utilizar apenas espaços (não usar tabulações). Como tal pode ser entediante, alguns editores que foram programados para lidar com a linguagem Python já evitam estes problemas. Reconhecem a criação de blocos e indentam as linhas automaticamente, ou permitem que ao introduzir uma tabulação (uma tecla pressionada) sejam na realidade introduzidos vários espaços (normalmente 4). No entanto, muitos editores não o fazem, em especial os editores genéricos para editar texto (exemplos: Bloco de Notas, TextEdit, gedit, etc.).

As tabulações e os espaços são invisíveis, o que torna este problema difícil de depurar. Recomenda-se que encontre um editor de texto apropriado ao desenvolvimento de código, em especial na linguagem Python. Existem muitos, não deve ser difícil.

Recomenda-se que não se esqueça de guardar o programa antes de o executar. Alguns editores também fazem isto de forma automática, mas muitos não. É difícil depurar um programa que não foi guardado, pois as instruções executadas serão diferentes das que analisa quando procura a causa de um problema. Aliás, recomenda-se que guarde e execute o programa **frequentemente**, depurando de forma isolada à medida que a implementação se torna mais complexa. É muito mais fácil encontrar problemas em programas pequenos, ou em programas onde se sabe que as instruções foram adicionada e/ou alteradas recentemente. Provavelmente um novo erro será devido a uma dessa instruções.

A depuração é um processo iterativo e que deve caminhar na direção de um programa correto. Executar o mesmo programa de forma repetitiva sem que se corrijam os problemas irá levar que este processo demore muito tempo!

Tenha a certeza que o código que está a ver no editor é o código que está a executar. É frequente adicionarem-se instruções que permitem verificar qual o fluxo de execução real e qual os valores das variáveis. Isto é tão simples como adicionar instruções do tipo `print("estou aqui")`. Também pode ser utilizado para verificar se o programa atual é o que estamos a ver. Adicione `print('aqui')` ao início do ficheiro e execute o programa. Se não aparecer esta mensagem é porque não está a executar o programa correto!

Capítulo 4

Caso de Estudo: Desenho de Interfaces

Este capítulo tem por base o código e pacote Python desenvolvido pelo autor Allen B. Downey, e disponível em `http://thinkpython.com/code/polygon.py` e `http://thinkpython.com/swampy`

4.1 TurtleWorld

Neste capítulo será feito um uso extensivo de um pacote python denominado `swampy`. O `swampy` poderá ser obtido a partir do seu site oficial, ou de forma mais simples, utilizando o gestor de pacotes `pip`.

```
pip install swampy
```

Um **pacote** é uma colecção de módulos, sendo que um dos módulos do `swampy` é o `TurtleWorld`, que disponibiliza um conjunto de funções que permitem desenhar linhas conduzindo uma tartaruga pelo ecrã.

Se o swampy estiver instalado como um pacote no seu sistema, pode importar o `TurtleWorld` da seguinte forma:

```
from swampy.TurtleWorld import *
```

Caso tenha obtado por fazer o download do pacote e uma instalação manual, pode trabalhar no directório do pacote ou acrescentar o mesmo à configuração de directórios em que o Python procura por pacotes (*search path*). Pode depois importar o `TurtleWorld` da seguinte forma:

```
from TurtleWorld import *
```

A maneira como deve ser instalado o pacote e a alteração à configuração do *search path* do Python dependem de sistema para sistema, pelo que ficam fora do objectivo deste capitulo. No entanto, pode encontrar mais informação no site do pacote.

Crie um ficheiro com o nome poligono.py e escreva o seguinte código:

```
from swampy.TurtleWorld import *

world = TurtleWorld()
bob = Turtle()
print(bob)

wait_for_user()
```

A primeira linha importa tudo que o modulo `TurtleWorld` disponibiliza no pacote `swampy`

As linhas seguintes criam o "mundo da tartaruga" atribuído a `world` e uma tartaruga atribuída a `bob`. A instrução `print` da tartaruga, resulta em algo como:

```
<TurtleWorld.Turtle instance at 0xb7bfbf4c>
```

Isto significa que `bob` se refere a uma **instancia** de `Turtle` (é uma tartaruga) tal como definido no módulo `TurtleWorld`. Neste contexto, "instancia" significa ser membro de um conjunto; esta tartaruga é parte de um conjunto de tartarugas (`Turtle`) possíveis.

`wait_for_user` indica ao "mundo da tartaruga" que deve aguardar que o utilizador faça qualquer coisa, muito embora não haja nada para fazer exceptuando fechar a janela.

O módulo `TurtleWorld` disponibiliza diversas funções que permitem conduzir a tartaruga: `fd` e `bk` para andar para a frente e para trás (***f**orward* e ***b**ack**k**ward*), e `lt` e `rt` para virar à esquerda e direita (***l**eft* ***t**urn* e ***r**ight* ***t**urn*). Cada tartaruga segura ainda uma caneta que pode estar levantada ou poisada na superficie; se a caneta estiver levantada não deixa nenhuma marca, caso contrário, risca a superficie à medida que a tartaruga se move. As funções `pu` e `pd` correspondem a levantada e pousada (***p**en* ***u**p* e ***p**en* ***d**own*).

Para desenhar um angulo recto, acrescente as seguintes linhas ao programa (depois de criar a tartaruga e antes de chamar por `wait_for_user`):

```
fd(bob, 100)
lt(bob)
fd(bob, 100)
```

A primeira linha ordena à tartaruga `bob` que se desloque 100 passos para a frente (*forward*). A segunda linha que bob vire à esquerda (*left turn*).

Quando o programa é executado, deverá ver a tartaruga a mover-se para éste e depois para norte, deixando para trás dois segmentos de recta.

Agora modifique o programa para desenhar um quadrado. Não avance enquanto não conseguir executar esta tarefa!

4.2 Repetição Simples

É provável que tenha escrito algo semelhante a (omite-se o código necessário para criar o "mundo da tartaruga" e que espera resposta do utilizador):

```
fd(bob, 100)
lt(bob)
```

```
fd(bob, 100)
lt(bob)

fd(bob, 100)
lt(bob)

fd(bob, 100)
```

Podemos fazer o mesmo de forma mais resumida utilizando a instrução `for`. Acrescente o seguinte exemplo ao seu programa e execute-o de novo:

```
for i in range(4):
    print("Hello!")
```

Deverá obter o seguinte resultado:

```
Hello!
Hello!
Hello!
Hello!
```

Este é o exemplo mais simples da utilização da instrução `for` que mais tarde iremos aprofundar. Agora deverá ser mais simples re-escrever o programa que desenha um quadrado. Não avance até o conseguir.

Aqui esta uma instrução `for` que desenha um quadrado:

```
for i in range(4):
    fd(bob, 100)
    lt(bob)
```

O sintaxe do `for` é muito semelhante ao usado para definir uma função. Existe um cabeçalho que termina com dois-pontos e um corpo indentado. O corpo pode conter um número indefinido de instruções.

A instrução `for` é muitas vezes apelidada de **ciclo** porque o seu fluxo de execução repete o corpo um número determinado de vezes. Neste caso, o corpo repete quatro vezes.

Esta versão é ligeiramente diferente da versão anterior pois existe uma curva extra no final. Esta curva extra leva um pouco mais de tempo, mas simplifica o código já que permite repetir sempre o mesmo corpo. Esta versão deixa a tartaruga de volta na posição inicial, orientada na direcção inicial.

4.3 Exercícios

Os seguintes exercícios usam o modulo `TurtleWorld`. Para lá de um caracter lúdico, são também pedagógicos, e cada um tem um ensinamento subjacente. Enquanto trabalhar em cada um deles, procurar identificar o ensinamento.

Uma vez que as próximas secções contêm as soluções para estes exercícios, não avance antes de os terminar (ou pelo menos tentar).

1. Escreva uma função chamada `square` que receba um parâmetro `t`, que será uma tartaruga. Deverá usar a tartaruga para desenhar um quadrado.

   Escreva uma chamada à função que passe o `bob` como argumento da função `square`, e execute o programa novamente.

2. Acrescente um novo parâmetro à função `square`, chamado `length`. Altere o corpo de maneira a que o tamanho dos lados seja `length`, e altere a chamada à função para incluir mais um parâmetro. Execute o programa novamente. Teste o seu programa com um conjunto de valores distintos de tamanho.

3. As funções `lt` e `rt` fazem curvas de 90 graus por omissão, mas é possível passar um segundo argumento onde se especifica o número de graus. Por exemplo, `lt(bob, 45)` vira a tartaruga `bob` 45 graus para a esquerda.

   Crie uma nova função chamada `polygon` com base no corpo de `square` e acrescentando um novo parâmetro `n` proceda Às alterações necessárias para poder desenhar um polígono regular de n-lados. Dica: os ângulos internos de um polígono regular de n-lados têm $360/n$ graus.

4. Escreva uma função `circle` que pegue numa tartaruga, `t`, num raio, `r` como parâmetros e desenhe uma aproximação a um circulo invocando a função `polygon` com o tamanho de lado e numero de lados apropriados. Teste o seu programa com um conjunto de valores distintos para `r`.

   Dica: determine a circunferencia do circulo e garanta que `length * n = circunferência`.

   Outra pista: se o `bob` levar muito tempo a desenhar, pode aumentar a velocidade alterando o valor de `bob.delay`, que não é mais que a pausa entre movimentos da tartaruga em segundos. `bob.delay = 0.01` deverá surtir efeito.

5. Crie agora uma versão mais genérica de `circle` chamada `arc` que aceite um parâmetro extra `angle`, que determina que fracção do circulo deverá ser desenhado. `angle` tem por unidade o grau, tal que `angle=360` implique o desenho de um circulo completo.

4.4 Encapsulamento

O primeiro exercício pedia para colocar o código que desenha o quadrado numa função e depois chamar pela mesma passando a tartaruga como parâmetro. Eis uma possível solução:

```
def square(t):
    for i in range(4):
        fd(t, 100)
        lt(t)

square(bob)
```

As instruções interiores, `fd` e `lt` estão indentadas duas vezes para denotar o facto de pertencerem ao corpo do `for` que por sua vez faz parte do corpo da função. A linha seguinte,

`square(bob)`, está encostada à esquerda, pelo que está fora da definição da função e consequentemente do `for`.

Dentro da função, `t` refere-se à mesma tartaruga a qual`bob` se refere, assim `lt(t)` produz o mesmo efeito que `lt(bob)`. Então porque não chamamos o parâmetro de `bob`? A ideia é `t` poder ser qualquer tartaruga, não apenas `bob`, possibilitando que se crie uma nova tartaruga e que a mesma possa ser o argumento de `square`:

```
ray = Turtle()
square(ray)
```

Ao acto de envolver um pedaço de código dentro da definição de uma função chama-se **encapsulamento**. Um dos benefícios do encapsulamento é que associa um nome ao código, conferindo uma propriedade documental. Outra vantagem é que permite a re-utilização de código, permitindo chamar a função múltiplas vezes sem necessitar de copiar o corpo várias vezes.

4.5 Generalização

O próximo passo foi dotar a função do parâmetro `length`:

```
def square(t, length):
    for i in range(4):
        fd(t, length)
        lt(t)

square(bob, 100)
```

Adicionar parâmetros a uma função chama-se **generalização** uma vez que torna a função mais geral: na versão anterior o quadrado tem sempre o mesmo tamanho, agora pode ter qualquer tamanho.

O passo seguinte é também ele uma generalização. Em vez de desenhar apenas quadrados, `polygon` permite desenhar polígonos com um número arbitrário de lados. Aqui fica uma solução:

```
def polygon(t, n, length):
    angle = 360.0 / n
    for i in range(n):
        fd(t, length)
        lt(t, angle)

polygon(bob, 7, 70)
```

Este exemplo desenha um heptágono com o tamanho dos lados a ser 70. Se o numero de argumentos for um pouco mais do que dois ou três, é fácil nos esquecermos dos mesmos e da sua ordem. É válido, e muitas vezes útil, incluir o nome do parâmetro na lista de argumentos:

```
polygon(bob, n=7, length=70)
```

Estes chamam-se de **argumentos chave** já que estes servem de chave para o interpretador (não confundir com as palavras chave do Python como `while` e `def`).

Este sintaxe torna o programa mais legível. Permite ainda nos lembrar como os argumentos e parâmetros funcionam: quando se chama uma função, os argumentos são atribuídos aos parâmetros.

4.6 Desenho de Interfaces

O próximo passo foi escrever `circle`, que recebe como parâmetro o raio `r`. Uma possível solução para este exercício tendo por base um polígono de 50 lados (pentacontágono):

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

A primeira linha calcula a circunferência de um circulo com raio `r` usando a fórmula $2\pi r$. Uma vez que utilizamos `math.pi`, necessitamos de importar o pacote `math`. Por convenção as instruções que importam pacotes são inscritas nas primeiras linhas do *script*.

`n` é o numero de segmentos de recta na aproximação proposta a um circulo, pelo que `length` é o comprimento de cada segmento. Assim `polygon` desenha um pentacontágono que aproxima um circulo de raio `r`.

Uma das limitações desta solução é o facto de `n` ser uma constante, o que significa que em circulos de grandes dimensões, os segmentos de recta serão muito longos, e que para círculos de pequenas dimensões dos segmentos serão muito curtos e levarão muito tempo a desenhar. Uma solução será generalizar a função para receber o argumento `n` como parâmetro. Tal permitirá ao utilizador (quem quer que invoque a função `circle`) um maior controlo, mas a interface será menos limpa.

A **interface** de uma função é o sumário de como a mesma é utilizada: quais são os parâmetros? o que faz a função? E qual é o valor retornado? Uma interface diz-se limpa se for "o mais simples possível mas não simplista. (A. Einstein)"

Neste exemplo, `r` pertence à interface porque especifica o circulo a desenhar. `n` não é tão apropriado porque se intromete nos detalhes de **como** o circulo deve ser implementado.

Em vez de atropelar a interface, é mais apropriado escolher um valor de `n` que tenha por base a circunferência (`circumference`).

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 1
    length = circumference / n
    polygon(t, n, length)
```

Agora que o número de segmentos é (aproximadamente) `circumference/3`, presenciamos um valor suficientemente pequeno para que o aspecto seja bom, grande o suficiente para ser eficiente e apropriado para qualquer tamanho.

4.7 *Refactoring*

Quando `circle` foi escrito, foi possível re-utilizar `polygon` pois um polígono com muitos lados é uma boa aproximação a um circulo. Já `arc` não é tão cooperante, não se pode utilizar `polygon` ou `circle` para desenhar um arco.

Uma alternativa é começar com uma cópia de `polygon` e transformá-la em `arc`. O resultado é algo como:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n

    for i in range(n):
        fd(t, step_length)
        lt(t, step_angle)
```

A segunda parte da função é parecida com `polygon`, mas não é possível re-utilizar `polygon` sem alterar a interface. Seria possível generalizar `polygon` para ter como terceiro parâmetro um angulo, mas `polygon` deixaria ser um nome apropriado! A opção, é chamar a função mais geral de `polyline`:

```
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Agora é possível re-escrever `polygon` e `arc` com base em `polyline`:

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)

def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Finalmente, podemos re-escrever `circle` para utilizar `arc`:

```
def circle(t, r):
    arc(t, r, 360)
```

A este processo (re-organizar o programa para melhorar as interfaces e facilitar a reutilização de código) é dado onome de re-factorização (em inglês: **refactoring**). Neste caso, foi fácil reparar que o código utilizado em `arc` e `polygon` eram similares, pelo que colocámos em evidência ("factored out") `polyline`.

Caso tivesse planeado melhor no inicio, poderia ter escrito `polyline` primeiro e ter evitado a re-factorização, mas o mais comum é não ter conhecimento de causa no inicio do projecto quando desenha todas as interfaces. Só quando iniciamos a escrita do código, é que realmente compreendemos o problema em mãos. A re-factorização não deve ser vista como uma correcção de um erro arquitectural mas sim como um sinal que compreendeu melhor o problema e optimizou a solução.

4.8 Plano de desenvolvimento

Um **plano de desenvolvimento** é uma processo para escrever programas. O processo utilizado neste caso foi o "encapsulamento e generalização." Os passos deste processo são:

1. Começar por escrever um pequeno programa sem definição de qualquer função.
2. Assim que o programa funcionar, encapsular o mesmo dentro de uma função com um nome adequado.
3. Generalizar a função acrescentando parâmetros apropriados.
4. Repetir os passos 1–3 até ter um conjunto de funções a funcionar.
5. Esteja sempre atento a oportunidades de fazer re-factorização. Por exemplo, se encontrar um trecho de código igual em duas funções deverá ser possível colocar esse mesmo código numa nova função a ser invocada pelas já existentes.

Este processo tem algumas desvantagens (mais à frente iremos abordar alternativas) mas pode ser útil quando à partida não sabe em que funções dividir o código. Esta aproximação permite desenhar a solução à medida que a vai construindo.

4.9 Docstring

Uma **docstring** é uma *string* colocada no inicio de uma função que explica a interface ("doc" é um apelido para "documentação"). Aqui fica um exemplo:

```
def polyline(t, n, length, angle):
    """Desenha n segmentos de recta com o tamanho
    length e o angulo angle (em graus) entre segmentos.
    t é uma tartaruga."""
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Esta *docstring* encontra-se entre aspas triplas, também conhecida por *string* multi-linha, que permitem à string se extender por múltiplas linhas.

É muito resumida, mas contém a informação essencial para que a função possa ser compreendida por alguém. Explica de forma concisa o que a função faz (sem entrar em detalhes de como é feito). Explica os efeitos que cada parametro tem no comportamento da função, e de que tipo são (uma vez que não é óbvio).

Escrever este tipo de documentação é uma parte importante do desenha de interfaces. Uma interface bem desenhada deve ser fácil de explicar; Se for dificil explicar uma função, então é sinal que a interface não é a melhor e deve ser melhorada.

4.10 Debugging

Um interface é como um contracto entre a função e quem a invoca. Quem invoca aceita disponibilizar um conjunto de parâmetros em troca a função aceita computar um resultado.

Por exemplo, a função `polyline` precisa de quatro parâmetros: `t` tem que ser uma tartaruga; `n` é o número de segmentos de recta, pelo que tem que ser um inteiro; `length` deverá ser um número positivo; e `angle` necessita ser um número, que se depreende ser em graus.

A estes requisitos chamamos de **pré-condições** uma vez que precisam ser verdadeiros antes de a função poder executar. Por oposição, condições no final da função chamam-se de **pós-condições**. As pós-condições incluem o efeito pretendido da função (como desenhar um segmento de recta) e todos efeitos colaterais (como mover a tartaruga ou modificar o mundo).

As pré-condições são da responsabilidade de quem invoca. Se as violar (não respeitando a documentação) e a função não funcionar correctamente, o erro encontra-se do lado de quem invocou, não é um problema da função.

Capítulo 5

Condicionais e Recursividade

5.1 Divisão Inteiros

A divisão de dois números dá sempre origem a um número do tipo `float`. No entanto existem vários casos de uso em que necessitamos de obter a divisão inteira, correspondente a um quociente e resto.

O **operador divisão** \\ diferencia-se do operador divisão \ por retornar o quociente da divisão inteira. Este operador pode ser utilizado tanto com numeros inteiros como decimais.

O **operador módulo** trabalha sobre inteiros e produz como resultado o resto da divisão do primeiro operando pelo segundo. Em Python, o operador módulo é indicado através do símbolo de percentagem (%). A sintaxe é o mesmo que para os outros operadores:

```
>>> quociente = 7 // 3
>>> print(quociente)
2
>>> resto = 7 % 3
>>> print(resto)
1
```

A divisão de 7 por 3 é 2 com o resto de 1.

O operador módulo é extremamente útil em inúmeras ocasiões. Por exemplo, pode verificar se um numero é divisível por outro (se `x % y` for zero, então `x` é divisível por `y`).

É também possível extrair o dígito mais à direita de um número. Por exemplo, `x % 10` produz o digito mais à direita de `x` (em base 10). De forma semelhante, `x % 100` produz os últimos dois dígitos.

5.2 Expressões boleanas

Uma **expressão boleana** é uma expressão que pode ter apenas dois valores: verdadeiro ou falso. Os seguintes exemplos usam o operador `==`, que compara dois operandos e produz `True` (verdadeiro) se forem iguais e `False` (falso) caso sejam diferentes:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

Os valores `True` e `False` são especiais, pois pertencem ao tipo `bool`; não são *strings*:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> type(5 == 6)
<class 'bool'>
```

O operador == é um dos **operadores relacionais**, sendo os outros:

```
x != y           # x não é igual a y
x > y            # x é maior que y
x < y            # x é menor que y
x >= y           # x é maior ou igual a y
x <= y           # x é menor ou igual a y
```

Muito embora estas operações já lhe sejam familiares, em Python os símbolos são diferentes dos usados na matemática. Um erro comum é utilizar um único símbolo de igualdade (=) em vez de dois (==) para fazer uma comparação. Lembre-se que o = é o operador de atribuição e que == é que é usado para fazer comparações, e que os operadores =< e => não existem.

5.3 Operadores Lógicos

Existem três **operadores lógicos**: `and`, `or`, e `not`. A semântica (significado) destes operadores é semelhante ao significado em Inglês e consequentemente em Português, *e*, *ou* e *não* respectivamente. Por exemplo, `x > 0 and x < 10` é `True` (verdadeiro) apenas se `x` for maior que 0 `and` (e) menor que 10.

`n%2 == 0 or n%3 == 0` é `True` (verdadeiro) se a primeira ou segunda condição for verdadeira, isto é, se o número for divisível por 2 `or` (ou) 3.

Finalmente, o operador `not` permite negar uma expressão, tal que `not (x > y)` é verdade se `x > y` for falso, isto é , se `x` for menor ou igual a `y`.

De forma muito estrita, os operandos de um operador lógico devem eles próprios ser expressões boleanas, mas a linguagem Python não é muito restritiva. Qualquer valor diferente de zero é interpretado como sendo verdadeiro (`True`):

```
>>> 17 and True
True
```

Esta flexibilidade pode ser muito útil, mas também pode dar origem a diversas subtilezas que podem causar alguma confusão. É aconselhado evitar este uso (a menos que tenha a certeza do que está a fazer).

5.4 Execução Condicional

Para se construírem programas úteis, é necessário possuir a capacidade de poder verificar condições e alterar o comportamento do programa de acordo. As **instruções condicionais** fornecem essa capacidade. A instrução mais simples é o `if`, que em português se traduz literalmente para *se*:

```
if x > 0:
    print("x é positivo")
```

A expressão a seguir ao `if` é chamada de **condição**. Se a mesma for verdadeira, então a instrução indentada é executada. Caso contrário, nada acontece. O exemplo anterior pode ser lido da forma: Se `x` for superior a 0, imprime a mensagem `x é positivo`.

As instruções `if` têm a mesma estrutura que as definições de funções: um cabeçalho seguido por um corpo indentado. Estas instruções são chamadas de **instruções compostas**.

Não existe limite ao número de instruções que podem aparecer no corpo, mas tem de existir pelo menos uma instrução. Ocasionalmente, pode ser útil um corpo que não execute nada (normalmente quando ainda não estamos preparados para decidir que instrução devemos chamar). Nestes casos, podemos utilizar a instrução `pass`, que não faz nada.

```
if x < 0:
    pass          # é preciso decidir o que fazer com os números negativos
```

5.5 Execução alternativa

A instrução `if` pode ainda disponibilizar uma **execução alternativa** em que existem duas possibilidades de execução e a condição determina qual a que é executada. O sintaxe assemelha-se a:

```
if x%2 == 0:
    print("x é par")
else:
    print("x é impar")
```

Quando o resto da divisão de `x` por 2 for 0, então `x` é necessariamente um número par e o programa irá imprimir para o ecrã a mensagem correspondente. Caso contrário será impressa a mensagem presente na segunda instrução. Uma vez que a condição só pode ser verdadeira ou falsa, exactamente uma das alternativas terá que ser executada. A cada alternativa damos o nome de **ramo**, correspondendo a uma bifurcação no caminho de execução do programa.

5.6 Condições encadeadas

Por vezes existem mais do que duas possibilidades e é necessário criar mais do que dois ramos. Uma maneira de expressar tal computação é através de **condições encadeadas**:

```
if x < y:
    print("x é menor que y")
elif x > y:
    print("x maior que y")
else:
    print("x e y são iguais")
```

`elif` é uma abreviação de "else if". Mais uma vez, apenas um dos ramos será executado. Não existe limite ao número de instruções `elif`. Se existir um termo `else`, este terá que ser o último, mas poderá não existir.

```
if choice == 'a':
    desenhar_a()
elif choice == 'b':
    desenhar_b()
elif choice == 'c':
    desenhar_c()
```

Cada condição é verificada por ordem. Se a primeira for falsa, a próxima é verificada, e por aí em diante. Se uma delas for verdadeira, o ramo correspondente é executado e a instrução termina. Mesmo que mais do que uma condição seja verdadeira, apenas o primeiro ramo verdadeiro executa.

5.7 Condições Endógenas

Uma condição pode também surgir dentro de uma outra condição. Poderíamos ter escrito a condição tricotómica anterior da seguinte forma:

```
if x == y:
    print("x e y são iguais")
else:
    if x < y:
        print("x é menor que y")
    else:
        print("x é maior que y")
```

A condição exterior tem dois ramos. O primeiro ramo contem uma única instrução. O segundo ramo contem outra instrução `if`, que por sua vez contém dois ramos próprios. Estes dois ramos contêm cada um uma única instrução, mas poderiam também ser instruções condicionais.

Embora a indentação da instrução torne a estrutura aparente, as **condicionais endógenas** tornam-se rapidamente de difícil leitura. Em geral, devem ser evitadas dentro do possível.

Os operadores lógicos permitem muitas das vezes simplificar condições endógenas. Por exemplo, é possível re-escrever o código seguinte utilizando uma única condição:

```
if 0 < x:
    if x < 10:
        print("x é um número com um único digito positivo.")
```

A função `print` é executada apenas se ambas as condições forem verdadeiras, pelo que podemos usar o operador `and`:

```
if 0 < x and x < 10:
        print("x é um número com um único digito positivo.")
```

5.8 Recursividade

É permitida a uma função invocar outra função, é também permitida a uma função invocar-se a si própria. Poderá não ser obvio o propósito de tal caso, mas trata-se de uma das mais importantes coisas que um programa pode fazer. Por exemplo, analise a seguinte função:

```
def contagem_decrescente(n):
    if n <= 0:
        print("Ignição!")
    else:
        print(n)
        contagem_decrescente(n-1)
```

Se `n` for 0 ou negativo, será impressa a palavra "Ignição!". Caso contrário, será impresso o valor da variável `n` seguindo-se a invocação da função `contagem_decrescente` — ela própria — passando o argumento `n-1`.

O que irá acontecer se chamarmos a função desta maneira?

```
>>> contagem_decrescente(3)
```

A execução de `contagem_decrescente` começa com `n=3`, e uma vez que `n` é maior que 0, irá imprimir o valor 3, de seguida chama-se a si própria...

> A execução de `contagem_decrescente` começa com `n=2`, e uma vez que `n` é maior que 0, irá imprimir o valor 2, de seguida chama-se a si própria...
>
> > A execução de `contagem_decrescente` começa com `n=1`, e uma vez que `n` é maior que 0, irá imprimir o valor 3, de seguida chama-se a si própria...
> >
> > > A execução de `contagem_decrescente` começa com `n=0`, e uma vez que `n` não é maior que 0, será impressa a palavra "Ignição!" e a função irá retornar.
> >
> > A `contagem_decrescente` que receber `n=1` retorna.
>
> A `contagem_decrescente` que receber `n=2` retorna.

A `contagem_decrescente` que receber `n=3` retorna.

Até que voltamos a `__main__`. Pelo que o resultado final do programa será algo como:

```
3
2
1
Ignição
```

A uma função que se invoca a si própria chamamos de **recursiva**, ao processo chamamos de **recursividade**.

Outro exemplo, podemos escrever uma função que imprime uma *string* `n` vezes.

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

Se `n<=0` a instrução `return` pára a função. O fluxo de execução retorna imediatamente a quem invocou, e as restantes linhas da função não são executadas.

O resto da função é muito semelhante a `contagem_decrescente`: se `n` for maior que 0, então é impresso `s` e a função invoca-se a si própria para imprimir `s` $n-1$ vezes mais. Assim o numero de linhas na saída é $1+(n-1)$, o que somado dá n.

Para exemplos simples como estes, é provavelmente mais fácil utilizar um ciclo `for`. Mas veremos mais tarde exemplos que são difíceis de escrever utilizando o ciclo `for` mas fáceis utilizando a recursividade, pelo que é adequada a introdução do conceito nesta altura.

5.9 Diagramas de pilha para funções recursivas

Na secção 3.10, fez-se de um diagrama de pilha para representar o estado do program durante a invocação de uma função. O mesmo tipo de diagrama pode ser utilizado para ajudar na interpretação de uma função recursiva.

Sempre que uma função é chamada, o Python cria um novo quadro para a função, que contém as variáveis locais e parâmetros da função. No caso das funções recursivas, podem existir vários quadros na pilha em simultâneo.

A Figura 5.1 mostra um diagrama de pilha para a função `contagem_decrescente` chamada com `n = 3`.

No topo da pilha encontramos como habitualmente o `__main__`. Este está vazio pois não criámos qualquer variável no seu corpo nem foram passados quaisqueres argumentos.

Os quatro quadros `contagem_decrescente` teem valores diferentes para o parâmetro `n`. Ao fundo da pilha, onde `n=0`, chamamos de **caso base**. Não faz qualquer chamada recursiva, pelo que não existem mais quadros.

5.10 Recursão infinita

Se a recursão nunca alcançar um caso base, as invocações recursivas vão se repetir para sempre, e o programa nunca terminará. A esta situação dá-se o nome de **recursão infinita**, e de forma geral é um erro. Eis um exemplo de um programa mínimo com uma recursão infinita:

```
def recursiva():
    recursiva()
```

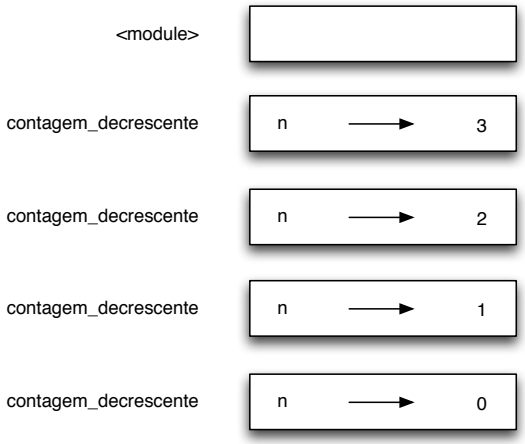

Figura 5.1: Diagrama de *S*tack

Tal como na maioria dos ambientes de programação, um programa com uma recursão infinita não fica a correr para sempre. O Python reporta uma mensagem de erro quando a profundidade de recursão atinge um limite:

```
Traceback (most recent call last):
  File "recursiva.py", line 4, in <module>
    rec()
  File "recursiva.py", line 2, in rec
    rec()
  File "recursiva.py", line 2, in rec
    rec()
  File "recursiva.py", line 2, in rec
    rec()
  [Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
```

O *traceback* é um pouco maior do que o presenciámos no capitulo anterior. Quando um erro ocorre, existem 1000 quadros recursivos na pilha! No entanto o Python sumariza essa extensão para não nos perdermos na analise do erro.

5.11 Entrada de dados pelo teclado

Os programas feitos até ao momento são um pouco rudimentares no sentido que não aceitam a entrada de dados pelo utilizador. Apenas executam a mesma operação vezes sem conta.

O Python 3 disponibiliza uma função embutida que permite a entrada de dados pelo teclado, chama-se `input`. Quando esta função é chamada, o programa pára e espera que o utilizador insira qualquer coisa. Quando o utilizador pressiona o Return ou Enter, a execução do programa contínua e `input` retorna o que o utilizador inseriu na forma de uma *string*.

```
>>> texto = input()
O que é que eu fiz?
>>> print(texto)
O que é que eu fiz?
```

Antes de receber dados do utilizador é uma boa ideia imprimir qualquer coisa que indique ao utilizador que deve escrever. `input` pode receber essa interjeição através de um argumento:

```
>>> horas = input("Que horas é isto?\n")
Que horas é isto?
É tarde. Está na hora de actualizar as notícias!
>>> print(horas)
É tarde. Está na hora de actualizar as notícias!
```

A sequência `\n` no final da interjeição representa uma nova linha (*newline*, que é um caracter especial que provoca uma quebra de linha. É essa a razão para que o texto inserido pelo utilizador surja na linha seguinte.

Se pretender que o utilizador insira um numero inteiro, então pode tentar converter o valor retornado para `int`:

```
>>> prompt = "Que velocidade atinge uma andorinha?\n"
>>> speed = input(prompt)
Que velocidade atinge uma andorinha?
17
>>> int(speed)
17
```

Mas se o utilizador escrever algo que não seja uma *string* de dígitos, terá um erro:

```
>>> speed = input(prompt)
Que velocidade atinge uma andorinha?
Qual delas, a Africana ou Europeia?
>>> int(speed)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: ''
```

Num próximo capitulo iremos analisar como tratar este tipo de erros.

5.12 Depuração

O *traceback* do Python aparece sempre que um erro acontece e apresenta muita informação. A quantidade de informação é tão grande que muitas das vezes se torna difícil a sua leitura e compreensão, especialmente sempre que existem muitos quadros na pilha. As partes mais úteis são no entanto:

- Qual é o tipo de erro, e

- Onde é que o mesmo se deu

Os erros de sintaxe são normalmente fáceis de detectar, mas alguns são esquivos. Os erros de indentação são especialmente difíceis pois os espaços e tabs são invisíveis e estamos habituados a os ignorar.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

Neste exemplo, o problema é que a segunda linha está indentada com um espaço. Mas a mensagem de erro aponta para um problema com o `y`, o que é enganador. Geralmente, a mensagem de erro aponta para onde o erro foi encontrado, mas o verdadeiro erro pode ter ocorrido antes no código, muitas vezes na linha anterior.

O mesmo se aplica aos erros de *runtime*.

Imaginemos que pretendemos calcular o valor da relação sinal-ruido (SNR) em decibeis. A formula é $SNR_{db} = 10\log_{10}(P_{sinal}/P_{ruido})$. Em Python, pode escrever algo como:

```
import math
potencia_sinal = 9
potencia_ruido = 10
racio = potencia_sinal//potencia_ruido
decibeis = 10 * math.log10(racio)
print(decibeis)
```

Mas quando executa em Python2, surge a seguinte mensagem de erro.

```
Traceback (most recent call last):
  File "snr.py", line 5, in <module>
    decibeis = 10 * math.log10(racio)
ValueError: math domain error
```

A mensagem de erro aponta para a linha 5, mas não encontra nada de errado nessa linha. Para descobrir o verdadeiro erro, pode ser util imprimir o valor de `racio`, que aparenta ser 0. O problema está na linha 4, uma vez que a divisão inteira tem por quociente o valor 0. A solução passa por utilizar a divisão normal `/`.

Em geral, a mensagem de erro diz-nos onde o erro foi descoberto mas raramente onde o mesmo foi gerado.

Capítulo 6

Funções Produtivas

6.1 Retornar Valores

Algumas das funções embutidas que já utilizou, tais como as funções matemáticas, produzem resultados. A invocação de uma função gera um valor, que normalmente é atribuído a uma variável ou utilizado como parte de uma expressão.

```
e = math.exp(1.0)
altura = raio * math.sin(radianos)
```

Todas as funções que escreveu até este momento são *void* (vazias); imprimem qualquer coisa ou movimentam uma tartaruga, mas o valor retornado é `None`.

Neste capitulo, irá (finalmente) escrever funções produtivas. O primeiro exemplo é o calculo da área de uma circunferência dado o seu raio através da função `area`:

```
def area(raio):
    temp = math.pi * raio**2
    return temp
```

Anteriormente já se deparou com a instrução `return`, mas numa função produtiva a instrução `return` inclui uma expressão. Esta instrução pode ser interpretado como "retornar imediatamente desta função e utilizar a seguinte expressão como o valor retornado". A expressão utilizada pode ser mais ou menos complicada, pelo que pode escrever de forma mais concisa a função:

```
def area(raio):
    return math.pi * raio**2
```

Por outro lado, o uso de **variáveis temporárias** como `temp` facilitam em muitos casos a depuração.

É por vezes útil ter múltiplas instruções `return` numa dada função, tipicamente uma por ramo de uma condicional:

```
def valor_absoluto(x):
    if x < 0:
        return -x
    else:
        return x
```

Uma vez que as instruções `return` se encontram em ramos alternativos da condicional, apenas uma será executada.

Assim que uma instrução `return` é executada, a função termina sem executar qualquer instrução subsequente. Todo o código que apareça a seguir a uma instrução `return`, ou noutro local que o fluxo de execução não possa alcançar, tem o nome de **código morto**.

Numa função produtiva, é essencial garantir que todos os caminhos possíveis atingam uma instrução `return`. Por exemplo:

```
def valor_absoluto(x):
    if x < 0:
        return -x
    if x > 0:
return x
```

A função está errada uma vez que para `x` igual a 0, nenhuma das condições é verdadeira, e a função termina sem alcançar uma instrução `return`. Se o fluxo de execução atingir o fim da função, o valor retornado será `None`, que não corresponde ao valor absoluto de 0.

```
>>> print valor_absoluto(0)
None
```

A propósito, o Python disponibiliza uma função embutida denominada `abs` que calcula o valor absoluto de um valor.

6.2 Desenvolvimento Incremental

À medida que desenvolve função maiores, por certo irá despender mais tempo na sua depuração.

Para poder lidar com programas cada vez mais complexos, pode experimentar utilizar um processo denominado **Desenvolvimento Incremental**. O objectivo do desenvolvimento incremental é o de evitar processos longos de depuração através da implementação e teste de pequenos pedaços de código que depois de juntos formam o programa.

Por exemplo, suponha que pretende determinar a distância entre dois pontos, cujas as coordenadas são (x_1, y_1) e (x_2, y_2). Pelo teorema de Pitágoras, a distância pode ser calculada:

$$\text{distncia} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

O primeiro passo é pensar o que deve ser uma função em Python para calcular a distância. Por outras palavras, quais são os dados (parâmetros) e qual é o resultado (valor retornado)?

Neste caso, os dados são dois pontos, que pode representar utilizando quatro números. O valor retornado é a distância, que é um valor com virgula flutuante.

Pode então começar por escrever o esqueleto da função:

```
def distância(x1, y1, x2, y2):
    return 0.0
```

Obviamente, esta versão não calcula distâncias, retorna sempre zero. No entanto, é sintacticamente correcta, e executa, o que significa que pode ser testada antes de a tornar mais complicada.

Para testar a nova função, invoque a mesma utilizando argumentos de teste:

```
>>> distância(1, 2, 4, 6)
0.0
```

A escolha destes valores prendeu-se ao facto de a distância horizontal ser de 3 e a vertical de 5; desta forma, o resultado é 5 (a hipotenusa de um triângulo 3-4-5). Quando testamos uma função, é importante conhecer à priori o resultado correcto.

Neste momento temos confirmado que a função é sintacticamente correcta, e pode começar a acrescentar código ao corpo da função. O próximo passo passa por determinar a diferença $x_2 - x_1$ e $y_2 - y_1$. A próxima versão armazena estes valores em variáveis temporárias e imprime os mesmos.

```
def distância(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print 'dx é', dx
    print 'dy é', dy
    return 0.0
```

Se a função estiver a funcionar, deverá imprimir `'dx é 3'` e `'dy é 4'`. Se tal acontecer, sabe que a função está a receber os argumentos correctos e a calcular correctamente a primeira parte do problema. Caso contrário, são poucas as linhas onde se pode ter enganado, facilitando a procura do erro.

De seguida, calculamos a soma dos quadrados de `dx` e `dy`:

```
def distância(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dquadrado = dx**2 + dy**2
    print 'dquadrado é: ', dquadrado
    return 0.0
```

De novo, deverá executar o programa neste fase do seu desenvolvimento e validar que o valor impresso está correcto (deverá ser 25). Finalmente, pode utilizar `math.sqrt` para calcular o resultado a retornar:

```
def distância(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dquadrado = dx**2 + dy**2
    resultado = math.sqrt(dquadrado)
    return resultado
```

Se funcionar, está terminado o programa. Caso contrário, poderá querer imprimir o valor de `resultado` antes da instrução `return`.

A versão final da função não imprime para o écran quando é executada; apenas retorna um valor. As instruções `print` que escreveu são úteis para a depuração, mas assim que a função estiver a funcionar devem ser removidas. Estas instruções serviram como andaimes do seu programa, permitiram a construção do programa, mas assim que o mesmo ficou completo foram retiradas. A esta técnica chama-se ***scaffolding***.

Quando começa, deve apenas acrescentar uma ou outra linha de código de cada vez. À medida que vai ganhando experiência, seguramente conseguirá escrever e depurar pedaços maiores de código. De qualquer maneira, o desenvolvimento incrementar pode lhe salvar tempo a depurar.

Os aspectos chave do processo são:

1. Começar com um programa pequeno e fazer alterações incrementais. A qualquer momento, se existir um erro, deverá ter uma boa percepção de onde o mesmo se encontra.

2. Fazer uso de variáveis temporárias para armazenar valores intermédios de maneira a poder imprimir e verificar o seu valor.

3. Assim que o programa estiver a funcionar, deverá poder remover algumas dessas instruções ou até mesmo consolidar múltiplas instruções em expressões *inline*, mas apenas se não tornar o programa de difícil ler/compreender.

6.3 Composição

Como expectável, pode invocar uma função a partir de outra. Esta propriedade tem o nome de **composição**.

Considere por exemplo uma função que recebe dois pontos, o centro de uma circunferência e um ponto algures no seu perímetro, e calcula a area da circunferência.

Assuma que o ponto central se encontra atribuído à variável `xc` e `yc`, e que o ponto no perímetro se encontra atribuído a `xp` e `yp`. O primeiro passo é encontrar o raio do circulo, que não é mais do que a distância entre os dois pontos. Na secção anterior escrevemos uma função `distancia` que cumpre esse propósito:

```
raio = distancia(xc, yc, xp, yp)
```

O próximo passo é determinar a área da circunferência dada esse mesmo raio; também já escrevemos num capitulo anterior tal função:

```
resultado = area(raio)
```

Se encapsular estes passos numa função, obtém:

```
def area_circunferencia(xc, yc, xp, yp):
    raio = distância(xc, yc, xp, yp)
    resultado = area(raio)
    return resultado
```

As variáveis temporárias `raio` e `resultado` são úteis para o desenvolvimento e depuração, mas assim que o programa esteja a funcionar, podemos tornar o programa mais conciso através da composição das invocações às funções:

```
def area_circunferencia(xc, yc, xp, yp):
    return area(distância(xc, yc, xp, yp))
```

6.4 Funções Booleanas

As funções podem retornar valores booleanos, o que é normalmente conveniente quando pretendemos esconder testes complicados dentro de funções. Por exemplo:

```
def e_divisivel(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

É normal dar nomes que se pareçam com perguntas Às funções que retornam booleanos; `e_divisivel` retornar `True` ou `False` para indicar se `x` é divisível por `y`.

Eis um exemplo:

```
>>>   e_divisivel(6, 4)
False
>>>   e_divisivel(6, 3)
True
```

O resultado do operador == é um booleano, pelo que podemos re-escrever a função de uma forma mais concisa ao retornar directamente:

```
def e_divisivel(x, y):
    return x % y == 0
```

As funções booleanas são muitas das vezes utilizadas em instruções condicionais:

```
if e_divisivel(x, y):
    print 'x é divisível por y'
```

Pode ser tentador escrever algo como:

```
if e_divisivel(x, y) == True:
    print 'x é divisível por y'
```

Mas a comparação é desnecessária.

6.5 Mais recursividade

Apenas foi coberto até este capitulo um pequeno subconjunto da linguagem Python, no entanto, é interessante saber que este subconjunto é uma linguagem de programação *completa*, o que significa que tudo o que possa ser computado pode ser expresso nesta linguagem. Qualquer programa alguma vez escrito pode ser re-escrito utilizando apenas as propriedades da linguagem abordadas até ao momento (na realidade ainda faltam alguns comandos que permitem controlar dispositivos como teclado, rato, discos, etc., mas é tudo).

Demonstrar que esta afirmação é verdadeira é um exercício não trivial conseguido pela primeira vez por Alan Turing, um dos primeiros informáticos (algumas pessoas garantem que ele era um matemático, mas nesta altura a maioria dos informáticos tinham começado como matemáticos). Apropriadamente, é conhecida como o teste de Turing. Para uma mais completa (e precisa) discussão da teste de Turing, recomenda-se a leitura do livro *Introdução à Teoria da Computação* por Michael Sipser.

Para ter uma ideia do que é capaz de realizar com as ferramentas que abordámos até ao momento, vamos analisar algumas funções matemáticas recursivas. Uma definição recursiva é semelhante a uma definição circular, no sentido que a definição tem uma referencia para si mesma. Uma definição verdadeiramente circular não tem grande utilidade:

xpto: adjectivo utilizado para descrever algo que é xpto.

Se encontra-se esta definição no dicionário, poderia ficar aborrecido. Por outro lado, se pesquisar a definição de função factorial, denotada pelo símbolo !, deverá encontrar algo como:

$$0! = 1$$
$$n! = n(n-1)!$$

Esta definição postula que o factorial de 0 é 1, e que o factorial de qualquer outro valor, n, é n multiplicado pelo factorial de $n-1$.

Assim 3! é 3 vezes 2!, que é 2 vezes 1!, que é 1 vez 0!. Juntando tudo, 3! é igual a 3 vezes 2 vezes 1 vezes 1, que é 6.

Se for capaz de escrever a definição recursiva de qualquer coisa, então será capaz de escrever um programa em Python capaz de avaliar a mesma. O primeiro passo será decidir que parâmetros utilizar. Neste caso deverá ser claro que a função `factorial` recebe um inteiro:

```
def factorial(n):
```

Se o argumento for 0, tudo o que há a fazer é retornar 1:

```
def factorial(n):
    if n == 0:
        return 1
```

Caso contrário, e esta é a parte interessante, terá que fazer uma chamada recursiva para encontrar o factorial de $n-1$ e depois multiplicar por n:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recursao = factorial(n-1)
        resultado = n * recursao
        return resultado
```

O fluxo de execução deste programa é similar ao fluxo da `contagem decrescente` da Secção 5.8. Se invocar `factorial` com o valor 3:

Uma vez que 3 não é 0, o programa segue pelo segundo ramo e calcula o factorial de `n-1` ...

> Uma vez que 2 não é 0, o programa segue pelo segundo ramo e calcula o factorial de `n-1` ...
>
> > Uma vez que 1 não é 0, o programa segue pelo segundo ramo e calcula o factorial de `n-1` ...

Figura 6.1: Diagrama da pilha

Uma vez que 0 *é* 0, o programa segue pelo primeiro ramo e retorna 1 sem fazer mais nenhuma chamada recursiva.

O valor retornado (1) é multiplicado por *n*, que é 1, e o resultado é retornado.

O valor retornado (1) é multiplicado por *n*, que é 2, e o resultado é retornado.

O valor retornado (2) é multiplicado por *n*, que é 3, e o resultado, 6, torna-se no valor retornado pela função chamada originalmente e que despoletou todo processo.

A Figura 6.1 mostra o diagrama da pilha para esta sequência de chamadas à função.

Podemos ver os valores retornados a serem passados para cima na pilha. Em cada quadro, o valor retornado é o valor de `resultado`, que é o produto de `n` e `recursao`.

No ultimo quadro, a variável local `recursao` e `resultado` não existem, porque o ramo que as cria não é executado.

6.6 Salto de Fé

Seguir o fluxo de execução de um programa é uma das formas de ler o código de um programa, mas rapidamente pode-se tornar em algo labiríntico. Uma alternativa é aquilo a que se chama de "Salto de fé" (do inglês *leap of faith*. Quando analisa a chamada a uma função, em vez de seguir o fluxo de execução, deverá *assumir* que a função funciona correctamente e retorna o resultado certo.

Na realidade, sempre que utiliza uma função embutida está a dar um salto de fé. Quando invocou `math.cos` ou `math.exp`, não examinou o corpo destas funções. Simplesmente assumiu que estavam correctas uma vez que as pessoas que escreveram essas funções eram bons programadores.

O mesmo acontece quando invoca uma das suas funções. Por exemplo, na Secção 6.4, analisámos uma função chamada `e_divisivel` que determina se um número é divisível por outro. Assim que nos convencemos que esta função está correcta—porque examinámos o código e testámos—podemos utilizar a função sem voltar a analisar o seu código.

O mesmo acontece com os programas recursivos. Quando chegamos à invocação recursiva, em vez de seguirmos o fluxo de execução, deverá assumir que a chamada recursiva

funciona (retorna o valor correcto) e perguntamo-nos, "Assumindo que consigo descobrir o factorial de $n-1$, posso calcular o factorial de n?" Obviamente consegue, multiplicando por n.

É claro que é um pouco estranho assumir que uma função funciona correctamente antes de terminar a função e a testar, mas por essa razão é que se trata de um salto de fé!

6.7 Mais um exemplo

Para lá do `factorial`, o exemplo mais importante de recursividade definida por uma função matemática é a sequência de `fibonacci`, que tem a seguinte definição (ver `http://pt.wikipedia.org/wiki/Sequência_de_Fibonacci`):

$$\begin{aligned} \text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \end{aligned}$$

Traduzindo para Python, é algo como:

```
def fibonacci (n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Se tentar seguir o fluxo de execução, até que para valores pequenos de n, ficará perdido. Mas de acordo com o salto de fé, se assumir que as duas chamadas recursivas funcionam correctamente, então é claro que obterá o resultado correcto se os somar.

6.8 Verificar Tipos

O que acontece se invocar a função `factorial` e passar 1.5 como argumento?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

Parece-se com uma recursão infinita. Mas como pode tal acontecer? Existe um caso base—quando `n == 0`. Mas se `n` não for um inteiro, podemos *falhar* o caso base e chamar a recursão para sempre.

Na primeira chamada recursiva, o valor de `n` é 0.5. Na próxima, é de -0.5. A partir daí, vai ficando cada vez menor (mais negativo), mas nunca será 0.

Tem duas opções. Pode tentar generalizar a função `factorial` para funcionar com números reais, ou pode garantir que a função `factorial` verifica o tipo dos seus argumentos. A primeira opção é chamada de função `gamma` e está um pouco fora do âmbito deste livro. Pelo que deve escolher a segunda.

Pode utilizar a função embutida `isinstance` para verificar o tipo do argumento. E já que está a verificar coisas, pode garantir que se trata de um número positivo:

```
def factorial (n):
    if not isinstance(n, int):
        print 'Factorial só está definido para inteiros'
        return None
    elif n < 0:
        print 'Factorial não está definido para inteiros negativos.'
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

O primeiro ramo lida com números não inteiros; o segundo detecta números negativos. Em ambos os caso, o programa imprime uma mensagem de erro e retorna `None` para indicar que qualquer coisa de errado aconteceu:

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is not defined for negative integers.
None
```

Se ultrapassar as duas verificações, então sabe que n é positivo ou zero, pelo que podemos provar que a recursão termina.

Este programa demonstra um padrão chamado de **guardian** (guardião). As duas primeiras condições funcionam como guardiões que protegem o código seguinte de valores que podem causar um erro. Os guardiões possibilitam a prova que o código está correcto.

Na Secção **??** verá uma maneira alternativa mais flexível para imprimir uma mensagem de erro: despoletando uma excepção.

6.9 Depuração

Ao partir um programa em pequenas funções, cria um conjunto natural de pontos de controlo para depuração. Se uma função não funcionar, existem três possibilidades a considerar:

- Existe algo de errado com os argumentos que a função está a receber; a pré-condição é violada.
- Existe algo de errado com a função; a pós-condição é violada.
- Existe algo de errado com o valor retornado ou a maneira como é utilizado.

Para descartar a primeira possibilidade, pode acrescentar uma instrução `print` no inicio da função que mostre os valores dos parâmetros (e talvez os seus tipos). Ou pode escrever o código necessário para verificar as pré-condições explicitamente.

Se os parâmetros parecerem correctos, acrescente uma instrução `print` antes de cada instrução `return` e mostre o valor retornado. Se possível, verifique externamente o resultado. Considere chamar a função com valores que lhe permitam facilmente verificar o resultado (como no caso da Secção 6.2).

Se a função parecer estar a funcionar, verifique o código que chama a função e que o valor retornado está a ser utilizado correctamente (ou simplesmente utilizado!)

Acrescentar instruções `print` no início e fim de uma função pode ajudar a tornar o fluxo de execução mais visível. Por exemplo, eis uma versão de `factorial` com instruções `print`:

```
def factorial(n):
    espaco = ' ' * (4 * n)
    print espaco, 'factorial', n
    if n == 0:
        print espaco, 'retorna 1'
        return 1
    else:
        recursao = factorial(n-1)
        resultado = n * recursao
        print espaco, 'retorna', resultado
        return resultado
```

`espaco` é uma *string* de caracteres espaço que controla a indentação da saída do programa. Aqui fica o resultado para `factorial(5)`:

```
                    factorial 5
                factorial 4
            factorial 3
         factorial 2
     factorial 1
 factorial 0
 retorna 1
     retorna 1
         retorna 2
             retorna 6
                 retorna 24
                     retorna 120
```

Se ainda se sente confuso com o fluxo de execução, esta saída do programa pode ser útil. Demora algum tempo o desenvolvimento de *scaffolding*, mas um pouco de *scaffolding* pode poupar muita depuração.

Capítulo 7

Iteração

7.1 Múltipla atribuição

Como já deve ter reparado, é permitido fazer-se mais do que uma atribuição à mesma variável. Uma nova atribuição faz com que uma variável existente passe a referir a um novo valor (e deixe de se referir ao antigo).

```
nota=5
print(nota)
nota=7
print(nota)
```

Este programa irá imprimir `5 7`, uma vez que aquando da primeira impressão de `nota` o seu valor é 5, e na segunda vez é 7. A virgula no final da instrução `print` suprime a mudança de linha, daí que ambos os valores apareçam na mesma linha.

A Figura 7.1 mostra o que é a **múltipla atribuição** através de um diagrama de estados.

Com a múltipla atribuição, é importante distinguir entre a instrução de atribuição e a operação de igualdade. Uma vez que o Python utiliza o sinal de igual (=) para fazer atribuição,

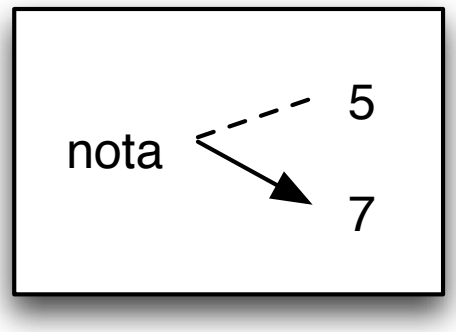

Figura 7.1: Diagram de estado

é tentador interpretar uma linha com `a = b` como sendo uma operação de igualdade. Mas não é!

Em primeiro lugar, a igualdade é uma operação simétrica e a atribuição não. Por exemplo, em matemática, se $a = 7$ então $7 = a$. Mas em Python, a instrução `a = 7` é permitida e `7 = a` não.

Acresce que em matemática uma igualdade é ou verdadeira ou falsa, para todo o sempre. Se agora $a = b$, então a será sempre igual a b. Em Python, uma atribuição permite que duas variaveis sejam iguais, but não é necessário que se mantenham assim indefinidamente.

```
a = 5
b = a # a e b são agora iguais
a = 3 # a e b não são mais iguais
```

A terceira linha altera o valor de `a` mas não altera o valor de `b`, pelo que não são mais iguais.

Embora a múltipla atribuição seja frequentemente útil, deverá ser utilizada com precaução. Se um valor for alterado frequentemente, pode tornar o código dificil de ler e depurar.

7.2 Actualização de variaveis

Uma das formas mais comuns de multipla atribuição é a **actualização**, onde o novo valor de uma variavel depende do seu valor anterior.

```
x = x+1
```

Podemos ler como "vai buscar o valor actual de `x`, acrescenta-lhe um, e actualiza `x` com o novo valor."

Se tentar actualizar uma variavel que não existe, irá deparar-se com um erro, uma vez que o Python analisa o lado direito antes de atribuir um valor a `x`:

```
>>> x = x+1
NameError: name 'x' is not defined
```

Antes de uma variavel poder ser actualizada, é necessário **inicializar**, normalmente com uma atribuição simples:

```
>>> x = 0
>>> x = x+1
```

Quando se actualiza uma variavel acrescentando 1, chama-se **incrementar**; subtraindo 1 chama-se **decrementar**.

7.3 A instrução `while`

Os computadores são normalmente utilizados para automatizar tarefas repetitivas. Repetir tarefas identicas ou similares sem cometer qualquer erro é algo que os computadores fazem muito bem e as pessoas péssimamente.

Já analisámos dois programas, `contagem_decrescente` e `print_n`, que utilizaram da recursividade para conseguir repetição, que toma também o nome de **iteração**. Uma vez que a iteração é tão comum, a linguagem Python apresenta várias caracteristicas que a tornam mais fácil. Uma é a instrução `for` que vimos na Secção **??**. Voltaremos a esta mais tarde.

Outra é a instrução `while`. Aqui fica uma versão da `contagem_decrescente` que utiliza a instrução `while`:

```
def contagem_decrescente(n):
    while n > 0:
        print n
        n = n-1
    print 'Ignição!'
```

Se lermos o código como se fosse um texto em inglês é facil perceber o que faz. Traduzindo, "Enquanto (`while`) `n` for maior que 0, imprimir no ecran o valor de `n` e decrementar o valor de n em 1. Quando chegar a 0, imprimir no ecran `Ignição!`"

Analisando de forma mais formal o fluxo de execução da instrução `while`:

1. Avaliar a condição, que é `True` (verdadeira) ou `False` (falsa).
2. Se a condição for falsa, sair da instrução `while` e continuar a execução do programa com a próxima instrução.
3. Se a condição for verdadeira, executar o corpo da instrução e voltar ao primeiro passo.

A este tipo de fluxo chama-se de **ciclo** uma vez que o terceiro passo redireciona para o primeiro de forma repetida.

O corpo de um ciclo deve alterar o valor de uma ou mais variaveis para que o eventualmente a condição no passo 1 passe a falsa e o ciclo seja interrompido. Caso contrário, o ciclo irá se repetir indefinidademente criando aquilo que se chama um **ciclo infinito**. Um exemplo cómico para um informático de um ciclo infinito são as instruções na maioria dos shampos: "Lavar, Enxaguar, Repetir".

No caso da `contagem_decrescente`, é possivel provar que o ciclo termina porque sabemos que o valor de `n` é finito, e podemos observar que o valor de `n` vai diminuindo a cada ciclo, até eventualmente ser 0. Noutros casos, não é tão facil ter certezas:

```
def sequencia(n):
    while n != 1:
        print n,
        if n%2 == 0:        # n é par
            n = n/2
        else:               # n é impar
            n = n*3+1
```

A condição neste ciclo é que `n != 1`, pelo que o ciclo se irá perpétuar até que `n` seja `1`, o que leva a condição a tornar-se falsa.

A cada iteração, o programa imprime no ecran o valor de `n` e verifica se é par ou impar. Caso seja par, `n` é dividido por 2. Se for impar, o valor de `n` é substituido por `n*3+1`. Por exemplo, se o argumento passo a `sequencia` for 3, o resultado da sequencia é 3, 10, 5, 16, 8, 4, 2, 1.

Uma vez que `n` algumas vezes cresce outras diminui, não é possivel provar que `n` alguma vez será igual a 1, e que o programa termine. Outras há, em que para certos valores de `n` é possivel provar que termina. Por exemplo, se o valor inicial de `n` for uma potencia de

base dois, então o valor de `n` será par em todas as iterações até que chegará a 1. O exemplo anterior termina com uma dessas sequencias, começada em 16.

A grande questão é saber se é possivel provar que este programa termina para *todos valores positivos* de `n`. Até à data, ninguem conseguiu provar *ou* desmentir tal afirmação. (ver `http://en.wikipedia.org/wiki/Collatz_conjecture`.)

7.4 `break`

Muitas vezes não sabemos que devemos interromper um ciclo antes de irmos a meio do seu corpo. Nessas situações é possivel utilizar a instrução **break** para interromper o ciclo.

Por exemplo, imagine que pretende receber valores do utilizador através do teclado até que o mesmo escreva `feito`. Poderia escrever:

```
while True:
    line = raw_input('> ')
    if line == 'feito':
        break
    print line

print 'Feito!'
```

A condição do ciclo é `True` (verdadeiro), o que é sempre verdade, pelo que o ciclo repete até que alcance a instrução `break`.

Em cada iteração, pede ao utilizador informação com o sinal de maior. Se o utilizador inserir `feito`, a instrução `break` sai do corpo do ciclo. Caso contrário o program imprime o quer que o utilizador tenha digitado e volta ao inicio do ciclo. Eis um possivel comportamento:

```
> ainda nao
ainda nao
> feito
Feito!
```

Esta maneira de escrever um ciclo `while` é bastante comum uma vez que permite verificar condições em qualquer parte do ciclo (e não apenas no inicio), e permite ainda exprimir a condição de forma assertiva (“pára quando isto acontecer”) em vez de forma negativa (“continua até isto acontecer.”).

7.5 Raizes Quadradas

Os ciclos são normalmente utilizados em programas que calculam resultados numericos começando com um valor aproximado que é iterativamente melhorado. Por exemplo, uma forma de calcular a raiz quadrada é utilização no método Newtonian. Imagine que pretende determinar a raiz quadrada de a. Se começar com qualquer valor aproximado, x, pode calcular uma melhor estimativa utilizando a formula:

$$y = \frac{x + a/x}{2}$$

Por exemplo, ise a for 4 e x for 3:

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print y
2.16666666667
```

O que está mais perto da resposta correcta ($\sqrt{4} = 2$). Se repetir o processo com a nova estimativa, fica ainda mais perto:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00641025641
```

Após mais algumas iterações, a estimativa está quase correcta:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00000000003
```

Geralmente não sabemos de antemão quantas vezes teremos que repetir para obter a resposta correcta, mas sabemos que alcançamos o objectivo quando o valor estimado deixar de se alterar:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
```

Quando `y == x`, podemos para. Aqui fica um ciclo que começa com um valor estimado, `x`, e melhora até que o valor deixe de sofrer alterações:

```
while True:
    print x
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

Para a maioria dos valores de `a` esta aproximação funciona correctamente, mas geralmente é perigoso testar a igualdade entre `float`. Os valores representado em virgula flutuante (`float`) são apenas aproximações: números racionais tais como $1/3$, e irracionais, como $\sqrt{(2)}$, não podem ser representados com exactidão através de um `float`.

Em vez de verificar se `x` e `y` são exactamente iguais, é mais seguro utilizar a função nativa `abs` para calcular o valor absoluto ou norma da diferença entre ambos:

```
    if abs(y-x) < delta:
        break
```

Onde `delta` é uma valor como `0.0000001` que determina o quão aproximados deverão ser os valores.

7.6 Algoritmos

O método Newtoniano é um exemplo de um **algoritmo**: é um processo mecanico que permite resolver uma classe de problemas (neste caso, calcular a raiz quadrada).

Não é fácil criar um algoritmo. Talvez seja mais útil começar por algo que não é um algoritmo. Quando aprendemos a multiplicar algarismos de um digito, provavelmente decorámos a tabuada. Na realidade, memorizámos 100 operações especificas. Este tipo de conhecimento não é algoritmico.

Mas se fosse um pouco "preguiçoso", muito provavelmente descobriu alguns truques. Por exemplo, para encontrar o produto de n por 9, pode escrever $n - 1$ como primeiro digito e $10 - n$ como segundo digito. Este truque é uma solução genérica que permite multiplicar qualquer número de apenas um digito por 9. Isto é um algoritmo!

De forma similar, as tecnicas que aprendeu para adicionar e subtrair com transporte e a divisão são todas algortimos. Uma das caracteristicas dos algoritmos é que não necessitam de inteligencia para operar. São processos mecanicos em que um passo se segue ao anterior de acordo com um conjunto de regras pré estabelecidas.

É até algo embaraçoso que os humanos passem tanto tempo na escola a aprender a executar um processo algoritmico, que literalmente não requere qualquer inteligencia.

Por outro lado, o processo de desenhar algoritmos é interessante, desafiante intelectualmente, e parte central daquilo que chamamos de programação.

Algumas das coisas que as pessoas fazem naturalmente, sem qualquer dificuldade or consciencia, são das mais dificeis de exprimir algoritmicamente. Compreender a linguagem natural é disso um exemplo. Todos nós o fazemos, mas até hoje ninguem foi capaz de explicar *como* o fazemos, pelo menos na forma de um algoritmo.

7.7 Depuração

À medidade que começa a escrever programas maiores, o tempo que leva a depurar irá também aumentar. Mais código significa maior probabilidade de se cometer um erro, e mais locais onde um *bug* se pode esconder.

Uma forma de reduzir o tempo de depuração é seccionando a depuração. Por exemplo, se o programa tiver 100 linhas de código e as verificar uma a uma, irá gastar 100 passos.

Em vez disso, tente partir o problema a meio. Procure uma linha sensivelmente a meio do código, onde possa analizar um valor intermédio. Acrescente um `print` (ou algo com um efeito muito especifico) e execute o programa.

Se a verificação a meio do programa estiver incorrecta, o problema necessáriamente encontra-se na primeira parte do programa.

Sempre que fizer uma verificação como esta, está a poupar metade das linhas de código que tem que analisar. Após seis passos (muito menos que 100), deverá ter singido o problema a uma ou duas linhas apenas, pelo menos em teoria.

Na prática não é sempre claro o que é o "meio do programa" e como verificar nesse ponto. Não faz sentido contar as linhas para encontrar o meio exacto do programa . Em vez disso opte sempre por procurar locais onde os erros podem surgir ou onde seja mais fácil testar. Depois procure um local onde a probabilidade do erro existir seja igual antes e depois da verificação.

Capítulo 8

Strings

8.1 Uma *String* é uma sequência

Uma *string* é uma **sequência** de caracteres. Pode aceder a cada um dos caracteres recorrendo ao operador 'parênteses recto'.:

```
>>> fruta = 'banana'
>>> letra = fruta[1]
```

A segunda instrução selecciona o caracter numero 1 de `fruta` e atribui a `letra`.

À expressão dentro de parênteses rectos chamamos de **índice**. O índice indica qual o caracter na sequência a que pretendemos aceder (daí o nome).

Mas os resultados podem não ser os esperados:

```
>>> print letra
a
```

Para a maioria das pessoas, a primeira letra de `'banana'` é `b`, e não `a`. No entanto, para um programador, o índice é uma distancia do inicio da *string*, sendo que a primeira letra está à distância zero.

```
>>> letra = fruta[0]
>>> print letra
b
```

Assim `b` é a 0ª letra de `'banana'`, `a` é a 1ª letra, e `n` é a 2ª letra.

Pode utilizar qualquer expressão, incluindo variáveis e operadores, como um índice, mas o seu valor necessita ser um inteiro. Caso contrário:

```
>>> letra = fruta[1.5]
TypeError: string indices must be integers, not float
```

8.2 `len`

`len` é uma função nativa que devolve o número de caracteres numa *string*, isto é, o seu tamanho:

```
>>> fruta = 'banana'
>>> len(fruta)
6
```

Para obter a última letra de uma *string*, poderá ser tentado a escrever algo como:

```
>>> tamanho = len(fruta)
>>> ultima = fruta[tamanho]
IndexError: string index out of range
```

A razão do erro `IndexError` é o não existir nenhuma letra na *string* `'banana'` com o índice 6. Uma vez que a contagem começa em zero, as seis letras estão numeradas de 0 a 5. Para aceder ao ultimo caracter, é necessário subtrair 1 ao `tamanho`.

```
>>> ultima = fruta[tamanho-1]
>>> print ultima
a
```

Em alternativa, pode usar índices negativos, que conta para trás a partir do inicio da *string*. A expressão `fruta[-1]` produz também a ultima letra, `fruta[-2]` produz a penúltima, e por ai a diante.

8.3 Atravessamento com o ciclo `for`

Inumeras vezes é necessário processar *strings* um caracter de cada vez. Normalmente começamos pelo inicio, e seleccionamos um caracter de cada vez, operando sobre este, até chegar ao fim. Este padrão de processamento é apelidado de **Atravessamento**. Uma forma de escrever este padrão é recorrendo ao ciclo `while`:

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print letra
    indice = indice + 1
```

Este ciclo atravessa a *string* e apresenta cada um dos caracteres na sua própria linha. A condição do ciclo é `indice < len(fruta)`, pelo que quando `indice` for igual ao tamanho da *string*, a condição será falsa, e o corpo do ciclo não será executado. O último caracter acedido é portanto o com o índice `len(fruta)-1`, que não é mais do que o ultimo caracter da *string*.

Outra maneira de fazer um atravessamento é com o ciclo `for`:

```
for char in fruta:
    print char
```

A cada passagem do ciclo, o próximo caracter na *string* é atribuído à variável `char`. O ciclo repete até não haver mais caracteres na *string*.

O próximo exemplo demonstra como utilizar a concatenação (adição de *string*) e um ciclo `for` para gerar uma serie alfabética (isto é, ordenada alfabeticamente) de palavras que partilham o mesmo sufixo mas começam por uma vogal diferente.

```
prefixes = 'AEIOU'
suffix = 'ma'

for letter in prefixes:
    print letter + suffix
```

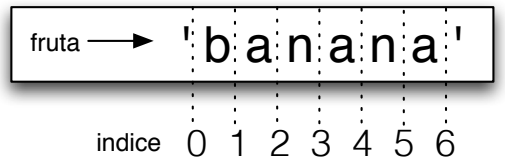

Figura 8.1: Indices das partes

O resultado é:

```
Ama
Ema
Ima
Oma
Uma
```

O exemplo é trivial e falha a acentuação em Imã e Omã. Como exercício modifique o código para corrigir esta situação.

8.4 Partes de *String*

Podemos dividir uma *string* em várias **partes**. Seleccionar uma parte é similar a seleccionar um caracter:

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:12]
Python
```

O operador `[n:m]` devolve a parte da *string* do "n-ésimo" caracter ao "m-ésimo" caracter, incluindo o primeiro mas excluindo o ultimo. Este comportamento é contra-intuitivo, mas ajuda se imaginarmos os indices a apontar *entre* os caracteres, como na Figura 8.1.

Se omitir o primeiro indice (antes dos dois pontos), a parte começa no ínicio da *string*. Já se omitir o segundo índice, a parte estende-se até ao final da *string*

```
>>> fruta = 'banana'
>>> fruta[:3]
'ban'
>>> fruta[3:]
'ana'
```

Se o primeiro índice for maior ou igual ao segundo o resultado é uma ***string* vazia**, representada pelas aspas:

```
>>> fruta = 'banana'
>>> fruta[3:3]
''
```

Uma *string* vazia não contém qualquer caracter e tem um tamanho 0, mas tirando isso, é igual a todas as outras *strings*.

8.5 As *String* são imutáveis

É tentador utilizar o operador `[]` no lado esquerdo de uma operação de atribuição, com a intenção de alterar um caracter numa string. Por exemplo:

```
>>> cumprimento = 'ola, Mundo!'
>>> cumprimento[0] = 'O'
TypeError: 'str' object does not support item assignment
```

O "object" neste caso é a *string* e o "item" é o caracter que pretendemos atribuir. Para já, um **object** é a mesma coisa que um valor, mas voltaremos a esta definição mais à frente. Um **item** é um dos valores numa sequência.

A razão pela qual existe um erro é porque as *strings* são imutáveis, o que significa que não se pode alterar uma string já existente. O melhor que se consegue é criar uma nova *string* que seja uma variação da original.

```
>>> cumprimento = 'ola, Mundo!'
>>> novo_cumprimento = 'O' + cumprimento[1:]
>>> print novo_cumprimento
Ola, Mundo!
```

O exemplo concatena uma nova letra inicial a uma parte de `cumprimento`. Não afectando a *string* original.

8.6 Pesquisa

O que faz a próxima função?

```
def procura(palavra, letra):
    indice = 0
    while indice < len(palavra):
        if palavra[indice] == letra:
            return indice
        indice = indice + 1
    return -1
```

De certa forma, `procura` é o contrário do operador `[]`. Em vez de receber o indice e extrair o caracter correspondente, recebe um caracter e procura o primeiro índice onde o caracter aparece. Se não encontrar o caracter, a função devolve `-1`.

Este é o primeiro exemplo em que se demonstra a utilização da instrução `return` dentro de um ciclo. Se `palavra[indice] == letra`, o ciclo é interrompido e a função retorna imediatamente.

Se o caracter não aparecer na *string*, o programa termina o ciclo normalmente e devolve `-1`.

A este padrão de programação—atravessar uma sequência e devolver assim que encontremos o que procuramos–chamamos de **pesquisa**.

8.7 Ciclos e contagens

O próximo programa conta o número de vezes que a letra `a` aparece numa *string*:

```
palavra = 'banana'
contagem = 0
for letra in palavra:
    if letra == 'a':
        contagem = contagem + 1
print contagem
```

Este programa demonstra outro padrão de programação chamado **contagem**. A variável `contagem` é inicializada a 0 e é depois incrementada sempre que `a` é encontrado. Quando o ciclo termina, `contagem` contém o resultado—o número total de `a`'s.

8.8 Métodos sobre *Strings*

Um **método** é similar a uma função–recebe argumentos e retorna um valor—mas o sintaxe é diferente. Por exemplo, o método `upper` recebe uma *string* e devolve uma nova *string* com todos os caracteres em maiúsculas:

Em vez de utilizar o sintaxe de uma função `upper(word)`, utiliza o sintaxe de um método `palavra.upper()`.

```
>>> palavra = 'banana'
>>> nova_palavra = palavra.upper()
>>> print nova_palavra
BANANA
```

Esta notação que utiliza um ponto antes do nome do método, `upper`, e o nome da *string* sobre o qual o método deve ser aplicado. Os parênteses vazios indicam que este metodo não tem qualquer argumento.

Uma chamada a um método tem o nome de **invocação**; neste caso podemos dizer que invocámos `upper` sobre `palavra`.

```
>>> palavra = 'banana'
>>> indice = palavra.find('a')
>>> print indice
1
```

Neste outro exemplo, invocamos `find` sobre `palavra` e passamos a letra que procuramos como argumento do método.

Na realidade, `find` é um método muito mais genérico que a nossa função `procura`; já que pode encontrar partes da *string* e não apenas caracteres:

```
>>> palavra.find('na')
2
```

Pode ainda receber um segundo argumento com o índice a partir do qual deve começar:

```
>>> palavra.find('na', 3)
4
```

E um terceiro argumento com o índice onde deve terminar:

```
>>> name = 'ama'
>>> name.find('a', 1, 2)
-1
```

Esta procurar falha porque a não existe no intervalo de índices 1 a 2 (não inclui 2).

Leia mais sobre os métodos sobre *strings* em `http://docs.python.org/2/library/stdtypes.html#string-methods`. Deverá experimentar com alguns deles para ter a certeza que os compreende. `strip` e `replace` por exemplo são extremamente úteis.

A documentação no entanto utiliza uma sintaxe que se pode tornar confusa. Por exemplo, em in `find(sub[, start[, end]])`, os parênteses rectos indicam que o argumento é opcional. Assim `sub` é necessário, mas `start` é opcinal, e se incluir `start`, então `end` é opcional.

8.9 O operador `in`

O operador `in` é um operador booleano que aceita duas *strings* e devolve `True` se a primeira estiver contida na segunda:

```
>>> 'a' in 'banana'
True
>>> 'caroço' in 'banana'
False
```

Por exemplo, a próxima função imprime todas as letras de `palavra1` que também aparecem em `palavra2`:

```
def em_ambas(palavra1, palavra2):
    for letra in palavra1:
        if letra in palavra2:
            print letra
```

Se utilizássemos nomes de variáveis em Inglês, até seria possível ler de forma natural a função: "for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter."

O que acontece quando comparamos alhos com bugalhos:

```
>>> em_ambas('alhos', 'bugalhos')
a
l
h
o
s
```

8.10 Comparação de *Strings*

Os operadores relacionais funcionam com as *strings*. Para verificar se duas *strings* são iguais:

```
if palavra == 'banana':
    print 'Então, bananas.'
```

Outras operações relacionais são úteis para ordenar palavras alfabeticamente:

```
if palavra < 'banana':
    print 'A palavra ' + palavra + ', vem antes de banana.'
elif palavra > 'banana':
    print 'A palavra ' + palavra + ', vem depois de banana.'
else:
    print 'Então, bananas.'
```

O Python não trata as letras minúsculas e maiúsculas da mesma maneira que um humano. Todas as maiúsculas veem antes das minúsculas, pelo que:

```
'A palavra Cenoura, vem antes de banana.'
```

A maneira mais comum de ultrapassar esta situação é converter a string para um formato comum antes de proceder à comparação, por exemplo converter tudo para minúsculas.

8.11 Depuração

Quando utiliza índices para atravessar os valores numa sequência, é difícil acertar com o inicio e com o fim. Aqui temos uma função em que é suposto comparar duas palavras e devolver `True` se as palavras forem um palíndromo (a primeira é igual à segunda, escrita de trás para a frente). No entanto contém dois erros:

```
def e_inverso(palavra1, palavra2):
    if len(palavra1) != len(palavra2):
        return False

    i = 0
    j = len(palavra2)

    while j > 0:
        if palavra1[i] != palavra2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

O primeiro `if` verifica se as palavras têm o mesmo tamanho. Caso contrário, podemos retornar `False` imediatamente e assim no resto da função podemos assumir que as palavras têm o mesmo tamanho. Isto é um exemplo do padrão guardião da secção **??**.

`i` e `j` são índices: `i` atravessa `palavra1` em frente enquanto `j` atravessa `palavra2` para trás. Se encontrarmos duas letras que não são iguais, podemos retornar `False` imediatamente. Se chegarmos ao fim do ciclo e todas as letras forem iguais, retornamos `True`

Se testar esta função com as palavras "sala" e "alas", obteríamos um valor `True`, mas deparamo-nos com um `IndexError`:

```
>>> e_inverso('sala', 'alas')
...
  File "e_inverso.py", line 15, in e_inverso
    if palavra1[i] != palavra2[j]:
IndexError: string index out of range
```

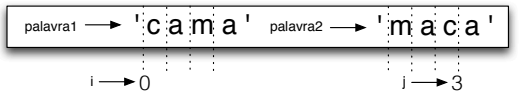

Figura 8.2: Diagram de estados

Para fazer a depuração de um erro destes, a primeira sugestão é adicionar um `print` antes da linha onde surge o erro.

```
while j > 0:
 print i, j #imprimir aqui!
    if palavra1[i] != palavra2[j]:
        return False
    i = i+1
    j = j-1
```

Agora quando executamos o programa de novo, temos um pouco mais de informação:

```
>>> e_inverso('sala', 'alas')
0 4
...
IndexError: string index out of range
```

Na primeira passagem pelo ciclo, o valor de `j` é 4, que está fora do intervalo da *string* `'sala'`. O indice do ultimo caracter é 3, pelo que o valor inicial de `j` devia ser `len(palavra2)-1`.

Se corrigir este erro e executar de novo, obtém:

```
>>> e_inverso('sala', 'alas')
0 3
1 2
2 1
True
```

Desta vez, obtém a resposta correcta mas aparentemente o ciclo executa apenas três vezes, o que é suspeito. Para ter uma melhor ideia do que está a acontecer, é útil desenhar um diagrama de estados. Na primeira iteração, o diagrama para `e_inverso` está representado na Figura 8.2.

Com alguma liberdade criativa fez-se um arranjo do diagrama para que os valores de `i` e `j` indiquem os caracteres em `palavra1` e `palavra2`.

É fácil agora perceber que o ciclo é interrompido sem que `j` tome o valor 0 que é um índice válido. É pois necessário corrigir a expressão para `while j >= 0:`.

Capítulo 9

Caso de Estudo: Jogos de Palavras

9.1 Ler listas de palavras

Para este capitulo é necessário uma lista de palavras Portuguesas. Existem inúmeras listas de palavras disponíveis na internet, para os nossos propósitos vamos fazer uso de uma lista de palavras disponibilizada pelo Projecto Natura da Universidade do Minho em `http://natura.di.uminho.pt/download/sources/Dictionaries/wordlists/LATEST/`. Esta lista é um verdadeiro dicionários da lingua portuguesa e é utilizada em inúmeros dicionários como por exemplo dos projectos OpenOffice e LibreOffice. Deverá fazer download da última versão do ficheiro `wordlist-ao-*.txt.xz` e extrair o seu conteúdo, no final obterá um ficheiro txt que contém uma palavra por linha.

O ficheiro obtido é de texto simples, pelo que pode ser acedido por através de qualquer editor de texto, mas também pode ser lido utilizando Python. A função nativa `open` recebe por argumento o nome do ficheiro e retorna um objecto do tipo **file** que pode ser utilizado para ler o conteúdo.

```
>>> fin = open('wordlist-ao-latest.txt')
>>> print fin
<open file 'wordlist-ao-latest.txt', mode 'r' at 0x10be17c90>
```

`fin` é um nome habitualmente utilizado para designar um objecto do tipo **file** utilizado para leitura (do inglês *file input*). O Modo `'r'` indica que o ficheiro se encontra aberto para leitura (*read*), por oposição ao modo `'w'` (*write*).

O objecto **file** possui diferentes métodos de leitura, incluindo o `readline`, que lê os caracteres de um ficheiro até chegar à próxima linha e retorna o resultado como uma *string*:

```
>>> fin.readline()
'AAC\n'
```

A primeira palavra, nesta lista particular de `wordlist-ao-latest.txt`, é "AAC", que é na realidade um acrónimo (Advanced Audio Coding). A sequência `\n` representa um caracter em branco (não visível) chamado de *newline* (mudança de linha), que separam esta palavra da próxima.

O objecto **file** encarrega-se de manter um registo de onde se encontra no ficheiro (mantém estado), pelo que se chamar novamente por `readline`, obtém a próxima palavra:

```
>>> fin.readline()
'ABC\n'
```

O caracter em branco pode perturbar os nossos propósitos, mas facilmente pode ser removido recorrendo ao método `strip` de *string*:

```
>>> linha = fin.readline()
>>> palavra = linha.strip()
>>> print palavra
ACP
```

Também é possível um objecto do tipo **file** como parte de um ciclo. Este programa lê `wordlist-ao-latest.txt` e imprime cada palavra, uma por linha:

```
fin = open('wordlist-ao-latest.txt')
for linha in fin:
    palavra = linha.strip()
    print palavra
```

Exercício 9.1. *Escreva um programa que leia* `wordlist-ao-latest.txt` *e imprima apenas as palavras com mais de 20 caracteres (não contabilizando os caracteres brancos)*

9.2 Exercícios

Existem soluções para os próximos exercícios na secção seguinte. Mas antes de as analisar, tente resolver por si.

Exercício 9.2. *Um lipograma é um texto escrito propositadamente com o proposito de omitir determinada(s) letra(s) do alfabeto. Existem lipogramas famosos na lingua inglesa como o* Gadsby *de Ernest Vincent Wright com 50.000 palavras, ou o "La Disparition"de Georges Perec.*

Escreva uma função chamada `nao_tem_a` *que retorne* `True` *se uma dada palavra não tiver a letra "a".*

Altere o seu programa anterior para imprimir as palavras que não teem "a" e calcule a percentagem de palavras na lista que não teem "a".

Exercício 9.3. *Escreva uma função chamada* `evita` *que recebe uma palavra e uma string de letras proibidas, e que retorne* `True` *se a palavra não utilizar nenhuma das letras proibidas.*

Altere o programa para pedir ao utilizador que insira uma string de letras proibidas que imprima o numero de palavras que não contém nenhuma delas. Consegue encontrar uma combinação de 5 letras proibidas que excluem o menor numero de palavras ?

Exercício 9.4. *Escreva uma função chamada* `usa_apenas` *que recebe uma palavra e uma string de letras disponíveis, e que retorna* `True` *se a palavra utilizar apenas as disponíveis. Será que existe alguma palavra nestas condições?*

Exercício 9.5. *Escreva uma função chamada* `usa_todas` *que recebe uma palavra e uma string de letras, e que retorna* `True` *se a palavra utilizar todas as letras necessárias pelo menos uma vez. Quantas palavras existem que utilizem todas as vogais?*

Exercício 9.6. *Escreva uma função chamada* `e_alfabetica` *que retorna* `True` *se as letras numa palavra aparecerem por ordem alfabética (não há problema se as letras repetirem). Quantas destas palavras existem?*

9.3 Pesquisa

Todos os exercícios na secção anterior tem uma coisa em comum; podem ser resolvidos com o padrão de pesquisa utilizado na secção **??**. O exemplo mais simples é:

```
def nao_tem_a(palavra):
    for letra in palavra:
        if letra == 'a':
            return False
    return True
```

O ciclo `for` atravessa todos os caracteres em `palavra`. Se encontrar a letra "a", podemos imediatamente retornar `False`, caso contrário processa a próxima letra. Se o ciclo terminar normalmente, então tal significa que não foi encontrada a letra "a", e é retornado `True`.

`evita` é uma versão mais geral de `nao_tem_a` que mantém a mesma estrutura:

```
def evita(palavra, proibida):
    for letra in palavra:
        if letra in proibida:
            return False
    return True
```

Novamente, podemos retornar `False` assim que nos deparamos com uma letra proibida, se o ciclo terminou normalmente então podemos retornar `True`.

`usa_apenas` é similar, excepto o facto da condição ser a inversa:

```
def usa_apenas(palavra, disponiveis):
    for letra in palavra:
        if letra not in disponiveis:
            return False
    return True
```

Em vez de uma lista de letras proibidas, temos uma lista de letras disponíveis. Se encontrarmos uma `letra` que não pertença à lista de letras `disponiveis`, podemos retornar `False`.

`usa_todas` é também muito similar, mas trocamos o papel da variável `palavra` e `disponiveis`:

```
def usa_todas(palavra, necessarias):
    for letra in necessarias:
        if letra not in palavra:
            return False
    return True
```

Em vez de atravessar as letras da `palavra` o ciclo percorre a lista de letras `necessarias`. Se qualquer uma das necessárias não aparecer em `palavra`, podemos retornar `False`.

Se estivéssemos a pensar como um informático, teríamos reconhecido que `usa_todas` não era mais que uma instanciação do problema anterior, e teríamos resolvido com:

```
def usa_todas(palavra, necessarias):
    return usa_apenas(necessarias, palavra)
```

Isto é um exemplo de um método de desenvolvimento de programas chamado de **reconhecimento problemas**, que significa que reconhecemos que o problema com que nos deparamos é uma instancia de um problema previamente resolvido, pelo que aplicamos a solução anterior.

9.4 Ciclos com índices

As funções na secção anterior foram escritas com um `for` que iterou sobre cada caracter na *string* sem qualquer necessidade de conhecer o índice de cada um.

Para a função `e_alfabetica` é necessário comparar letras adjacentes, o que é um pouco difícil com um ciclo `for`:

```
def e_alfabetica(palavra):
    anterior = palavra[0]
    for c in palavra:
        if c < anterior:
            return False
        previous = c
    return True
```

Em alternativa podemos escrever de forma recursiva:

```
def e_alfabetica(palavra):
    if len(palavra) <= 1:
        return True
    if palavra[0] > palavra[1]:
        return False
    return is_abecedarian(palavra[1:])
```

Ou ainda recorrendo a um ciclo `while`:

```
def e_alfabetica(palavra):
    i = 0
    while i < len(palavra)-1:
        if palavra[i+1] < palavra[i]:
            return False
        i = i+1
    return True
```

O ciclo começa com `i=0` e termina quando `i=len(palavra)-1`. A cada iteração, compara o i^{o} caracter (que podemos assumir como sendo o caracter actual) com o $i+1^{o}$ caracter (que podemos assumir tratar-se do próximo).

Se o próximo caracter for menor (alfabeticamente anterior) ao caracter actual, então descobrimos uma falha na progressão alfabética, e é possível terminar com um `return False`.

Se chegar ao final o ciclo sem detectar qualquer falha, então a palavra passou o teste. Para ter a certeza que o ciclo termina correctamente, considere por exemplo a palavra `'amor'`. O tamanho da palavra é 4, pelo que o ciclo executa uma ultima vez quando `i` for 2, que é o endereço do penúltimo caracter. Na ultima iteração, é feita a comparação do último caracter com o penúltimo, que é o pretendido.

Aqui fica uma versão de `e_palindromo` que recorre ao uso de dois indices, uma começa no inicio e vai até ao fim, e outro começa no fim e regride até ao inicio.

```
def e_palindromo(palavra):
    i = 0
    j = len(palavra)-1

    while i<j:
        if palavra[i] != palavra[j]:
            return False
        i = i+1
        j = j-1

    return True
```

Ou, caso tenha reparado que estamos a comparar a palavra com o seu inverso:

```
def e_palindromo(palavra):
        return palavra == palavra[::-1]
```

`palavra[::-1]` inverte o conteúdo de de , no próximo capitulo iremos abordar como.

9.5 Depuração

Testar um programa é uma tarefa árdua. As funções neste capitulo são muito simples de testar pois é possível o fazer à mão. Mesmo assim, é difícil ou até talvez impossível escolher um conjunto de palavras que permita testar todas as possíveis condições de erro.

Tomemos `nao_tem_a` como exemplo, existem casos óbvios a testar: palavras que teem 'a' devem retornar `False`, palavras que não teem devem retornar `True`. Não será difícil encontrar uma palavra para cada um destes casos.

Mas dentro de cada caso, existem sub-casos menos óbvios. Entre as palavras que teem um "a", devem ser testadas palavras com "a" no inicio, no meio e no fim. Deve testar palavras compridas, curtas e muito curtas, como uma *string* vazia. A *string* vazia é um caso especial, onde muitos dos erros não óbvios costumam ocorrer.

Para lá dos casos de teste que gerou, pode também testar o programa com uma lista maior como `wordlist_ao_latest.txt`. Passando os olhos nos resultados, poderá encontrar erros, mas tenha cuidado: pode encontrar alguns tipos de erro (palavras que não deviam aparecer, mas que aparecem) mas não de outro tipo (palavras que deviam aparecer, mas não aparecem).

De forma geral, testar permite-nos encontrar erros (*bugs*), mas não é fácil gerar um bom conjunto de casos de teste, e mesmo que o faça, não pode ter a certeza que o programa está completamente correcto.

Como disse um celebre Informático:

O teste a um programa informático pode ser usado para provar que existem erros, mas nunca para provar a sua ausência! — Edsger W. Dijkstra