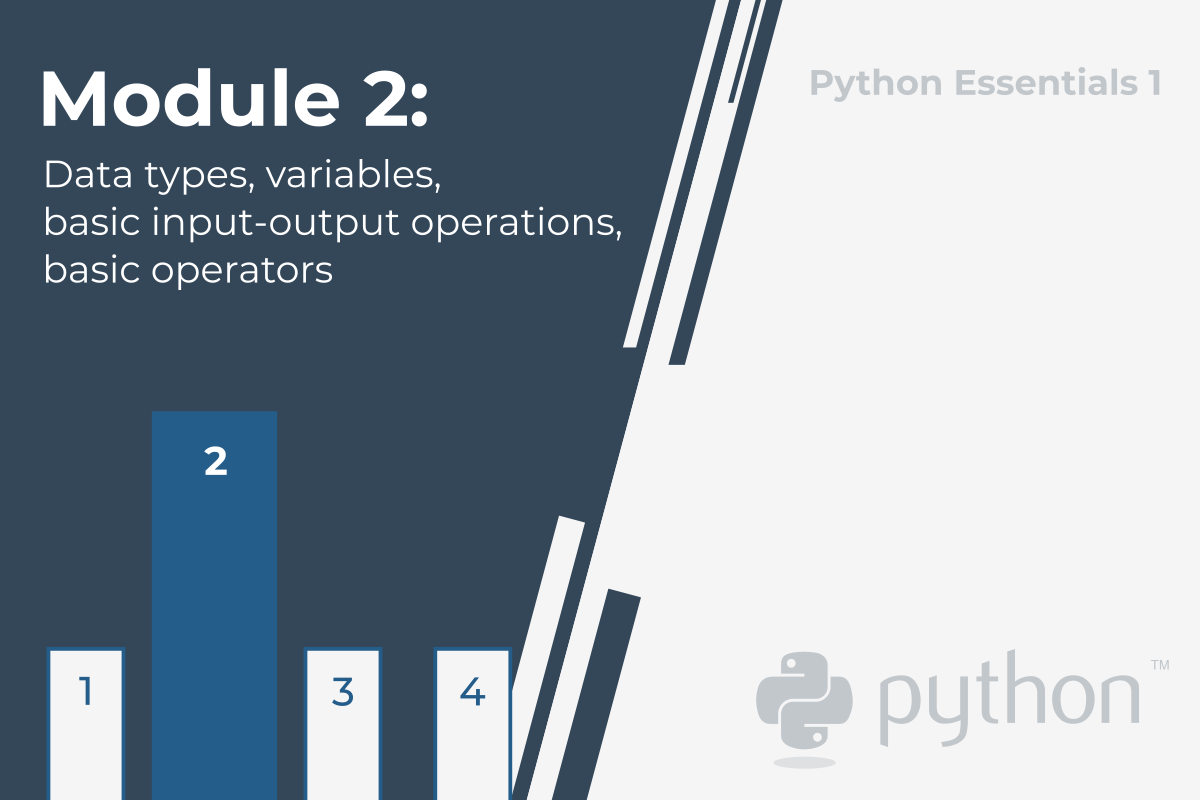
**Python Essentials 1:  
Módulo 2**

**Tipos de dados, variáveis, operações básicas de input-output, operadores básicos**

Neste módulo, aprenderá:

* como escrever e executar programas simples de Python;
* o que são literais, operadores, e expressões Python;
* o que são as variáveis e quais as regras que as regem;
* como realizar operações básicas de input e output.



**Hello, World!**

É tempo de começar a escrever algum **código Python real e funcional**. Vai ser muito simples por enquanto.

Como vamos mostrar-lhe alguns conceitos e termos fundamentais, estes snippets de código não serão sérios ou complexos.

Execute o código na janela do editor à direita. Se tudo correr bem aqui, verá a **linha de texto** na janela da consola.

Em alternativa, lançe o IDLE, crie um novo source file Python, preencha-o com este código, nomeie o ficheiro e guarde-o. Agora execute-o. Se tudo correr bem, verá o texto contido dentro das aspas na janela da consola IDLE. O código que executou deve parecer familiar. Viu algo muito semelhante quando o conduzimos através da criação do ambiente IDLE.

Agora vamos passar algum tempo a mostrar e a explicar-lhe o que está realmente a ver, e porque é que se parece com isto.

Como pode ver, o primeiro programa consiste nas seguintes partes:

* a palavra print;
* um parêntesis de abertura;
* umas aspas;
* uma linha de texto: Hello, World!;
* outras aspas;
* um parêntesis de fecho.

Cada um dos itens acima desempenha um papel muito importante no código.

**A função print()**

Veja a linha de código abaixo:

print("Hello, World!")

A palavra **print** que se pode ver aqui é um **nome de função**. Isso não significa que, onde quer que a palavra apareça, é sempre um nome de função. O significado da palavra vem do contexto em que a palavra foi usada.

Provavelmente já encontrou o termo função muitas vezes antes, durante as aulas de matemática. Provavelmente também pode listar vários nomes de funções matemáticas, como seno ou log.

As funções Python, no entanto, são mais flexíveis e podem conter mais conteúdo do que as suas irmãs matemáticas.

Uma função (neste contexto) é uma parte separada do código do computador capaz de:

* **causar um qualquer efeito** (por exemplo, enviar texto para o terminal, criar um ficheiro, desenhar uma imagem, reproduzir um som, etc.); isto é algo completamente inédito no mundo da matemática;
* **avaliar um valor** (por exemplo, a raiz quadrada de um valor ou o comprimento de um dado texto) e **devolvê-lo como o resultado da função**; é isto que faz as funções Python serem os parentes dos conceitos matemáticos.

Além disso, muitas das funções Python podem fazer as duas coisas acima juntamente.

De onde vêm as funções?

* Podem vir **do próprio Python**; a função print é uma deste tipo; tal função é um valor acrescentado recebido juntamente com o Python e o seu ambiente (é **incorporada**); não é necessário fazer nada de especial (por exemplo, perguntar a alguém por qualquer coisa) se quiser fazer uso dela;
* podem ser provenientes de um ou mais dos add-ons de Python chamados **módulos**; alguns dos módulos vêm com Python, outros podem requerer instalação separada - seja qual for o caso, todos eles precisam de estar explicitamente ligados ao seu código (mostrar-lhe-emos como fazê-lo em breve);
* pode **escrevê-los você mesmo**, colocando tantas funções quantas quiser e precisar dentro do seu programa para o tornar mais simples, mais claro e mais elegante.

O nome da função deve ser **significativo** (o nome da função print é evidente por si mesmo).

Claro que, se vai fazer uso de qualquer função já existente, não tem influência no seu nome, mas quando começar a escrever as suas próprias funções, deve considerar cuidadosamente a sua escolha de nomes.

**A função print()**

Como dissemos antes, uma função pode ter:

* um **efeito**;
* um **resultado**.

Há também uma terceira, muito importante, componente de função - o(s) **argumento(s)**.

As funções matemáticas normalmente aceitam um argumento, por exemplo, sen(x) toma um x, que é a medida de um ângulo.

As funções de Python, por outro lado, são mais versáteis. Dependendo das necessidades individuais, elas podem aceitar qualquer número de argumentos - tantos quantos forem necessários para desempenhar as suas tarefas. Nota: qualquer número inclui zero - algumas funções de Python não precisam de qualquer argumento.

print("Hello, World!")

Apesar do número de argumentos necessários/fornecidos, as funções Python exigem fortemente a presença de **um par de parêntesis** - de abertura e de fecho, respetivamente.

Se quiser entregar um ou mais argumentos a uma função, coloque-os **dentro dos parêntesis**. Se for utilizar uma função que não aceita qualquer argumento, ainda assim tem de ter os parêntesis.

Nota: para distinguir palavras comuns de nomes de funções, coloque **um par de parêntesis vazios** após os seus nomes, mesmo que a função correspondente queira um ou mais argumentos. Esta é uma convenção padrão.

A função de que estamos a falar aqui é print().

A função print() no nosso exemplo tem algum argumento?

Claro que sim, mas o que são eles?

# A função print()

O único argumento entregue à função print() neste exemplo é uma **string**:

print("Hello, World!")

Como pode ver, **a string é delimitada com aspas** - de facto, as aspas fazem a string - cortam uma parte do código e atribuem-lhe um significado diferente.

Pode imaginar que as aspas dizem algo como: o texto entre nós não é código. Não se destina a ser executado, e deve tomá-lo como está.

Quase tudo o que colocar dentro das aspas será tomado literalmente, não como código, mas como **dados**. Tente jogar com esta string em particular - modificá-la, introduzir algum conteúdo novo, apagar algum do conteúdo existente.

Há mais do que uma maneira de especificar uma string dentro do código Python, mas por agora, esta é suficiente.



Até agora, aprendeu sobre duas partes importantes do código: a função e a string. Falámos sobre elas em termos de sintaxe, mas agora é altura de os discutir em termos de semântica.

**A função print()**

O nome da função (***print*** neste caso) juntamente com os parêntesis e o(s) argumento(s), formam a **invocação da função**.

Discutiremos isto com mais profundidade em breve, mas devemos dar-lhe umas luzes de momento.

print("Hello, World!")

O que acontece quando o Python encontra uma invocação como esta abaixo?

function\_name(argument)

Vamos ver:

* Primeiro, o Python verifica se o nome especificado é **legal** (navega nos seus dados internos a fim de encontrar uma função existente com o mesmo nome; se esta pesquisa falhar, o Python aborta o código);
* segundo, o Python verifica se os requisitos da função para o número de argumentos **lhe permitem invocar** a função desta forma (por exemplo, se uma função específica exigir exatamente dois argumentos, qualquer invocação que apresente apenas um argumento será considerada errada, e abortará a execução do código);
* terceiro, o Python **deixa o seu código por um momento** e salta para a função que pretende invocar; claro, também leva o(s) seu(s) argumento(s) e passa-o(s) para a função;
* quarto, a função **executa o seu código**, causa o efeito desejado (se houver um), avalia o(s) resultado(s) desejado(s) (se existir(em)) e termina a sua tarefa;
* finalmente, o Python **regressa ao seu código** (ao local imediatamente após a invocação) e retoma a sua execução.

# LAB - A função print()

## Objetivos

* familiarizar-se com a função print() e as suas capacidades de formatação;
* experimentar com código Python.

### Cenário

O comando print() , que é uma das diretivas mais fáceis em Python, imprime simplesmente uma linha para o ecrã.

No seu primeiro lab:

* use a função print() para imprimir a linha Hello, Python! para o ecrã. Use aspas duplas à volta da string;
* tendo feito isso, utilize a função print() novamente, mas desta vez imprima o seu primeiro nome;
* remova as aspas duplas e execute o seu código. Veja a reação do Python. Que tipo de erro é lançado?
* depois, retire os parêntesis, volte a colocar as aspas duplas, e execute novamente o seu código. Que tipo de erro é lançado desta vez?
* experimente o máximo que puder. Altere aspas duplas com aspas simples, use várias funções print() na mesma linha e, em seguida, em linhas diferentes. Veja o que acontece.

**A função print()**

Três questões importantes têm de ser respondidas assim que possível:

**1. Qual é o efeito que a função**print()**causa?**

O efeito é muito útil e muito espetacular. A função:

* toma os seus argumentos (pode aceitar mais do que um argumento e pode também aceitar menos do que um argumento)
* converte-os numa forma legível para o ser humano, se necessário (como pode suspeitar, as strings não requerem esta ação, uma vez que a string já é legível)
* e **envia os dados resultantes para o dispositivo de output** (normalmente a consola); por outras palavras, qualquer coisa que coloque na função print() aparecerá no ecrã.

Não admira, então, que a partir de agora utilize print() muito intensivamente para ver os resultados das suas operações e avaliações.

**2. Que argumentos**print()**espera?**

Quaisquer. Mostrar-lhe-emos em breve que print() é capaz de operar com virtualmente todos os tipos de dados oferecidos pelo Python. Strings, números, carateres, valores lógicos, objetos - qualquer um destes pode ser passado com sucesso para print().

**3. Que valor é devolvido pela função**print()**?**

Nenhum. O seu efeito é suficiente.

**A função print() - instruções**

Já viu um programa de computador que contém uma invocação de função. Uma invocação de função é um dos muitos tipos possíveis de **instruções** Python.

É claro que qualquer programa complexo contém geralmente muito mais instruções do que uma. A questão é: como se acoplam mais do que uma instrução no código Python?

A sintaxe de Python é bastante específica nesta área. Ao contrário da maioria das linguagens de programação, o Python requer que **não haja mais do que uma instrução numa linha**.

Uma linha pode estar vazia (ou seja, pode não conter qualquer instrução) mas não deve conter duas, três ou mais instruções. Isto é estritamente proibido.

Nota: o Python faz uma exceção a esta regra - permite que uma instrução se espalhe por mais do que uma linha (o que pode ser útil quando o seu código contém construções complexas).

Vamos expandir um pouco o código, pode vê-lo no editor. Execute-o e anote o que vê na consola.

A sua consola Python deve agora ter este aspeto:

The itsy bitsy spider climbed up the waterspout.

Down came the rain and washed the spider out.

**output**

Esta é uma boa oportunidade para fazer algumas observações:

* o programa **invoca a função**print()**duas vezes**, e pode ver duas linhas separadas na consola - isto significa que print() começa o seu output a partir de uma nova linha cada vez que inicia a sua execução; pode alterar este comportamento, mas também pode utilizá-lo em seu proveito;
* cada invocação print() contém uma string diferente, uma vez que o seu argumento e o conteúdo da consola a refletem - isto significa que **as instruções no código são executadas pela mesma ordem** em que foram colocadas no source file; nenhuma instrução seguinte é executada até que a anterior esteja concluída (há algumas exceções a esta regra, mas pode ignorá-las por agora)

# A função print() - instruções

Mudámos um pouco o exemplo - acrescentámos uma invocação de função **vazia** print() . Chamamos-lhe vazia porque não apresentámos quaisquer argumentos para a função.

Pode vê-lo na janela do editor. Execute o código.

O que acontece?

Se tudo correr bem, deverá ver algo como isto:

The itsy bitsy spider climbed up the waterspout.

Down came the rain and washed the spider out.

**output**

Como pode ver, a invocação vazia print() não é tão vazia como se poderia esperar - produz uma linha vazia, ou (esta interpretação também é correta) o seu output é apenas uma newline.

Esta não é a única forma de produzir uma **newline** na consola de output. Vamos agora mostrar-lhe outra forma.

# A função print() - os carateres de escape e de newline

Modificámos novamente o código. Olhe com atenção.

Há duas mudanças muito subtis - inserimos um estranho par de carateres dentro da rima. Têm este aspeto: \n.

Curiosamente, enquanto **se pode ver dois carateres, o Python vê um.**

A barra invertida (\) tem um significado muito especial quando usado dentro de strings - a isto chama-se o **caratere de escape**.

A palavra *escape* deve ser entendida especificamente - significa que a série de carateres na string escapa por um momento (um momento muito curto) para introduzir uma inclusão especial.

Por outras palavras, a barra invertida não significa nada em si, mas é apenas uma espécie de anúncio de que o próximo caratere após a barra invertida também tem um significado diferente.

A letra n colocada após a barra invertida vem da palavra *newline* (nova linha).

Tanto a barra invertida como o *n* formam um símbolo especial chamado **um caratere de newline**, que incita a consola a iniciar uma **nova linha de output**.

Execute o código. A sua consola deve agora ter este aspeto:

The itsy bitsy spider

climbed up the waterspout.

Down came the rain

and washed the spider out.

**output**

Como pode ver, duas newlines aparecem na canção de embalar, nos locais onde as \n foram usadas.

# A função print() - os carateres de escape e de newline

Esta convenção tem duas consequências importantes:

1. Se quiser colocar apenas uma barra invertida dentro de uma string, não se esqueça da sua natureza de escape - tem de a duplicar, por exemplo, uma tal invocação causará um erro:

print("\")

enquanto esta não o fará:

print("\\")

2. Nem todos os pares de escape (a barra invertida acoplada a outro caratere) significam algo.

Experimente o seu código no editor, execute-o e veja o que acontece.

**A função print() - utilizar múltiplos argumentos**

Até agora, testámos o comportamento da função print() sem argumentos, e com um argumento. Também vale a pena tentar alimentar a função print() com mais do que um argumento.

Olhe para a janela do editor. Isto é o que vamos testar agora:

print("The itsy bitsy spider" , "climbed up" , "the waterspout.")

Há uma invocação da função print() , mas contém **três argumentos**. Todos eles são strings.

Os argumentos são **separados por vírgulas**. Cercamo-los de espaços para os tornar mais visíveis, mas não é realmente necessário, e não o faremos mais.

Neste caso, as vírgulas que separam os argumentos desempenham um papel completamente diferente da vírgula dentro da string. A primeira é uma parte da sintaxe de Python, a segunda destina-se a ser mostrada na consola.

Se voltar a olhar para o código, verá que não há espaços dentro das strings.

Execute o código e veja o que acontece.

A consola deve agora mostrar o seguinte texto:

The itsy bitsy spider climbed up the waterspout.

**output**

Os espaços, removidos das strings, apareceram novamente. Pode explicar porquê?

Duas conclusões emergem deste exemplo:

* a função print() invocada com mais do que um argumento **faz output de todos eles numa só linha**;
* a função print() **coloca um espaço entre os argumentos de output**, por sua própria iniciativa.

# A função print() - a forma posicional de passar os argumentos

Agora que sabe um pouco sobre print() costumes da função, vamos mostrar-lhe como alterá-los.

Deverá ser capaz de prever o output sem correr o código no editor.

A forma como estamos a passar os argumentos para a função print() é a mais comum em Python, e é chamada **a forma posicional** (este nome vem do facto de o significado do argumento ser ditado pela sua posição, por exemplo, o segundo argumento será produzido após o primeiro, e não o contrário).

Execute o código e verifique se o output corresponde às suas previsões.

**A função print() - os argumentos de keyword**

O Python oferece outro mecanismo para a passagem de argumentos, que pode ser útil quando se quer convencer a print() função a alterar um pouco o seu comportamento.

Não o vamos explicar em profundidade neste momento. Planeamos fazê-lo quando falarmos de funções. Por agora, queremos simplesmente mostrar-lhe como funciona. Sinta-se à vontade para o utilizar nos seus próprios programas.

O mecanismo é chamado **argumentos de keyword**. O nome deriva do facto de o significado destes argumentos ser retirado não da sua localização (posição) mas da palavra especial (keyword) utilizada para os identificar.

A função print() tem dois argumentos de keyword que pode usar para os seus propósitos. O primeiro deles é nomeado end.

Na janela do editor pode ver um exemplo muito simples da utilização de um argumento de keyword.

Para a sua utilização, é necessário conhecer algumas regras:

* um argumento de keyword consiste em três elementos: uma **keyword** identificando o argumento (end aqui); um **sinal de igual** (=); e um **valor** atribuído a esse argumento;
* qualquer argumento de keyword tem de ser colocado **após o último argumento posicional** (isto é muito importante)

No nosso exemplo, fizemos uso do end argumento de keyword, e definimo-lo para uma string contendo um espaço.

Execute o código para ver como ele funciona.

A consola deve agora mostrar o seguinte texto:

My name is Python. Monty Python.

**output**

Como pode ver, o argumento de keyword end determina os carateres que a função print() envia para o output, uma vez que atinge o final dos seus argumentos posicionais.

O comportamento padrão reflete a situação em que o argumento de keyword end é **implicitamente** usado da seguinte maneira: end="\n".

# A função print() - os argumentos de keyword

E agora é tempo de tentar algo mais difícil.

Se olhar cuidadosamente, verá que utilizámos o argumento end , mas a string atribuída a ele está vazia (não contém nenhum caratere).

O que vai acontecer agora? Execute o programa no editor para descobrir.

Como o argumento end foi definido para nada, a função print() também não produz nenhum output, uma vez que os seus argumentos posicionais foram esgotados.

A consola deve agora mostrar o seguinte texto:

My name is Monty Python.

**output**

Nota: **não foram enviadas newlines para o output**.

A string atribuída ao argumento de keyword end pode ter qualquer comprimento. Experimente-a se quiser.

# A função print() - os argumentos de keyword

Já dissemos anteriormente que a função print() separa os seus argumentos de output com espaços. Este comportamento também pode ser alterado.

O **argumento de keyword** que pode fazer isto é chamado sep (como *separador*).

Veja o código no editor, e execute-o.

O argumento sep fornece os seguintes resultados:

My-name-is-Monty-Python.

**output**

A função print() agora utiliza um traço, em vez de um espaço, para separar os argumentos de output.

Nota: o valor do argumento sep também pode ser uma string vazia. Experimente você mesmo.

# A função print() - os argumentos de keyword

Ambos os argumentos de keyword podem ser **misturados numa só invocação**, tal como aqui na janela do editor.

O exemplo não faz muito sentido, mas apresenta visivelmente as interações entre end e sep.

Consegue prever o output?

Execute o código e veja se corresponde às suas previsões.

Agora que compreende a função print() , está pronto a considerar como armazenar e processar dados em Python.

Sem print(), não seria capaz de ver nenhum resultado.

## LAB – A função print()

## Objetivos

* familiarizar-se com a função print() e as suas capacidades de formatação;
* experimentar com código Python.

## Cenário

Modifique a primeira linha de código no editor, usando as keywords sep e end , para corresponder ao output esperado. Use as duas print() funções no editor.

Não mude nada na segunda invocação print() .

## Output esperado

Programming\*\*\*Essentials\*\*\*in...Python

**output**

## LAB – Formatar o output

## Objetivos

* experimentar com o código Python existente;
* descobrir e corrigir erros básicos de sintaxe;
* familiarizar-se com a função print() e os seus recursos de formatação.

## Cenário

Encorajamo-lo vivamente a **brincar com o código** que escrevemos para si, e a fazer algumas (talvez mesmo destrutivas) alterações. Sinta-se livre para modificar qualquer parte do código, mas há uma condição - aprenda com os seus erros e tire as suas próprias conclusões.

Tente:

* minimizar o número de invocações da função print() inserindo a sequência \n nas strings
* fazer a seta duas vezes maior (mas mantendo as proporções)
* duplicar a seta, colocando ambas as setas lado a lado; nota: uma string pode ser multiplicada usando o seguinte truque: "string" \* 2 produzirá "stringstring" (brevemente, falaremos mais sobre o assunto)
* retire qualquer uma das aspas, e veja cuidadosamente a resposta do Python; preste atenção ao local onde o Python vê um erro - é este o local onde o erro realmente existe?
* faça o mesmo com alguns dos parêntesis;
* altere qualquer uma das print palavras por outra coisa, diferindo apenas no caso (por exemplo, Print) - o que acontece agora?
* substitua algumas das aspas por apóstrofes; observe cuidadosamente o que acontece.

# Key takeaways

1. A função print() é uma função **incorporada**. Imprime/faz output de uma mensagem especificada para a janela do ecrã/consola.

2. As funções incorporadas, ao contrário das funções definidas pelo utilizador, estão sempre disponíveis e não têm de ser importadas. O Python 3.8 vem com 69 funções incorporadas. Pode encontrar a sua lista completa fornecida em ordem alfabética na [Biblioteca Padrão Python](https://docs.python.org/3/library/functions.html).

3. Para chamar uma função (este processo é conhecido como **invocação de função** ou **chamada de função**), é necessário usar o nome da função seguido de parêntesis. Pode passar argumentos para uma função, colocando-os dentro dos parêntesis. Deve separar os argumentos com uma vírgula, por exemplo, print("Hello,", "world!"). Uma função “vazia” print() faz output de uma linha vazia para o ecrã.

4. As strings de Python são delimitadas com **aspas**, por exemplo, "I am a string" (aspas duplas), ou 'I am a string, too' (aspas simples).

5. Os programas de computador são coleções de **instruções**. Uma instrução é um comando para executar uma tarefa específica quando executada, por exemplo, para imprimir uma determinada mensagem no ecrã.

6. Em strings de Python a **barra invertida** (\) é um caratere especial que anuncia que o próximo caratere tem um significado diferente, por exemplo \n (the **newline character**) starts a new output line.

7. Os **argumentos posicionais** são aqueles cujo significado é ditado pela sua posição, por exemplo, o segundo argumento é apresentado após o primeiro, o terceiro é apresentado após o segundo, etc.

8. Os **argumentos de keyword** são aqueles cujo significado não é ditado pela sua localização, mas por uma palavra especial (keyword) utilizada para os identificar.

9. Os loops end e sep podem ser usados para formatar o output da função print() . O parâmetro sep especifica o separador entre os argumentos de output (por exemplo, o parâmetro print("H", "E", "L", "L", "O", sep="-"), enquanto o parâmetro end especifica o que imprimir no final da declaração print.



# Literais - os dados em si

Agora que tem um pouco de conhecimento de algumas das poderosas características oferecidas pela função print() , é hora de aprender sobre algumas novas questões, e um novo termo importante - o **literal**.

**Um literal são dados cujos valores são determinados pelo próprio literal**.

Como este é um conceito difícil de compreender, um bom exemplo pode ser útil.

Dê uma vista de olhos ao seguinte conjunto de dígitos:

123

Consegue adivinhar que valor representa? Claro que consegue - é *cento e vinte e três*.

Mas e quanto a isto:

c

Representa algum valor? Talvez. Pode ser o símbolo da velocidade da luz, por exemplo. Também pode ser a constante de integração. Ou mesmo o comprimento de uma hipotenusa no sentido de um teorema de Pitágoras. Existem muitas possibilidades.

Não se pode escolher o certo sem algum conhecimento adicional.

E esta é a pista: 123 é um literal, e c não é.

Utiliza literais **para codificar dados e para os colocar no seu código**. Vamos agora mostrar-lhe algumas convenções a que tem de obedecer quando utiliza o Python.

**Literais - os dados em si**

Comecemos com uma simples experiência - veja o snippet no editor.

A primeira linha parece familiar. A segunda parece estar errada devido à visível falta de aspas.

Tente executá-lo.

Se tudo correu bem, deverá agora ver duas linhas idênticas.

O que aconteceu? O que significa isto?

Através deste exemplo, depara-se com dois tipos diferentes de literais:

* uma **string**, que já conhece,
* e um número **inteiro**, algo completamente novo.

A função print() apresenta-os exatamente da mesma forma - este exemplo é óbvio, uma vez que a sua representação legível em termos humanos é também a mesma. Internamente, na memória do computador, estes dois valores são armazenados de formas completamente diferentes - a string existe apenas como uma string - uma série de letras.

O número é convertido em representação mecânica (um conjunto de bits). A função print() é capaz de mostrar ambos de uma forma legível para os seres humanos.

Vamos agora gastar algum tempo a discutir os literais numéricos e a sua vida interna.

**Inteiros**

Pode já saber um pouco sobre como os computadores efetuam cálculos sobre números. Talvez já tenha ouvido falar do **sistema binário**, e saiba que é o sistema que os computadores utilizam para armazenar números, e que eles podem realizar qualquer operação sobre eles.

Não vamos explorar aqui as complexidades dos sistemas de numeração posicional, mas vamos dizer que os números tratados pelos computadores modernos são de dois tipos:

* **inteiros**, isto é, aqueles que são desprovidos da parte fraccionada;
* e números **floating-point** (ou simplesmente **float**), que contêm (ou são capazes de conter) a parte fraccionada.

Esta definição não é inteiramente exata, mas suficiente por agora. A distinção é muito importante, e a fronteira entre estes dois tipos de números é muito rigorosa. Ambos os tipos de números diferem significativamente na forma como são armazenados na memória de um computador e no intervalo de valores aceitáveis.

A característica do valor numérico que determina o seu tipo, intervalo, e aplicação, é chamada **type**.

Se codificar um literal e o colocar dentro do código Python, a forma do literal determina a representação (type) que o Python utilizará para **o armazenar na memória**.

Por agora, vamos deixar de lado os números floating-point (voltaremos a eles em breve) e considerar a questão de como o Python reconhece os inteiros.

O processo é quase como escrevê-los com um lápis no papel - é simplesmente uma string de dígitos que compõem o número. Mas há uma reserva - não deve interpor quaisquer carateres que não sejam dígitos dentro do número.

Tomemos, por exemplo, o número onze milhões cento e onze mil cento e onze. Se pegasse agora mesmo num lápis, escreveria o número desta forma: 11,111,111, ou assim: 11.111.111, ou mesmo assim: 11 111 111.

É evidente que esta disposição facilita a leitura, especialmente quando o número é composto por muitos dígitos. No entanto, o Python não aceita coisas como estas. É **proibido**. O que o Python permite, no entanto, é a utilização de **underscores** em literais numéricos.\*

Por conseguinte, pode escrever este número desta forma: 11111111, ou assim: 11\_111\_111.

**NOTA** \*Python 3.6 introduziu underscores em letras numéricas, permitindo a colocação de underscores únicos entre dígitos e após especificadores de base para melhorar a legibilidade. Este recurso não está disponível em versões mais antigas de Python.

E como codificamos números negativos em Python? Como de costume - adicionando um **menos**. Pode escrever: -11111111, ou -11\_111\_111.

Os números positivos não precisam de ser precedidos pelo sinal de mais, mas é permitido, se o desejar fazer. As linhas a seguir descrevem o mesmo número: +11111111 e 11111111.

# Inteiros: números octais e hexadecimais

Há duas convenções adicionais em Python que são desconhecidas no mundo da matemática. A primeira permite-nos utilizar números numa representação **octal**.

Se um número inteiro for precedido por um 0O ou 0o prefixo (zero-o), ele será tratado como um valor octal. Isto significa que o número deve conter apenas dígitos retirados do intervalo [0..7].

0o123 é um número **octal** com um valor (decimal) igual a 83.

A classe print() faz a conversão automaticamente. Experimente isto:

print(0o123)

A segunda convenção permite-nos utilizar números **hexadecimais**. Estes números devem ser precedidos pelo prefixo 0x ou 0X (zero-x).

0x123 é um número **hexadecimal** com um valor (decimal) igual a 291. A função print() também pode gerir estes valores. Experimente isto:

print(0x123)

# Floats

Agora é altura de falar de outro tipo, que foi concebido para representar e armazenar os números que (como diria um matemático) têm uma **fração decimal não vazia**.

São os números que têm (ou podem ter) uma parte fracionada após o ponto decimal, e embora tal definição seja muito pobre, é certamente suficiente para o que desejamos discutir.

Sempre que utilizamos um termo como *dois e meio* ou *menos zero ponto quatro*, pensamos em números que o computador considera números de **floating-point**:

2.5

-0.4

Nota: *dois e meio* parece normal quando se escreve num programa, embora se a sua língua materna preferir usar uma vírgula em vez de um ponto no número, deve assegurar-se de que o seu **número não contém quaisquer vírgulas**.

O Python não aceitará isso, ou (em casos muito raros mas possíveis) pode interpretar mal as suas intenções, uma vez que a própria vírgula tem o seu significado reservado em Python.

Se quiser usar apenas um valor de dois e meio, deve escrevê-lo como mostrado acima. Nota mais uma vez - há um ponto entre *2* e *5* - não uma vírgula.

Como provavelmente pode imaginar, o valor de **zero ponto quatro** pode ser escrito em Python como:

0.4

Mas não se esqueça desta regra simples - pode omitir o zero quando é o único dígito em frente ou após o ponto decimal.

Em essência, pode escrever o valor 0.4 como:

.4

Por exemplo: o valor de 4.0 pode ser escrito como:

4.

Isto não mudará nem o seu tipo nem o seu valor.

**Ints vs. floats**

O ponto decimal é essencialmente importante no reconhecimento de números de floating-point em Python.

Veja estes dois números:

4

4.0

Pode pensar que eles são exatamente os mesmos, mas o Python vê-os de uma forma completamente diferente.

4 é um número **inteiro**, enquanto que 4.0 é um número **floating-point**.

O ponto é o que faz um float.

Por outro lado, não são apenas os pontos que fazem um float. Também pode utilizar a letra e.

Quando quiser usar números muito grandes ou muito pequenos, pode usar **notação científica**.

Tome, por exemplo, a velocidade da luz, expressa em *metros por segundo*. Escrito diretamente, ficaria assim: 300000000.

Para evitar escrever tantos zeros, os livros de física utilizam uma forma abreviada, que provavelmente já viu: 3 x 108.

Lê-se: três vezes dez à potência de oito.

Em Python, o mesmo efeito é conseguido de uma forma ligeiramente diferente - veja:

3E8

A letra E (também pode utilizar a letra minúscula e - vem da palavra **expoente**) é um registo conciso da frase *vezes dez à potência de*.

Nota:

* o **expoente** (o valor após o *E*) deve ser um número inteiro;
* a **base** (o valor à frente do *E*) pode ser um inteiro.

# Codificação de floats

Vejamos como esta convenção é utilizada para registar números que são muito pequenos (no sentido do seu valor absoluto, que está próximo de zero).

Uma constante física chamada *constante de Planck* (e denotada com um *h*), de acordo com os manuais escolares, tem o valor de: **6.62607 x 10-34**.

Se quiser utilizá-la num programa, deve escrevê-la desta forma:

6.62607E-34

Nota: o facto de ter escolhido uma das formas possíveis de codificação de valores float não significa que o Python o apresente da mesma forma.

O Python pode, por vezes, escolher uma **notação diferente** da sua.

Por exemplo, digamos que decidiu usar o seguinte float literal:

0.0000000000000000000001

Quando executa este literal através do Python:

print(0.0000000000000000000001)

este é o resultado:

1e-22

**output**

O Python escolhe sempre **a forma mais económica de apresentação do número**, e deve ter isto em consideração ao criar literais.

# Strings

Strings são utilizadas quando é necessário processar texto (como nomes de todos os tipos, endereços, romances, etc.), não números.

Já sabe um pouco sobre elas. Por exemplo, que as **strings precisam de aspas** da mesma forma que floats precisam de pontos.

Esta é uma string muito típica: "I am a string."

No entanto, há um senão. O senão é como codificar uma aspa dentro de uma string que já está delimitada por aspas.

Vamos supor que queremos imprimir uma mensagem muito simples dizendo:

I like "Monty Python"

Como fazemos isto sem gerar um erro? Existem duas soluções possíveis.

A primeira baseia-se no conceito que já conhecemos do **caratere de escape**, que como se deve lembrar é representado pela **barra invertida**. A barra invertida também pode escapar às aspas. Uma aspa precedida por uma barra invertida muda o seu significado - não é um delimitador, mas apenas uma aspa. Isto funcionará como pretendido:

print("I like \"Monty Python\"")

Nota: existem duas aspas escapadas dentro da string - consegue vê-las?

A segunda solução pode ser um pouco surpreendente. O Python pode utilizar **uma apóstrofe em vez de uma aspa**. Qualquer um destes carateres pode delimitar strings, mas deve ser **consistente**.

Se abrir uma string com uma aspa, tem de fechá-la com uma aspa.

Se começar uma string com uma apóstrofe, tem de a acabar com uma apóstrofe.

Este exemplo também funcionará:

print('I like "Monty Python"')

Nota: não precisa de fazer nenhum escape aqui.

# Codificar strings

Agora, a questão seguinte é: como incorporar uma apóstrofe numa string colocada entre apóstrofes?

Já deve saber a resposta, ou, para ser mais preciso, as duas respostas possíveis.

Tente imprimir uma string contendo a seguinte mensagem:

I'm Monty Python.

Sabe como fazê-lo? Clique em *Verifique* em baixo para ver se estava certo:

Verifique

Como pode ver, a barra invertida é uma ferramenta muito poderosa - pode escapar não só às aspas, mas também às apóstrofes.

Já o mostrámos, mas queremos enfatizar mais uma vez este fenómeno - **uma string pode estar vazia** - pode não conter quaisquer carateres.

Uma string vazia ainda assim permanece uma string:

''

""

**Valores Booleanos**

Para concluir com os literais de Python, existem mais dois.

Não são tão óbvios como os anteriores, uma vez que são usados para representar um valor muito abstrato - **truthfulness** (veracidade).

Cada vez que pergunta ao Python se um número é maior que outro, a pergunta resulta na criação de alguns dados específicos - um valor **Booleano**.

O nome vem de George Boole (1815-1864), autor da obra fundamental, *As Leis do Pensamento*, que contém a definição de **álgebra Booleana** - uma parte da álgebra que faz uso de apenas dois valores distintos: True e False, denotado como 1 e 0.

Um programador escreve um programa, e o programa faz perguntas. O Python executa o programa, e fornece as respostas. O programa deve ser capaz de reagir de acordo com as respostas recebidas.

Felizmente, os computadores conhecem apenas dois tipos de respostas:

* Sim, isto é verdade;
* Não, isto é falso.

Nunca obterá uma resposta como: *Não sei* ou *Provavelmente sim, mas não sei ao certo*.

O Python, então, é um réptil **binário**.

Estes dois valores Booleanos têm denotações rigorosas em Python:

True

False

Não se pode mudar nada - é preciso tomar estes símbolos tal como eles são, incluindo **case-sensitivity**.

Desafio: Qual será o output do seguinte snippet de código?

print(True > False)

print(True < False)

Execute o código na Sandbox para verificar. Consegue explicar o resultado?

## LAB – Literais em Python - Strings

## Objetivos

* familiarizar-se com a função print() e as suas capacidades de formatação;
* praticar codificação de strings;
* experimentar com código Python.

## Cenário

Escreva um código de uma linha, utilizando a função print() , bem como os carateres de newline e de escape, para corresponder ao resultado esperado emitido em três linhas.

## Output esperado

"I'm"

""learning""

"""Python"""

**output**

# Key takeaways

1. Os **literais** são notações para representar alguns valores fixos em código. O Python tem vários tipos de literais - por exemplo, um literal pode ser um número (literais numéricos, por exemplo, 123), ou uma string (literais de string, por exemplo, “Eu sou um literal.“).

2. O **sistema binário** é um sistema de números que emprega *2* como base. Portanto, um número binário é composto apenas por 0s e 1s, por exemplo, 1010 é *10* em decimal.

Os sistemas de numeração octal e hexadecimal, do mesmo modo, empregam *8* e *16* como suas bases, respetivamente. O sistema hexadecimal utiliza os números decimais e seis letras extra.

3. **Inteiros** (ou simplesmente **int**s) são um dos tipos numéricos suportados pelo Python. São números escritos sem um componente fracionário, por exemplo, 256, ou -1 (inteiros negativos).

4. Números de **floating-point** (ou simplesmente **float**s) são outro dos tipos numéricos suportados pelo Python. São números que contêm (ou são capazes de conter) um componente fracionário, por exemplo 1.27.

5. Para codificar uma apóstrofe ou uma aspa dentro de uma string, pode usar o caratere de escape, por exemplo, 'I\'m happy.', ou abrir e fechar a string utilizando um conjunto de símbolos opostos aos que deseja codificar, por exemplo "I'm happy." codificar uma apóstrofe, e 'He said "Python", not "typhoon"' para codificar umas aspas (duplas).

6. **Valores booleanos** são os dois objetos constantes True e False usado para representar valores de verdade (em contextos numéricos 1 é True, enquanto 0 é False.

**EXTRA**

Há mais um literal especial que é usado em Python: o literal None . Este literal é um chamado NoneType objeto, e é utilizado para representar **a ausência de um valor**. Em breve, contar-lhe-emos mais sobre isso.

**Exercício 1**

Que tipos de literais são os dois exemplos seguintes?

"Hello ", "007"  
Verifique

**Exercício 2**

Que tipos de literais são os quatro exemplos seguintes?

"1.5", 2.0, 528, False  
Verifique

**Exercício 3**

Qual é o valor decimal do seguinte número binário?

1011  
Verifique

# Python como uma calculadora

Agora, vamos mostrar-lhe um lado completamente novo da função print(). Já sabe que a função é capaz de lhe mostrar os valores dos literais que lhe são transmitidos pelos argumentos.

Na verdade, pode fazer algo mais. Dê uma vista de olhos no snippet:

print(2+2)

Digite novamente o código no editor e execute-o. Consegue adivinhar o output?

Deve ver o número quatro. Sinta-se à vontade para experimentar com outros operadores.

Sem levar isto demasiado a sério, acabou de descobrir que o Python pode ser usado como calculadora. Não é muito útil, e definitivamente não é de bolso, mas é uma calculadora.

Levando-o mais a sério, estamos agora a entrar na província dos **operadores** e das **expressões**.

# Operadores básicos

Um **operador** é um símbolo da linguagem de programação, que é capaz de operar sobre os valores.

Por exemplo, tal como na aritmética, o sinal + (mais) é o operador que é capaz de **adicionar** dois números, dando o resultado da adição.

Mas nem todos os operadores Python são tão óbvios como o sinal de mais, por isso vamos analisar alguns dos operadores disponíveis em Python, e explicaremos que regras regem a sua utilização, e como interpretar as operações que realizam.

Começaremos pelos operadores que estão associados às operações aritméticas mais amplamente reconhecidas:

+, -, \*, /, //, %, \*\*

A ordem do seu aparecimento não é acidental. Falaremos mais sobre o assunto depois de termos passado por todos eles.

**Lembre-se**: Os dados e os operadores, quando ligados entre si, formam **expressões**. A expressão mais simples é um literal em si.

**Operadores aritméticos: exponenciação**

Um sinal \*\* (duplo asterisco) é um operador de **exponenciação** (potência). O seu argumento esquerdo é a **base**, o seu direito, o **expoente**.

A matemática clássica prefere a notação com sobrescrito, tal como esta: **23**. Os editores de texto puro não aceitam isso, por isso o Python usa \*\* ao invés, por exemplo, 2 \*\* 3.

Dê uma vista de olhos aos nossos exemplos na janela do editor.

Nota: rodeámos os asteriscos duplos com espaços nos nossos exemplos. Não é obrigatório, mas melhora a **legibilidade** do código.

Os exemplos mostram uma característica muito importante de praticamente todos os **operadores numéricos** de Python.

Execute o código e veja cuidadosamente os resultados que produz. Consegue ver alguma regularidade aqui?

**Lembre-se**: É possível formular as seguintes regras com base neste resultado:

* quando **ambos** os argumentos \*\* são inteiros, o resultado é também um inteiro;
* quando **pelo menos um** argumento \*\* é um float, o resultado é também um float.

Esta é uma distinção importante a lembrar.

# Operadores aritméticos: multiplicação

Um sinal \* (asterisco) é um**operador de multiplicação**.

Execute o código abaixo e verifique se a nossa regra de *inteiro vs. float* ainda está a funcionar.

print(2 \* 3)

print(2 \* 3.)

print(2. \* 3)

print(2. \* 3.)

# Operadores aritméticos: divisão

O sinal / (barra) é um **operador de divisão**.

O valor em frente da barra é um **dividendo**, o valor por detrás da barra, um **divisor**.

Execute o código abaixo e analise os resultados.

print(6 / 3)

print(6 / 3.)

print(6. / 3)

print(6. / 3.)

Deve ver que existe uma exceção à regra.

**O resultado produzido pelo operador da divisão é sempre um float**, independentemente de o resultado parecer ou não ser um float à primeira vista: 1 / 2, ou se se parecer com um inteiro puro: 2 / 1.

Isto é um problema? Sim, é. Acontece por vezes que é realmente necessária uma divisão que forneça um valor inteiro, não um float.

Felizmente, o Python pode ajudá-lo com isso.

**Operadores aritméticos: divisão inteira**

Um sinal // (dupla barra) é um operador de **divisão inteira**. Difere do operador padrão / em dois detalhes:

* o seu resultado não tem a parte fracionada - está ausente (para inteiros), ou é sempre igual a zero (para floats); isto significa que **os resultados são sempre arredondados**;
* está em conformidade com a *regra inteiro vs. float*.

Execute o exemplo abaixo e veja os resultados:

print(6 // 3)

print(6 // 3.)

print(6. // 3)

print(6. // 3.)

Como se pode ver, a *divisão inteiro por inteiro* dá um **resultado inteiro**. Todos os outros casos produzem floats.

Vamos fazer alguns testes mais avançados.

Veja o seguinte snippet:

print(6 // 4)

print(6. // 4)

Imagine que usámos / em vez de // - consegue prever os resultados?

Sim, seria 1.5 em ambos os casos. Isso é claro.

Mas que resultados devemos esperar com // divisão?

Execute o código e veja por si mesmo.

O que obtemos são dois uns - um inteiro e um float.

O resultado da divisão inteira é sempre arredondado para o valor inteiro mais próximo, que é inferior ao resultado real (não arredondado).

Isto é muito importante: **o arredondamento vai sempre para o número inteiro menor**.

Veja o código abaixo e tente prever os resultados mais uma vez:

print(-6 // 4)

print(6. // -4)

Nota: alguns dos valores são negativos. Isto irá obviamente afetar o resultado. Mas como?

O resultado são dois dois negativos. O resultado real (não arredondado) é -1.5 em ambos os casos. No entanto, os resultados são sujeitos a arredondamento. O **arredondamento vai para o menor valor inteiro**, e o menor valor inteiro é -2, logo: -2 e -2.0.

**NOTA**

A divisão inteira também pode ser chamada **floor division**. Definitivamente, no futuro, deparar-se-á com este termo.

**Operadores: remainder (modulo)**

O próximo operador é bastante peculiar, visto não ter equivalente entre os operadores aritméticos tradicionais.

A sua representação gráfica em Python é o sinal % (percentagem), o que pode parecer um pouco confuso.

Tente pensar nisto como uma barra (operador de divisão) acompanhada por dois pequenos círculos engraçados.

O resultado do operador é um **remainder (resto) deixado após a divisão inteira**.

Por outras palavras, é o valor que sobrou depois de dividir um valor por outro para produzir um quociente inteiro.

Nota: o operador às vezes é chamado **modulo** noutras linguagens de programação.

Dê uma vista de olhos no snippet - tente prever o seu resultado e, em seguida, execute-o:

print(14 % 4)

Como pode ver, o resultado é dois. Esta é a razão:

* 14 // 4 dá 3 → este é o **quociente** inteiro;
* 3 \* 4 dá 12 → como resultado da **multiplicação de quocientes e divisores**;
* 14 - 12 dá 2 → este é o **resto**.

Este exemplo é um pouco mais complicado:

print(12 % 4.5)

Qual é o resultado?

Verifique

**Operadores: como não dividir**

Como provavelmente sabe, a **divisão por zero não funciona**.

**Não** tente:

* executar uma divisão por zero;
* executar uma divisão inteira por zero;
* encontrar um remainder de uma divisão por zero.

# Operadores: adição

O operador de **adição** é o sinal + (mais), que está totalmente de acordo com os padrões matemáticos.

Novamente, dê uma vista de olhos no snippet do programa em baixo:

print(-4 + 4)

print(-4. + 8)

O resultado não deve ser nada surpreendente. Execute o código para o verificar.

# O operador de subtração, operadores unários e binários

O operador de **subtração** é obviamente o sinal - (menos), embora deva notar que este operador também tem outro significado - **ele pode alterar o sinal de um número**.

Esta é uma grande oportunidade para apresentar uma distinção muito importante entre operadores **unários** e **binários**.

Em aplicações de subtração, o **operador menos espera dois argumentos**: o da esquerda (um **minuendo** em termos aritméticos) e o da direita (um **subtraendo**).

Por esta razão, o operador de subtração é considerado um dos operadores binários, assim como os operadores de adição, multiplicação e divisão.

Mas o operador menos pode ser usado de uma forma diferente (unária) - veja a última linha do snippet em baixo:

print(-4 - 4)

print(4. - 8)

print(-1.1)

A propósito: há também um operador + unário. Pode utilizá-lo assim:

print(+2)

O operador preserva o sinal de seu único argumento - o correto.

Embora tal construção seja sintaticamente correta, a sua utilização não faz muito sentido, e seria difícil encontrar uma boa razão para o fazer.

Dê uma vista de olhos no snippet acima - consegue adivinhar o seu output?

**Os operadores e as suas prioridades**

Até agora, temos tratado cada operador como se não tivesse qualquer ligação com os outros. Obviamente, uma situação tão ideal e simples é uma raridade na programação real.

Além disso, encontrará muito frequentemente mais do que um operador numa só expressão, e então esta presunção já não é tão óbvia.

Considere a seguinte expressão:

2 + 3 \* 5

Provavelmente lembra-se da escola que **as multiplicações precedem as adições**.

Deve certamente lembrar-se que primeiro deve multiplicar 3 por 5 e, mantendo o 15 na sua memória, depois adicioná-los a 2, obtendo assim o resultado de 17.

O fenómeno que leva alguns operadores a agir antes de outros é conhecido como **a hierarquia de prioridades**.

O Python define com precisão as prioridades de todos os operadores, e assume que os operadores de maior (mais alta) prioridade realizam as suas operações antes dos operadores de menor prioridade.

Portanto, se sabe que \* tem uma prioridade maior do que +, o cálculo do resultado final deve ser óbvio.

**Os operadores e as suas ligações**

A **ligação** do operador determina a ordem dos cálculos efetuados por alguns operadores com igual prioridade, colocados lado a lado numa só expressão.

A maioria dos operadores de Python têm ligação do lado esquerdo, o que significa que o cálculo da expressão é realizado da esquerda para a direita.

Este exemplo simples mostrar-lhe-á como funciona. Veja:

print(9 % 6 % 2)

Há duas formas possíveis de avaliar esta expressão:

* da esquerda para a direita: primeiro 9 % 6 dá 3e, em seguida, 3 % 2 dá 1;
* da direita para a esquerda: primeiro 6 % 2 dá 0e, em seguida, 9 % 0 causa **um erro fatal**.

Execute o exemplo e veja o que obtém.

O resultado deve ser 1. Este operador tem **ligação do lado esquerdo**. Mas há uma exceção interessante.

**Operadores e seus bindings: exponenciação**

Repita a experiência, mas agora com exponenciação.

Use este snippet de código:

print(2 \*\* 2 \*\* 3)

Os dois resultados possíveis são:

* 2 \*\* 2 → 4; 4 \*\* 3 → 64
* 2 \*\* 3 → 8; 2 \*\* 8 → 256

Execute o código. O que vê?

O resultado mostra claramente que **o operador de exponenciação utiliza a ligação do lado direito**.

# Lista de prioridades

Uma vez que é novo nos operadores Python, não queremos apresentar neste momento a lista completa de prioridades dos operadores.

Em vez disso, vamos mostrar-lhe a sua forma truncada, e vamos expandi-la de forma consistente à medida que introduzimos novos operadores.

Veja a tabela abaixo:

|  |  |  |
| --- | --- | --- |
| **Prioridade** | **Operador** |  |
| 1 | +, - | unário |
| 2 | \*\* |  |
| 3 | \*, /, //, % |  |
| 4 | +, - | binário |

Nota: enumerámos os operadores por ordem **das prioridades mais altas (1) até às mais baixas (4)**.

Tente trabalhar com a seguinte expressão:

print(2 \* 3 % 5)

Ambos os operadores (\* e %) têm a mesma prioridade, pelo que o resultado só pode ser adivinhado quando se conhece a direção de ligação. Como pensa? Qual é o resultado?

Verifique

# Operadores e parêntesis

Claro, é sempre permitido utilizar **parêntesis**, o que pode alterar a ordem natural de um cálculo.

De acordo com as regras aritméticas, as **subexpressões entre parêntesis são sempre calculadas em primeiro lugar**.

Pode-se usar tantos parêntesis quantos forem necessários, e são frequentemente usados para **melhorar a legibilidade** de uma expressão, mesmo que não alterem a ordem das operações.

Um exemplo de uma expressão com vários parêntesis é este:

print((5 \* ((25 % 13) + 100) / (2 \* 13)) // 2)

Tente calcular o valor que é impresso para a consola. Qual é o resultado da função print() ?

Verifique

**Key takeaways**

1. Uma **expressão** é uma combinação de valores (ou variáveis, operadores, chamadas a funções - em breve aprenderá sobre elas) que avalia a um valor, por exemplo, 1 + 2.

2. Os **operadores** são símbolos especiais ou keywords capazes de operar sobre os valores e realizar operações (matemáticas), por exemplo, o \* operador multiplica dois valores: x \* y.

3. Operadores aritméticos em Python: + (adição), - (subtração), \* (multiplicação), / (divisão clássica - devolve sempre um float), % (módulo - divide o operando esquerdo pelo operando direito e devolve o resto da operação, por exemplo, 5 % 2 = 1), \*\* (exponenciação - operando esquerdo elevado à potência do operando direito, por exemplo, 2 \*\* 3 = 2 \* 2 \* 2 = 8), // (divisão por piso/inteiro - devolve um número resultante da divisão, mas arredondado para baixo para o número inteiro mais próximo, por exemplo 3 // 2.0 = 1.0)

4. Um operador **unário** é um operador com apenas um operando, por exemplo, -1, ou +3.

5. Um operador **binário** é um operador com dois operandos, por exemplo, 4 + 5, ou 12 % 5.

6. Alguns operadores atuam antes de outros - **a hierarquia de prioridades**:

* unário + e - têm a prioridade mais alta
* depois: \*\*, depois: \*, /, e %e, depois, a prioridade mais baixa: binário + e -.

7. Subexpressões **entre parêntesis** são sempre calculadas em primeiro lugar, por exemplo, 15 - 1 \* (5 \* (1 + 2)) = 0.

8. O operador de **exponenciação** utiliza **ligação do lado direito**, por exemplo 2 \*\* 2 \*\* 3 = 256.

**Exercício 1**

Qual é o output do seguinte snippet?

print((2 \*\* 4), (2 \* 4.), (2 \* 4))

Verifique

**Exercício 2**

Qual é o output do seguinte snippet?

print((-2 / 4), (2 / 4), (2 // 4), (-2 // 4))

Verifique

**Exercício 3**

Qual é o output do seguinte snippet?

print((2 % -4), (2 % 4), (2 \*\* 3 \*\* 2))

# O que são variáveis?

Parece bastante óbvio que o Python deveria permitir-lhe codificar literais transportando valores de número e texto.

Já sabe que pode fazer algumas operações aritméticas com estes números: adicionar, subtrair, etc. Fá-lo-á muitas vezes.

Mas é uma questão bastante normal de perguntar como **armazenar os resultados** destas operações, a fim de os utilizar noutras operações, e assim por diante.

Como guardar os resultados intermediários, e utilizá-los novamente para produzir os resultados subsequentes?

O Python irá ajudá-lo nisso. Oferece "caixas" especiais (em inglês, containers) para esse fim, e estas caixas são chamadas **variáveis** - o próprio nome sugere que o conteúdo destes containers pode ser variado em (quase) qualquer forma.

O que tem cada variável Python?

* um nome;
* um valor (o conteúdo do container)

Comecemos com as questões relacionadas com o nome de uma variável.

As variáveis não aparecem automaticamente num programa. Como programador, deve decidir quantas e quais as variáveis a utilizar nos seus programas.

Deve também nomeá-las.

Se quiser **dar um nome a uma variável**, deve seguir algumas regras estritas:

* o nome da variável deve ser composto por letras maiúsculas ou minúsculas, dígitos e o caratere \_ (underscore)
* o nome da variável deve começar com uma letra;
* o caratere underscore é uma letra;
* letras maiúsculas e minúsculas são tratadas como diferentes (um pouco diferente do que no mundo real - *Alice* e *ALICE* são os mesmos nomes próprios, mas em Python são dois nomes de variáveis diferentes, e consequentemente, duas variáveis diferentes);
* o nome da variável não deve ser nenhuma das palavras reservadas de Python (as keywords - explicaremos mais sobre isto em breve).



**Nomes de variável corretos e incorretos**

Note-se que as mesmas restrições se aplicam a nomes de funções.

O Python não impõe restrições ao comprimento dos nomes de variáveis, mas isto não significa que um nome de variável longo seja sempre melhor do que um curto.

Aqui estão alguns nomes de variáveis corretos, mas nem sempre convenientes:

MyVariable, i, t34, Exchange\_Rate, counter, days\_to\_christmas, TheNameIsSoLongThatYouWillMakeMistakesWithIt, \_.

Além disso, o Python permite-lhe utilizar não só letras latinas mas também carateres específicos de línguas que utilizam outros alfabetos.

Estes nomes de variáveis também estão corretos:

Adiós\_Señora, sûr\_la\_mer, Einbahnstraße, переменная.

E agora alguns **nomes incorretos**:

10t (não começa com uma letra), Exchange Rate (contém um espaço)

**NOTA**

O [PEP 8 — Style Guide for Python Code](https://www.python.org/dev/peps/pep-0008/) recomenda a seguinte convenção de nomenclatura para variáveis e funções em Python:

* os nomes de variáveis devem ser minúsculas, com palavras separadas por underscores para melhorar a legibilidade (por exemplo, var, my\_variable)
* os nomes de funções seguem a mesma convenção que os nomes de variáveis (por exemplo, fun, my\_function)
* também é possível utilizar um caso misto (por exemplo, myVariable), mas apenas em contextos onde esse já é o estilo predominante, para manter a retrocompatibilidade com a convenção adotada.

**Keywords**

Dê uma vista de olhos na lista de palavras que desempenham um papel muito especial em todos os programa Python.

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

São chamadas **keywords** ou (mais precisamente) **reserved keywords**. (keywords reservadas) São reservadas porque **não deve utilizá-las como nomes**: nem para as suas variáveis, nem para as suas funções, nem para quaisquer outras entidades nomeadas que queira criar.

O significado da palavra reservada é **predefinido**, e não deve ser alterado de forma alguma.

Felizmente, devido ao facto do Python ser sensível a maiúsculas e minúsculas, pode modificar qualquer uma destas palavras alterando qualquer letra como maiúscula ou minúscula, criando assim uma nova palavra que já não está reservada.

Por exemplo - **não pode nomear** a sua variável desta forma:

import

Não se deve ter uma variável assim nomeada - é proibido. Mas, em vez disso, pode fazer isto:

Import

Estas palavras podem ser um mistério para si agora, mas em breve aprenderá o seu significado.

**Criar variáveis**

O que se pode colocar dentro de uma variável?

Qualquer coisa.

Pode utilizar uma variável para armazenar qualquer valor de qualquer um dos tipos já apresentados, e muitos mais dos que ainda não lhe mostrámos.

O valor de uma variável é o que lhe foi atribuído. Pode variar com a frequência que precisar ou desejar. Pode ser um inteiro num momento, e um float um momento a seguir, acabando por se tornar uma string.

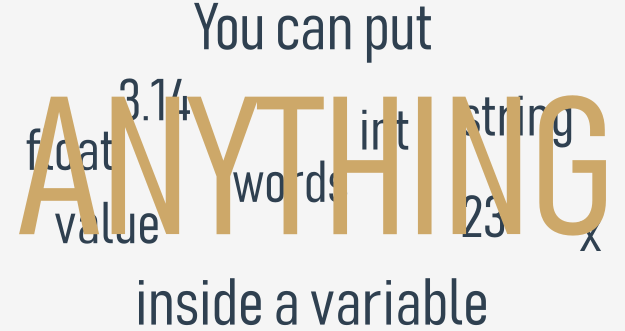
Falemos agora de duas coisas importantes - **como as variáveis são criadas**, e **como colocar valores dentro delas** (ou melhor - como dar ou **passar valores** a elas).

**LEMBRE-SE**

**Uma variável passa a existir como o resultado da atribuição de um valor a ela**. Ao contrário de outras linguagens, não precisa de a declarar de nenhuma forma especial.

Se atribuir qualquer valor a uma variável inexistente, a variável será **automaticamente criada**. Não precisa de fazer mais nada.

A criação (ou de outra forma - a sua sintaxe) é extremamente simples: **basta usar o nome da variável desejada, depois o sinal de igual (=) e o valor que se pretende colocar na variável.**



Dê uma vista de olhos no snippet:

var = 1

print(var)

Consiste em duas instruções simples:

* A primeira delas cria uma variável chamada var, e atribui um literal com um valor inteiro igual a 1.
* A segunda imprime o valor da variável recentemente criada para a consola.

Nota: print() tem ainda outro lado - também pode tratar de variáveis. Sabe qual será o output do snippet?

# Utilização de variáveis

É-lhe permitido utilizar tantas declarações de variáveis quantas forem necessárias para atingir o seu objetivo, como esta:

var = 1

account\_balance = 1000.0

client\_name = 'John Doe'

print(var, account\_balance, client\_name)

print(var)

**Não lhe é permitido utilizar uma variável que não existe** (por outras palavras, uma variável à qual não foi atribuído um valor).

Este exemplo irá **causar um erro**:

var = 1

print(Var)

Tentámos usar uma variável chamada Var, que não tem qualquer valor (nota: var e Var são entidades diferentes, e não têm nada em comum no que diz respeito ao Python).

**LEMBRE-SE**

Pode utilizar a função print() e combinar texto e variáveis usando o operador + para fazer output de strings e variáveis, por exemplo:

var = "3.8.5"

print("Python version: " + var)

Consegue adivinhar o output do snippet acima?

# Atribuição de um novo valor a uma variável já existente

Como se atribui um novo valor a uma variável já criada? Da mesma forma. Basta usar o sinal de igualdade.

O sinal de igualdade é, de facto, um **operador de atribuição**. Embora isto possa parecer estranho, o operador tem uma sintaxe simples e uma interpretação inequívoca.

Atribui o valor do seu argumento direito para a esquerda, enquanto que o argumento direito pode ser uma expressão arbitrariamente complexa envolvendo literais, operadores e variáveis já definidas.

Veja o código abaixo:

var = 1

print(var)

var = var + 1

print(var)

O código envia duas linhas para a consola:

1

2

**output**

A primeira linha do snippet **cria uma nova variável** chamada var e atribui 1 a ela.

A declaração diz: atribui um valor de 1 a uma variável chamada var.

Podemos dizê-lo duma forma mais curta: atribui 1 até var.

Alguns preferem ler uma declaração como esta: var torna-se 1.

A terceira linha **atribui a mesma variável com o novo valor** retirado da própria variável, somado com 1. Ao ver um registo como este, um matemático provavelmente protestaria - nenhum valor pode ser igual a si mesmo mais um. Isto é uma contradição. Mas o Python trata o sinal = não como *igual a*, mas como *atribuir um valor*.

Então, como se lê um tal registo no programa?

Tome o valor atual da variável var, adicione-lhe 1 e armazene o resultado na variável var.

Na verdade, o valor da variável var foi **incrementado** por um, o que nada tem a ver com a comparação da variável com qualquer valor.

Sabe qual será o output do seguinte snippet?

var = 100

var = 200 + 300

print(var)

# Resolução de problemas matemáticos simples

Agora deverá ser capaz de construir um programa curto que resolva problemas matemáticos simples, como o teorema de Pitágoras:

*O quadrado da hipotenusa é igual à soma dos quadrados dos catetos.*

O código seguinte avalia o comprimento da hipotenusa (ou seja, o lado mais longo de um triângulo retângulo, o oposto ao ângulo reto) usando o teorema de Pitágoras:

a = 3.0

b = 4.0

c = (a \*\* 2 + b \*\* 2) \*\* 0.5

print("c =", c)

Nota: precisamos de fazer uso do operador \*\* para avaliar a raiz quadrada como:

√(x) = x(½)

e

c = √a2 + b2

Consegue adivinhar o output do código?

Verifique abaixo e execute o código no editor para confirmar as suas previsões.

## LAB - Variáveis

## Objetivos

* familiarizar-se com o conceito de armazenar e trabalhar com diferentes tipos de dados em Python;
* experimentar com código Python.

## Cenário

Aqui está um conto:

Uma vez em Appleland, o John tinha três maçãs, a Mary tinha cinco maçãs, e o Adam tinha seis maçãs. Todos eles foram muito felizes e viveram durante muito tempo. Fim da história.

A sua tarefa é:

* criar as variáveis: john, mary, e adam;
* atribuir valores às variáveis. Os valores devem ser iguais ao número de frutos possuídos pelo John, pela Mary e pelo Adam, respetivamente;
* tendo armazenado os números nas variáveis, imprimir as variáveis numa linha, e separar cada uma delas com uma vírgula;
* criar agora uma nova variável chamada total\_apples igual à adição das três variáveis anteriores.
* imprimir o valor armazenado em total\_apples para a consola;
* **experimente com o seu código**: crie novas variáveis, atribua-lhes valores diferentes e efetue várias operações aritméticas sobre elas (por exemplo, +, -, -, \*, /, //, etc.). Tente imprimir uma string e um inteiro juntos numa linha, por exemplo, "Total number of apples:" e total\_apples.

# Operadores de atalho

Chegou a hora do próximo conjunto de operadores que facilitam a vida de um programador.

Muito frequentemente, queremos utilizar uma e a mesma variável tanto para o lado direito como para o esquerdo do = operador.

Por exemplo, se precisarmos de calcular uma série de valores sucessivos de potências de 2, podemos utilizar uma peça como esta:

x = x \* 2

Pode usar uma expressão como esta se não conseguir adormecer e estiver a tentar lidar com ela usando alguns métodos bons e antiquados:

sheep = sheep + 1

O Python oferece-lhe uma forma abreviada de escrever operações como estas, que podem ser codificadas como se segue:

x \*= 2

sheep += 1

Vamos tentar apresentar uma descrição geral para estas operações.

Se op é um operador de dois argumentos (esta é uma condição muito importante) e o operador é utilizado no seguinte contexto:

variable = variable op expression

Pode ser simplificado e mostrado da seguinte forma:

variable op= expression

Dê uma vista de olhos nos exemplos abaixo. Certifique-se de que os compreende a todos.

i = i + 2 \* j ⇒ i += 2 \* j

var = var / 2 ⇒ var /= 2

rem = rem % 10 ⇒ rem %= 10

j = j - (i + var + rem) ⇒ j -= (i + var + rem)

x = x \*\* 2 ⇒ x \*\*= 2

## LAB – Variáveis – Conversor simples

## Objetivos

* familiarizar-se com o conceito de, e trabalhar com, variáveis;
* realizar cálculos e conversões básicos;
* experimentar com código Python.

## Cenário

Milhas e quilómetros são unidades de comprimento ou distância.

Tendo em mente que 1 milha é aproximadamente igual a 1.61 quilómetros, complete o programa no editor para que ele converta:

* milhas para quilómetros;
* quilómetros para milhas.

Não altere nada no código existente. Escreva o seu código nos locais indicados por ###. Teste o seu programa com os dados que fornecemos no source code.

Preste especial atenção ao que está a acontecer dentro da função print() . Analise como fornecemos múltiplos argumentos à função, e como produzimos os dados esperados.

Note-se que alguns dos argumentos dentro da função print() são strings (por exemplo, "miles is", enquanto alguns outros são variáveis (por exemplo, miles).

**SUGESTÃO**

Há mais uma coisa interessante a acontecer ali. Consegue ver outra função dentro da função print() ? É a função round() . O seu trabalho é o de arredondar o resultado em output para o número de casas decimais especificadas nos parêntesis, e devolver um float (dentro da round() função pode encontrar o nome da variável, uma vírgula, e o número de casas decimais que pretendemos). Vamos falar sobre funções muito em breve, por isso não se preocupe caso ainda não esteja tudo totalmente claro. Só queremos despertar a sua curiosidade.

Depois de concluir o lab, abra a Sandbox e experimente mais. Tente escrever conversores diferentes, por exemplo, um conversor de USD para EUR, um conversor de temperatura, etc. - deixe que a sua imaginação voe! Tente fazer output dos resultados combinando strings e variáveis. Tente usar e experimentar com a round() função para arredondar os seus resultados a uma, duas ou três casas decimais. Verifique o que acontece se não fornecer qualquer número de dígitos. Lembre-se de testar os seus programas.

Experimente, tire conclusões, e aprenda. Seja curioso.

## Output esperado

7.38 miles is 11.88 kilometers

12.25 kilometers is 7.61 miles

## LAB – Operadores e expressões

## Objetivos

* familiarizar-se com o conceito de números, operadores, e operações aritméticas em Python;
* realizar cálculos básicos.

## Cenário

Veja o código no editor: ele lê um valor float , coloca-o numa variável chamada x, e imprime o valor de uma variável chamada y. A sua tarefa é completar o código para avaliar a seguinte expressão:

3x3 - 2x2 + 3x - 1

O resultado deve ser atribuído a y.

Lembre-se que a notação algébrica clássica gosta de omitir o operador de multiplicação - precisa de o utilizar explicitamente. Note como alteramos o tipo de dados para garantir que x é do tipo float.

Mantenha o seu código limpo e legível, e teste-o usando os dados que fornecemos, atribuindo-o cada vez à variável x (por hardcoding). Não desanime por quaisquer falhas iniciais. Seja persistente e inquisitivo.

## Dados de teste

Input de amostra

x = 0

x = 1

x = -1

Output esperado

y = -1.0

y = 3.0

y = -9.0

# Key takeaways

1. Uma **variável** é um local nomeado, reservado para armazenar valores na memória. Uma variável é criada ou inicializada automaticamente quando se lhe atribui um valor pela primeira vez. (2.1.4.1)

2. Cada variável deve ter um nome exclusivo - um **identificador**. Um nome de identificador válido deve ser uma sequência não vazia de carateres, deve começar com o underscore (\_), ou uma letra, e não pode ser uma keyword de Python. O primeiro caratere pode ser seguido por underscores, letras e dígitos. Os identificadores em Python diferenciam entre maiúsculas e minúsculas (são case-sensitive). (2.1.4.1)

3. O Python é uma **linguagem dinamicamente dactilografada**, o que significa que não precisa de *declarar* variáveis nela contidas. (2.1.4.3) Para atribuir valores a variáveis, pode-se utilizar um operador de atribuição simples na forma do sinal (=) igual, ou seja, var = 1.

4. Também pode utilizar **operadores de atribuição composta** (operadores de atalho) para modificar valores atribuídos a variáveis, por exemplo, var += 1, ou var /= 5 \* 2. (2.1.4.8)

5. Pode atribuir novos valores a variáveis já existentes, utilizando o operador de atribuição ou um dos operadores compostos, por exemplo: (2.1.4.5)

var = 2

print(var)

var = 3

print(var)

var += 1

print(var)

6. Pode combinar texto e variáveis utilizando o operador + , e utilizar a função print() para fazer output de strings e variáveis, por exemplo: (2.1.4.4)

var = "007"

print("Agent " + var)

**Exercício 1**

Qual é o output do seguinte snippet?

var = 2

var = 3

print(var)

Verifique

**Exercício 2**

Quais dos seguintes nomes de variáveis são inválidos em Python?

my\_var

m

101

averylongvariablename

m101

m 101

Del

del  
Verifique

**Exercício 3**

Qual é o output do seguinte snippet?

a = '1'

b = "1"

print(a + b)

Verifique

**Exercício 4**

Qual é o output do seguinte snippet?

a = 6

b = 3

a /= 2 \* b

print(a)

# Deixar comentários em código: porquê, como, e quando

Pode querer colocar algumas palavras dirigidas não ao Python mas aos humanos, normalmente para explicar a outros leitores do código como funcionam os truques utilizados no código, ou os significados das variáveis, e para manter a informação armazenada sobre quem é o autor e quando o programa foi escrito.

Uma observação inserida no programa, que é **omitida em runtime**, é chamada um **comentário**.

Como se deixa este tipo de comentário no source code? Tem de ser feito de uma forma que não force o Python a interpretá-lo como parte do código.

Sempre que o Python encontra um comentário no seu programa, o comentário é completamente transparente para ele - do ponto de vista de Python, este é apenas um espaço (independentemente da duração do comentário real).

Em Python, um comentário é um pedaço de texto que começa com um sinal # (cardinal, ou hash em inglês) e se estende até ao final da linha.

Se quiser um comentário que abranja várias linhas, tem de colocar um hash à frente de todas elas.

Tal como aqui:

# This program evaluates the hypotenuse c.

# a and b are the lengths of the legs.

a = 3.0

b = 4.0

c = (a \*\* 2 + b \*\* 2) \*\* 0.5 # We use \*\* instead of square root.

print("c =", c)

Programadores bons e responsáveis **descrevem cada peça de código importante**, por exemplo, explicando o papel das variáveis; embora deva ser declarado que a melhor maneira de comentar as variáveis é nomeá-las de uma forma inequívoca.

Por exemplo, se uma determinada variável for concebida para armazenar uma área de algum quadrado único, o nome square\_area será obviamente melhor do que aunt\_jane.

Dizemos que o primeiro nome é **self-commenting** (autocomenta-se).

Os comentários podem ser úteis noutro aspeto - pode utilizá-los para **marcar um pedaço de código que atualmente não é necessário** por qualquer razão. Veja o exemplo abaixo, se **descomentar** a linha realçada, isto afetará o output do código:

# This is a test program.

x = 1

y = 2

# y = y + x

print(x + y)

Isto é frequentemente feito durante os testes de um programa, a fim de isolar o local onde um erro possa estar escondido.

**SUGESTÃO**

Se quiser comentar ou descomentar rapidamente várias linhas de código, selecione a(s) linha(s) que deseja modificar e use o seguinte atalho de teclado: **CTRL** + **/** (Windows) ou **CMD** + **/** (Mac OS). É um truque muito útil, não é? Experimente [este código](https://edube.org/sandbox/ffa32cfe-a181-11e9-ade3-0242e71d5f55) na Sandbox.

## LAB - Comentários

## Objetivos

* familiarizar-se com o conceito de comentários em Python;
* usar e não usar comentários;
* substituir comentários por código;
* experimentar com código Python.

## Cenário

O código no editor contém comentários. Tente melhorá-lo: adicione ou remova comentários onde achar apropriado (sim, por vezes remover um comentário pode tornar o código mais legível), e altere nomes de variáveis onde achar que isso irá melhorar a compreensão do código.

**NOTA**

Os comentários são muito importantes. São utilizados não só **para facilitar a compreensão** dos seus programas, mas também para **desativar as partes do código que atualmente não são necessárias** (por exemplo, quando precisa de testar apenas algumas partes do seu código e ignorar outras). Bons programadores **descrevem** cada parte importante do código e dão nomes **self-commenting** às variáveis, já que por vezes é simplesmente muito melhor deixar informação no código.

É bom usar nomes de variáveis **legíveis**, e por vezes é melhor **dividir o seu código** em partes nomeadas (por exemplo, funções). Em algumas situações, é uma boa ideia escrever os passos dos cálculos de uma forma mais clara.

Mais uma coisa: pode acontecer que um comentário contenha uma informação errada ou incorreta - nunca se deve fazer isso de propósito!

# Key takeaways

1. Os comentários podem ser utilizados para deixar informações adicionais em código. São omitidos em runtime. A informação deixada no source code é dirigida aos leitores humanos. Em Python, um comentário é um pedaço de texto que começa com #. O comentário estende-se até ao fim da linha.

2. Se quiser colocar um comentário que abranja várias linhas, precisa de colocar # à frente de todas elas. Além disso, pode utilizar um comentário para marcar um pedaço de código que não é necessário neste momento (ver a última linha do snippet abaixo), por exemplo:

# This program prints

# an introduction to the screen.

print("Hello!") # Invoking the print() function

# print("I'm Python.")

3. Sempre que possível e justificado, deve dar **nomes self-commenting** às variáveis, por exemplo, se estiver a utilizar duas variáveis para armazenar um comprimento (em inglês, length) e largura (width) de algo, os nomes das variáveis length e width podem ser uma escolha melhor do que myvar1 e myvar2.

4. É importante utilizar comentários para tornar os programas mais fáceis de compreender, e utilizar nomes de variáveis legíveis e significativos em código. Contudo, é igualmente importante **não usar** nomes de variáveis que sejam confusos, ou deixar comentários que contenham informações erradas ou incorretas!

5. Os comentários podem ser importantes quando *você* estiver a ler o seu próprio código após algum tempo (confie em nós, os programadores esquecem-se do que o seu próprio código faz), e quando *outros* estão a ler o seu código (pode ajudá-los a compreender o que os seus programas fazem e como o fazem mais rapidamente).

**Exercício 1**

Qual é o output do seguinte snippet?

# print("String #1")

print("String #2")

Verifique

**Exercício 2**

O que acontecerá quando executar o seguinte código?

# This is

a multiline

comment. #

print("Hello!")

**A função input()**

Vamos agora apresentar-lhe uma função completamente nova, que parece ser um reflexo da boa e velha função print() .

Porquê? Bem, print() envia dados para a consola.

A nova função obtém dados dela.

print() não tem resultado utilizável. O significado da nova função é **devolver um resultado muito utilizável**.

A função é chamada input(). The name of the function says everything.

A tecla programável input() function is able to read data entered by the user and to return the same data to the running program.

O programa pode manipular os dados, tornando o código genuinamente interativo.

Virtualmente todos os programas **lêem e processam dados**. Um programa que não recebe um input do utilizador é um **programa surdo**.

Dê uma vista de olhos no nosso exemplo:

print("Tell me anything...")

anything = input()

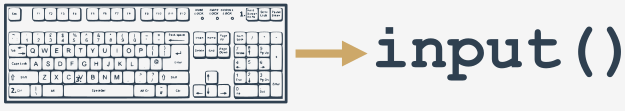
print("Hmm...", anything, "... Really?")

Isto mostra um caso muito simples de utilização da função input() .

Nota:

* O programa **pede ao utilizador que introduza alguns dados** da consola (muito provavelmente utilizando um teclado, embora também seja possível introduzir dados utilizando voz ou imagem);
* a função input() é invocada sem argumentos (esta é a forma mais simples de usar a função); a função irá **mudar a consola para o modo de input**; verá um cursor a piscar, e poderá introduzir algumas keystrokes, terminando com a tecla *Enter*; todos os dados introduzidos serão **enviados para o seu programa** através do resultado da função;
* nota: é necessário atribuir o resultado a uma variável; isto é crucial - a falta desta etapa fará com que os dados introduzidos se percam;
* então, utilizamos a função print() para fazer output dos dados que obtemos, com algumas observações adicionais.

Tente executar o código e deixe que a função demonstre o que pode fazer por si.



**A função input() com um argumento**

A função input() pode fazer algo mais: pode incitar o utilizador sem qualquer ajuda de print().

Modificámos um pouco o nosso exemplo, olha para o código:

anything = input("Tell me anything...")

print("Hmm...", anything, "...Really?")

Nota:

* a keyword input() é invocada com um argumento - é uma string contendo uma mensagem;
* a mensagem será exibida na consola antes de ser dada ao utilizador a oportunidade de introduzir qualquer coisa;
* input() fará então o seu trabalho.

Esta variante da invocação input() simplifica o código e torna-o mais claro.

**O resultado da função input()**

Já o dissemos, mas deve ser afirmado uma vez mais sem ambiguidade: o **resultado da função**input()**é uma string**.

Uma string contendo todos os carateres que o utilizador introduz a partir do teclado. Não é um inteiro ou um float.

Isto significa que **não deve usá-la como um argumento de qualquer operação aritmética**, por exemplo, não pode usar estes dados para os elevar ao quadrado, dividi-los por qualquer coisa, ou dividir qualquer coisa por eles.

anything = input("Enter a number: ")

something = anything \*\* 2.0

print(anything, "to the power of 2 is", something)

# A função input() - operações proibidas

Veja o código no editor. Execute-o, insira qualquer número e prima *Enter*.

O que acontece?

O Python deve ter dado o seguinte output:

Traceback (most recent call last):

File ".main.py", line 4, in <module>

something = anything \*\* 2.0

TypeError: unsupported operand type(s) for \*\* or pow(): 'str' and 'float'

**output**

A última linha da frase explica tudo - tentou aplicar o operador \*\* para 'str' (string) acompanhado com 'float'.

Isto é proibido.

Isto deve ser óbvio - pode prever o valor de "to be or not to be" elevado à potência de 2?

Não podemos. O Python também não pode.

Será que caímos num impasse? Existe uma solução para este problema? Claro que existe.

**Type casting**

O Python oferece duas funções simples para especificar um tipo de dados e resolver este problema - aqui estão elas: int() e float().

Os seus nomes são self-commenting:

* a função int() **toma um argumento** (por exemplo, uma string: int(string)) e tenta convertê-lo num inteiro; se falhar, todo o programa também falhará (há uma solução para esta situação, mas mostrar-lhe-emos isto um pouco mais tarde);
* a função float() toma um argumento (por exemplo, uma string: float(string)) e tenta convertê-lo num float (o resto é o mesmo).

Isto é muito simples e muito eficaz. Além disso, pode invocar qualquer uma das funções, passando os input() resultados diretamente para elas. Não há necessidade de utilizar qualquer variável como armazenamento intermédio.

Implementámos a ideia no editor - dê uma vista de olhos ao código.

Pode imaginar como flui a string introduzida pelo utilizador a partir de input() para print()?

Tente executar o código modificado. Não se esqueça de introduzir um **número válido**.

Verifique alguns valores diferentes, pequenos e grandes, negativos e positivos. Zero é também um bom input.

# Mais sobre input() e type casting

Ter uma equipa composta pelo trio input()-int()-float() abre muitas novas possibilidades.

Acabará por ser capaz de escrever programas completos, aceitando dados sob a forma de números, processando-os e exibindo os resultados.

É claro que estes programas serão muito primitivos e pouco utilizáveis, uma vez que não podem tomar decisões, e consequentemente não são capazes de reagir de forma diferente a diferentes situações.

No entanto, isto não é realmente um problema; vamos mostrar-lhe como ultrapassá-lo em breve.

O nosso próximo exemplo refere-se ao programa anterior para encontrar o comprimento de uma hipotenusa. Vamos reescrevê-lo e torná-lo capaz de ler o comprimento das pernas a partir da consola.

Veja a janela do editor - é este o aspeto que tem agora.

O programa pede duas vezes ao utilizador o comprimento de ambas as pernas, avalia a hipotenusa e imprime o resultado.

Execute-o e tente introduzir alguns valores negativos.

O programa - infelizmente - não reage a este erro óbvio.

Vamos ignorar esta fraqueza por agora. Voltaremos a este assunto em breve.

Note que no programa que pode ver no editor, a variável hypo é utilizada apenas para uma única finalidade - para guardar o valor calculado entre a execução da linha de código adjacente.

Como a função print() aceita uma expressão como seu argumento, pode **remover a variável** do código.

Tal como isto:

leg\_a = float(input("Input first leg length: "))

leg\_b = float(input("Input second leg length: "))

print("Hypotenuse length is", (leg\_a\*\*2 + leg\_b\*\*2) \*\* .5)

# Operadores de String - introdução

Chegou o momento de voltar a estes dois operadores aritméticos: + e \*.

Queremos mostrar-lhe que eles têm uma segunda função. Eles são capazes de fazer algo mais do que apenas **adicionar** e **multiplicar**.

Vimo-los em ação onde os seus argumentos são números (floats ou inteiros, não importa).

Agora vamos mostrar-lhe que eles também conseguem lidar com strings, embora de uma forma muito específica.

# Concatenação

O sinal + (mais), quando aplicado a duas strings, torna-se um **operador de concatenação**:

string + string

Simplesmente **concatena** (cola) duas strings numa. Claro que, tal como o seu irmão aritmético, pode ser usado mais de uma vez numa expressão, e em tal contexto comporta-se de acordo com a ligação do lado esquerdo.

Em contraste com o seu irmão aritmético, o operador da concatenação **não é comutativo**, ou seja "ab" + "ba" não é o mesmo que "ba" + "ab".

Não se esqueça - se quiser que o sinal + seja um **concatenador**, não um adicionador, deve assegurar-se de que **ambos os seus argumentos são strings**.

Não se podem misturar tipos aqui.

Este programa simples mostra o sinal + na sua segunda utilização:

fnam = input("May I have your first name, please? ")

lnam = input("May I have your last name, please? ")

print("Thank you.")

print("\nYour name is " + fnam + " " + lnam + ".")

Nota: usando + para concatenar strings permite construir o output de uma forma mais precisa do que com uma função pura print() , mesmo que enriquecida com os end= e sep= argumentos de keyword.

Execute o código e veja se o output corresponde às suas previsões.

**Replicação**

O sinal \* (asterisco), quando aplicado a uma string e número (ou um número e string, visto permanecer comutativo nesta posição) torna-se um **operador de replicação**:

string \* number

number \* string

Replica a string o mesmo número de vezes especificado pelo número.

Por exemplo:

* "James" \* 3 dá "JamesJamesJames"
* 3 \* "an" dá "ananan"
* 5 \* "2" (ou "2" \* 5) dá "22222" (não 10!)

LEMBRE-SE

Um número menor ou igual a zero produz uma **string vazia**.

Este programa simples "desenha" um retângulo, fazendo uso de um antigo operador (+) num novo papel:

print("+" + 10 \* "-" + "+")

print(("|" + " " \* 10 + "|\n") \* 5, end="")

print("+" + 10 \* "-" + "+")

Note a forma como utilizámos os parêntesis na segunda linha do código.

Tente praticar para criar outras formas ou a sua própria obra de arte!

# Conversão de tipo: str()

Já sabe como utilizar as funções int() e float() para converter uma string num número.

Este tipo de conversão não é uma rua de sentido único. Também se pode **converter um número numa string**, o que é muito mais fácil e seguro - esta operação é sempre possível.

Uma função capaz de o fazer chama-se str():

str(number)

Para ser honesto, pode fazer muito mais do que apenas transformar números em strings, mas isso pode ficar para mais tarde.

# O “triângulo de ângulo retângulo” novamente

Aqui está o nosso programa “triângulo de ângulo retângulo” novamente:

leg\_a = float(input("Input first leg length: "))

leg\_b = float(input("Input second leg length: "))

print("Hypotenuse length is " + str((leg\_a\*\*2 + leg\_b\*\*2) \*\* .5))

Modificámo-lo um pouco para lhe mostrar como a função str() funciona. Graças a isso, podemos **passar todo o resultado para a função**print()**como uma string**, esquecendo as vírgulas.

Deu alguns passos sérios no seu caminho para a programação em Python.

Já conhece os tipos de dados básicos, e um conjunto de operadores fundamentais. Sabe como organizar o output e como obter dados do utilizador. Estas são bases muito fortes para o Módulo 3. Mas antes de passarmos ao módulo seguinte, vamos fazer alguns laboratórios, e recapitular tudo o que aprendeu nesta secção.

## LAB – Input e output simples

## Objetivos

* familiarizar-se com o input e output de dados em Python;
* avaliar expressões simples.

## Cenário

A sua tarefa é completar o código a fim de avaliar os resultados de quatro operações aritméticas básicas.

Os resultados têm de ser impressos para a consola.

Poderá não conseguir proteger o código de um utilizador que queira dividir por zero. Tudo bem, não se preocupe com isso por agora.

Teste o seu código - produz os resultados que espera?

Não lhe mostraremos quaisquer dados de teste - isso seria demasiado simples.

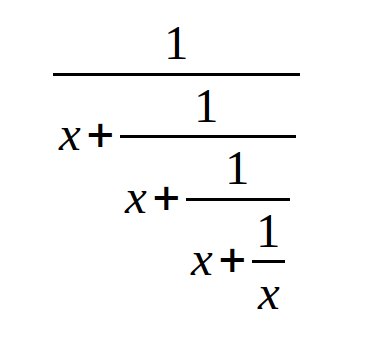
## LAB – Operadores e expressões

## Objetivos

* familiarizar-se com o conceito de números, operadores e operações aritméticas em Python;
* compreender a precedência e associatividade dos operadores Python, bem como o uso adequado de parêntesis.

## Cenário

A sua tarefa é a de completar o código, de forma a avaliar a seguinte expressão:



O resultado deve ser atribuído a y. Tenha cuidado - observe os operadores e mantenha as suas prioridades em mente. Não hesite em utilizar os parêntesis de que necessitar.

Pode utilizar variáveis adicionais para encurtar a expressão (mas não é necessário). Teste o seu código com cuidado.

## Dados de Teste

Input de amostra: 1

Output esperado:

y = 0.6000000000000001

Input de amostra: 10

Output esperado:

y = 0.09901951266867294

Input de amostra: 100

Output esperado:

y = 0.009999000199950014

Input de amostra: -5

Output esperado:

y = -0.19258202567760344

## LAB – Operadores e expressões

## Objetivos

* melhorar a capacidade de utilizar números, operadores, e operações aritméticas em Python;
* utilizar as capacidades de formatação da função print() ;
* aprender a expressar os fenómenos da vida quotidiana em termos de linguagem de programação.

## Cenário

A sua tarefa é preparar um código simples capaz de avaliar o **tempo final** de um período de tempo, dado como um número de minutos (pode ser arbitrariamente grande). O tempo inicial é dado como um par de horas (0.. 23) e minutos (0.. 59). O resultado tem de ser impresso para a consola.

Por exemplo, se um evento começar às **12:17** e durar **59 minutos**, ele terminará às **13:16**.

Não se preocupe com quaisquer imperfeições no seu código - não faz mal se aceitar um tempo inválido - o mais importante é que o código produza resultados válidos para dados de input válidos.

Teste o seu código com cuidado. Dica: utilizar o operador % pode ser a chave para o sucesso.

## Dados de teste

Input de amostra:12

17

59

Output esperado: 13:16

Input de amostra:23

58

642

Output esperado: 10:40

Input de amostra:0

1

2939

Output esperado: 1:0

# Key takeaways

1. A função print() **envia dados para a consola**, enquanto a função input() **obtém dados da consola**.

2. O método input() vem com um parâmetro opcional: **a string prompt**. Permite-lhe escrever uma mensagem antes do input do utilizador, por exemplo

name = input("Enter your name: ")

print("Hello, " + name + ". Nice to meet you!")

3. Quando a função input() é chamada, o fluxo do programa é interrompido, o símbolo de prompt continua a piscar (pede ao utilizador para tomar medidas quando a consola é mudada para o modo de input) até o utilizador ter introduzido um input e/ou premido a tecla *Enter*.

NOTA

Pode testar a funcionalidade da função input() em todo o seu scope localmente na sua máquina. Por razões de otimização de recursos, limitámos o tempo máximo de execução do programa no Edube a alguns segundos. Vá à Sandbox, copie-cole o snippet acima, execute o programa, e não faça nada - espere apenas alguns segundos para ver o que acontece. O seu programa deve ser interrompido automaticamente após um breve momento. Agora abra o IDLE, e execute lá o mesmo programa - consegue ver a diferença?

Dica: a característica acima mencionada da função input() pode ser utilizada para solicitar o utilizador a terminar um programa. Veja o código em baixo:

name = input("Enter your name: ")

print("Hello, " + name + ". Nice to meet you!")

print("\nPress Enter to end the program.")

input()

print("THE END.")

3. O resultado da função input() é uma string. Podem adicionar-se strings umas às outras usando a concatenação (+) operador. Verifique este código:

num\_1 = input("Enter the first number: ") # Enter 12

num\_2 = input("Enter the second number: ") # Enter 21

print(num\_1 + num\_2) # the program returns 1221

4. Também pode multiplicar (\* - replicação) strings, por exemplo:

my\_input = input("Enter something: ") # Example input: hello

print(my\_input \* 3) # Expected output: hellohellohello

**Exercício 1**

Qual é o output do seguinte snippet?

x = int(input("Enter a number: ")) # The user enters 2

print(x \* "5")

Verifique

**Exercício 2**

Qual é o output do seguinte snippet?

x = input("Enter a number: ") # The user enters 2

print(type(x))

**Parabéns! Completou o Módulo 2.**

Muito bem! Chegou ao fim do Módulo 2 e completou um marco importante na sua educação em programação Python. Aqui está um breve resumo dos objetivos que abordou e com os quais se familiarizou no Módulo 2:

* os métodos básicos de formatação e de output de dados oferecidos pelo Python, juntamente com os tipos primários de dados e operadores numéricos, as suas relações mútuas e ligações;
* o conceito de variáveis e convenções de nomeação de variáveis;
* o operador de atribuição, as regras que regem a construção das expressões;
* O input e a conversão de dados;

Está agora pronto para fazer o quiz do módulo e tentar o desafio final: Teste do Módulo 2, que o ajudará a avaliar o que aprendeu até agora.

