# Variáveis, entradas e saídas

## 1. Variáveis

Variáveis são "pedacinhos de memória" onde guardamos dados. Sempre que referenciamos o nome, o pedacinho de memória é acessado e seu dado é recuperado.

Criamos variáveis dando um nome a elas e usando o operador de atribuição (o sinal de igualdade: =) para atribuir um valor inicial.

x = 10

No exemplo acima, foi criada uma variável chamada x que guarda o valor 10. Ou seja, reservamos um pedacinho de memória e guardamos o número 10 lá.

### 2. Tipos de variáveis

Variáveis em Python possuem 4 tipos básicos:

* int: números inteiros, ou seja, números sem parte decimal: 0, 5, -1, 1000
* float: números reais, ou seja, números com parte decimal: 1.0, -2.7, 3.14
* str: cadeias de caracteres (strings), ou seja, dados textuais: 'Olá Mundo!', "eu tenho 18 anos"
* bool: valores lógicos (booleanos), ou seja, apenas um entre dois valores possíveis: True ou False

nome = 'Zé' # uma variável do tipo string - note as aspas

email = "ze@letscode.com.br" # outra string

idade = 22 # uma variável inteira

salario = 5999.85 # uma variável float - usamos ponto, não vírgula

receberNewsletter = True # uma variável bool

Dica: Tente sempre utilizar nomes intuitivos para suas variáveis. O nome deveria ser uma boa descrição do dado que a variável guarda. Nomes como 'x', 'y', 'z', 'a', 'b', 'c', 'a1', 'a2', 'a3' etc podem se tornar bastante confusos quando nossos códigos são muito grandes.

## 3. Comentários

Note que nos exemplos acima, escrevemos textos no meio do código utilizando o símbolo #. Esses textos são comentários: quando utilizamos o símbolo #, o Python irá ignorar tudo o que vier em seguida (na mesma linha). Utilizamos comentários para explicar pedaços do nosso código para que nós mesmos ou outros colegas no futuro entendam o que fizemos e possam modificar ou corrigir o código com mais facilidade. Também podemos escrever comentários de múltiplas linhas utilizando aspas triplas - neste caso, utilizamos-as para abrir e depois para fechar o bloco de comentários.

'''

Este é um comentário de várias linhas.

Tudo que veio após o primeiro trio de aspas e antes do segundo

será ignorado pelo Python.

'''

## 4. Saídas

Chamamos de saídas do nosso programa todos os dados que são gerados pelo programa e serão fornecidos para o usuário. A função de saída em tela no Python é o print. Colocamos entre parênteses o dado que queremos que apareça.

print('olá mundo!') # exibe a frase 'olá mundo' na tela

Os dados a serem exibidos não precisam ser valores constantes, como a frase fixa acima. Eles podem ser variáveis:

idade = 20

print(idade)

Note que quando usamos aspas, o Python trata o valor como uma string, um texto literal. Quando não usamos aspas, o Python irá considerar que aquele é o nome de uma variável e irá acessá-la para buscar seu valor.

Podemos exibir múltiplos dados em um print. Para isso, basta separá-los por vírgula e eles irão aparecer na tela na mesma ordem que apareceram no código:

nome = 'Mario'

linguagem = 'Python'

print('Oi, eu sou o', nome, 'e eu programo em', linguagem)

```

Resultado na tela:

Oi, eu sou o Mario e eu programo em Python

```

## 5. Entradas

Assim como temos dados de saída - dados gerados pelo código e fornecidos para o usuário - também temos dados de entrada: informações que o usuário possuí e deve fornecer ao código. Para receber entradas pelo teclado, utilizaremos a função input. Devemos fazer uma variável receber o valor capturado pelo input.

nome = input()

print('Olá', nome)

O programa acima captura o nome do usuário e em seguida mostra a mensagem "olá" seguida do nome do usuário. Note que o programa fica parado em uma tela em branco com um cursor piscando aguardando a digitação pelo usuário. Isso pode ser confuso para o usuário, que não sabe o que o programa está esperando. Por isso, dentro dos parênteses do input podemos colocar uma mensagem simples informando o que o programa gostaria que ele fizesse:

nome = input('Qual é o seu nome?')

print('Olá', nome)

### 5.1. Determinando o tipo da entrada

Vamos imaginar um programa que informa quantos anos falta para que uma criança atinja a maioridade. Podemos ler a idade da criança pelo teclado (entrada), subtrair a idade do número 18 (processamento) e exibir o resultado da conta na tela (saída). Considere a solução abaixo:

idade = input('Digite a sua idade: ')

resto = 18 - idade

print('Faltam', resto, 'anos.')

Se você copiar e executar o programa, ele dará erro na segunda linha. Isso ocorre porque o teclado é uma "máquina de escrever" um pouco mais moderna. Portanto, tudo que entra pelo teclado é considerado pelo Python como texto (ou seja, str). Porém, não podemos "fazer contas" com textos. Fazemos contas com números. Portanto, neste caso, precisamos falar para o Python interpretar a nossa entrada como um número. Um bom tipo de dado para "idade" seria um número inteiro. Fazemos isso colocando o nome do tipo desejado, e entre parênteses colocamos nosso input:

idade = int(input('Digite a sua idade: '))

resto = 18 - idade

print('Faltam', resto, 'anos.')

## 6. Operadores

Como podemos observar no exemplo anterior, o Python faz operações aritméticas de maneira bastante intuitiva, similar ao que estamos acostumados. Os operadores aceitos são:

* Soma: +
* Subtração: -
* Multiplicação: \*
* Divisão: /
* Divisão inteira: //
* Resto da divisão: %
* Potência: \*\*

numero1 = int(input('Digite um número: '))

numero2 = int(input('Digite outro número: ''))

soma = numero1 + numero2

subtracao = numero1 - numero2

multiplicacao = numero1 \* numero2

divisao = numero1 / numero2

divinteira = numero1 // numero2

resto = numero1 % numero2

elevado = numero1 \*\* numero2

print('Soma: ', soma)

print('Subtração: ', subtracao)

print('Multiplicação: ', multiplicacao)

print('Divisão: ', divisao)

print('Divisão inteira: ', divinteira)

print('Resto da divisão: ', resto)

print('Potência: ', elevado)

Operadores de divisão: Note que temos 3 operadores de divisão. O que seria cada um deles? Vamos supor que numero1 seja 15 e numero2 seja 6. Ao calcular essa divisão, chegamos na seguinte situação:

15 |\_\_ 6  
03 | 2

Note que, considerando apenas números inteiros, não conseguimos mais prosseguir com a divisão. Neste caso, a divisão inteira (numero1 // numero2) dará 2. Já o resto da divisão (numero1 % numero2) dará 3. Porém, considerando casas decimais é possível prosseguir com a divisão:

15 |\_\_ 6  
30 | 2,5  
0  
Portanto, a divisão comum (numero1 / numero2) dará 2.5.

# Operações lógicas

## 1. Operações booleanas

Na aula passada, vimos que nossos dados podem ter 4 tipos básicos: str (texto), int (número inteiro), float (número real) e bool (lógico). Vimos diversas operações aritméticas também, como a soma, a divisão ou a potência, cujos resultados são int ou float. Porém, podemos ter também operações cujo resultado é bool: são operações lógicas. Operações lógicas tipicamente são comparações:

comparacao1 = 5 > 3

print(comparacao1)

comparacao2 = 5 < 3

print(comparacao2)

Se executarmos o código acima, a saída que teremos na tela será:

True

False

Isso ocorre porque 5 é maior que 3. Portanto, comparacao1 recebeu uma expressão cujo valor lógico é verdadeiro, portanto seu resultado foi True, e o oposto ocorreu para comparacao2. O Python possui 6 operadores de comparação:

* Maior que: >
* Maior ou igual: >=
* Menor que: <
* Menor ou igual: <=
* Igual: ==
* Diferente: !=

Note que o operador para comparar se 2 valores são iguais é ==, e não =. Isso ocorre porque o operador = é o nosso operador de atribuição: ele diz que a variável à sua esquerda deve receber o valor da expressão à direita. O operador de == irá testar se o valor à sua esquerda é igual ao valor à sua direita e irá responder True ou False, como todos os outros operadores de comparação.

### 1.1. Conjunções lógicas

Em alguns casos precisamos testar se duas ou mais condições são verdadeiras. Imagine, por exemplo, que o critério de aprovação em uma escola seja média superior a 6.0 e presença superior a 75%. Neste caso, o aluno precisa atender a ambos os critérios para ser aprovado. Se ele tirou uma ótima nota mas faltou demais, será reprovado. Se ele compareceu a todas as aulas mas teve notas baixas, idem. O código abaixo testa se é verdade que o aluno foi aprovado:

media = float(input('Digite a média do aluno: '))

presenca = float(input('Digite as presenças do aluno: '))

aprovado = media >= 6.0 and presenca >= 0.75

print(aprovado)

Execute o código acima e teste algumas combinações diferentes de valores. Note que basta uma das condições ser falsa para que o resultado total seja False.

Porém, nem sempre precisamos que ambas as condições sejam verdadeiras. Vários de nós já nos deparamos com promoções de queima de estoque anunciadas da seguinte maneira: "promoção válida até o dia 15 deste mês ou enquanto durarem os estoques". Neste caso, para a promoção acabar, não é necessário que ambas as coisas ocorram (atingir o dia 15 e zerar o estoque). Se ainda temos 10 itens no estoque mas hoje é dia 16, a promoção acabou. Se hoje é dia 5 mas o estoque está zerado, a promoção acabou. O programa abaixo testa se a promoção acabou:

diaFinal = int(input('Digite o dia do mês para encerrar a promoção: '))

diaAtual = int(input('Digite o dia do mês atual: '))

estoque = int(input('Digite a quantidade de itens no estoque: '))

acabou = diaAtual > diaFinal or estoque == 0

print(acabou)

Faça alguns testes com o programa acima e note que basta uma condição ser verdadeira para seu resultado ser True.

Resumo:

* and: verdadeiro se condição 1 for verdadeira e condição 2 for verdadeira
* or: verdadeiro se condição 1 for verdadeira ou condiçã 2 for verdadeira

# Expressões condicionais

## 1.1. Se

Os programas do capítulo Operações Lógicas não são "amigáveis" para o usuário. Ao invés de mostrar True ou False, por exemplo, seria mais útil exibir se o aluno foi "Aprovado" ou "Reprovado". Para que possamos escrever na tela as mensagens "Aprovado" ou "Reprovado", é necessário que haja em algum ponto do código o trecho print('Aprovado') e o trecho print('Reprovado'). Porém, não gostaríamos que ambos fossem exibidos ao mesmo tempo. Uma condicional é uma instrução em Python que decide se outras linhas serão ou não executadas dependendo do resultado de uma condição (operação lógica). Se a condição for verdadeira, as linhas são executadas. Senão, são ignoradas. A condicional mais básica em Python é o if (se):

nota1 = float(input('Digite a nota 1: '))

nota2 = float(input('Digite a noat 2: '))

media = (nota1 + nota2)/2

if media >= 6.0:

print('Aprovado')

print('Média: ', media)

Execute o programa acima. Note que se (if) a média é maior ou igual a 6.0, ele exibe a mensagem "Aprovado" e depois a média. Caso contrário, ele apenas exibe a média.

Para dizermos que uma ou mais linhas "pertencem" ao nosso if, usamos um símbolo de parágrafo (tecla "Tab" no teclado). O programa sabe que o if "acabou" quando as linhas param de ter "tabs". Esses tabs são chamados de indentação.

## 1.2. Senão

Note que conseguimos fazer nosso programa decidir se ele exibe a mensagem "Aprovado" ou não. O próximo passo seria fazer ele decidir entre 2 mensagens diferentes: "Aprovado" ou "Reprovado". Um primeiro jeito de fazer isso seria um segundo if invertendo a condição:

nota1 = float(input('Digite a nota 1: '))

nota2 = float(input('Digite a noat 2: '))

media = (nota1 + nota2)/2

if media >= 6.0:

print('Aprovado')

if media < 6.0:

print('Reprovado')

print('Média: ', media)

O programa acima funciona. Porém, conforme nossos programas começam a ficar mais complexos e nossos if começam a ter linhas demais, podemos nos perder e esquecer que esses 2 if são 2 casos mutuamente exclusivos. Pior ainda, podemos vir a acrescentar condições novas em um e esquecer de atualizar no outro. Em casos onde temos condições mutuamente exclusivas, podemos utilizar um par if/else (se/senão). Se a condição for verdadeira, faça tal coisa. Senão, faça outra coisa.

nota1 = float(input('Digite a nota 1: '))

nota2 = float(input('Digite a noat 2: '))

media = (nota1 + nota2)/2

if media >= 6.0:

print('Aprovado')

else:

print('Reprovado')

print('Média: ', media)

Note que o else não possui condição. A condição dele é implícita: é a negação da condição do if. Se o if executar, o else não executa e vice-versa. Consequentemente, o else não pode existir sem um if.

## 1.3. Aninhando condições

É possível aninhar condições: ou seja, colocar um novo if dentro de outro if ou else. Imagine que nossa escola não reprova direto o aluno com nota inferior a 6, e sim permite que ele faça uma recuperação. Porém, o aluno precisa ter tirado no mínimo média 3 para que permitam que faça a recuperação. Assim temos:

* Se nota maior ou igual a 6: aprovado.
* Senão:
  + Se nota entre 6 e 3: recuperação.
  + Senão, reprovado. Em Python:

nota1 = float(input('Digite a nota 1: '))

nota2 = float(input('Digite a noat 2: '))

media = (nota1 + nota2)/2

if media >= 6.0:

print('Aprovado')

else:

if media >= 3.0:

print('Recuperação')

else:

print('Reprovado')

print('Média: ', media)

#### 1.4. Senão-se

Note que se começarmos a aninhar muitas condições (if dentro de else dentro de else dentro de else...), nosso código pode começar a ficar confuso, com a aparência de uma "escadinha":

Se

Senão:

Se

Senão:

Se

Senão:

Se:

Senão:

Se:

...

Isso pode tornar o código bastante complexo e difícil de atualizar ou corrigir erros posteriormente. Para quebrar a "escadinha", existe a possibilidade de juntarmos o "se" do próximo nível com o "senão" do nível anterior: o elif: else + if (senão + se). O elif só é executado se um if der errado (ou seja, ele é um else), mas ele também tem uma condição que deve ser respeitada (ou seja ele também é um if). Podemos reescrever nosso código anterior utilizando um elif:

nota1 = float(input('Digite a nota 1: '))

nota2 = float(input('Digite a noat 2: '))

media = (nota1 + nota2)/2

if media >= 6.0:

print('Aprovado')

elif media >= 3.0:

print('Recuperação')

else:

print('Reprovado')

print('Média: ', media)

Podemos usar quantos elif nós quisermos. Sempre que um deles der errado, o próximo será testado. Quando algum deles der certo, todo o restante será ignorado. Opcionalmente, podemos ter um else ao final do bloco, que só será executado se o if e todos os elif derem errado. O bloco obrigatoriamente deve ser iniciado com um if.

# Malhas de repetição

Imagine que você queira fazer um programa que exibe os números de 1 até 5, em ordem crescente. Uma possibilidade seria:

print(1)

print(2)

print(3)

print(4)

print(5)

Porém, imagine que os requisitos do programa acabem sendo alterados, e agora o seu programa deverá ir até 1000. Ou, pior, imagine que o usuário irá digitar um valor e seu programa deverá contar apenas até o valor digitado. Note como fica difícil resolver esses problemas apenas copiando e colando linhas de código.

Vamos pensar em outro tipo de problema. Na aula passada, fizemos um exercício onde precisávamos validar algumas entradas do usuário. Uma dessas entradas era a idade, e gostaríamos de aceitar apenas valores entre 0 e 150. Sua solução provavelmente foi parecida com o código abaixo:

idade = int(input('Digite a idade: '))

if idade < 0 or idade > 150:

print('Erro')

Mas imagine que, ao invés de apenas mostrar uma mensagem de erro, nós devêssemos obrigar o usuário a continuar digitando valores novos para idade até que ele digite um valor válido (entre 0 e 150). Isso não seria possível utilizando apenas if, elif e else.

## 1.1. Enquanto

Os problemas enunciados acima podem ser resolvidos utilizando estruturas do tipo "enquanto". Em Python, a instrução while é bastante parecida com o if: ela possui uma expressão lógica, e seu conteúdo só será executado se a expressão for verdadeira. Porém, após chegar ao final, ela retorna ao início e testa novamente a condição. Se ela for verdadeira, seu conteúdo será executado de novo. Ao final da nova execução, a condição é testada novamente, e assim sucessivamente. A execução só será interrompida quando o teste se tornar falso. Vejamos como resolver o problema da idade utilizando o while:

idade = int(input('Digite a idade: '))

while idade < 0 or idade > 150:

print('Erro! Idade deve estar entre 0 e 150!')

idade = int(input('Digite a idade: '))

print('Obrigado!')

Faça alguns testes com o programa acima. Note que se você digitar uma idade válida desde o início, ele nunca chega a mostrar erro: o while é como um if e será ignorado se sua condição for falsa. Porém, caso você digite valores inválidos, a condição será verdadeira e ele irá executar enquanto você estiver digitando valores falsos.

Estruturas do tipo "enquanto" são conhecidas como malhas de repetição ou loops.

## 1.2. Condição de parada

No exemplo anterior, o que determina se o loop prossegue ou não é o valor de idade. Esse valor, por sua vez, pode mudar em cada execução do loop, já que temos um input lá dentro. Experimente rodar o programa sem aquele input e verifique o que ocorre.

idade = int(input('Digite a idade: '))

while idade < 0 or idade > 150:

print('Erro! Idade deve estar entre 0 e 150!')

print('Obrigado!')

O que ocorreu é o que chamamos de loop infinito: se a condição foi verdadeira uma vez, ela será para sempre, já que nunca mais alteramos o valor da variável envolvida no teste lógico. É importante criar caminhos para que a condição possa se tornar falsa em algum momento. Isso é o que chamamos de condição de parada do nosso loop.

## 1.3. Sequências numéricas

Iniciamos essa aula enunciando um problema onde gostaríamos de exibir números sequencialmente na tela. Isso é possível de resolver utilizando loops. Primeiro, observe o exemplo abaixo e responda: qual valor aparecerá na tela?

x = 5

x = x + 1

print(x)

Essa construção parece pouco intuitiva porque na matemática o operador = é bidirecional: a expressão "a = b" significa que a é igual a b e b é igual a a. Ao vermos x aparecendo em ambos os lados, parece que podemos simplesmente cortar dos dois lados, resultando em 0 = 1, o que é uma inverdade. Em Python o operador = na verdade não é o operador de igualdade da matemática, e sim o operador de atribuição de valores. Ou seja, o que ele diz é "pegue o resultado da expressão à direita e guarde na variável à esquerda". Portanto, no exemplo acima ele primeiro pega o valor antigo de x, que era 5, adiciona 1, resultando em 6, e guarda este novo resultado na variável x, substituindo o valor antigo. Logo, a resposta na tela é 6.

Se colocarmos uma expressão desse tipo dentro de um loop, podemos gerar sequências numéricas:

final = int(input('Digite o valor final da sequência: '))

numero = 1

while numero <= final:

print(numero)

numero = numero + 1

O programa acima pede para o usuário digitar um número, que será o valor final da sequência. Então ele irá imprimir a variável numero, que vale 1, e somar +1 nela. Em seguida imprimirá de novo a variável, agora valendo 2, e somará +1 nela. E assim sucessivamente até que ela ultrapasse o valor final, quando o loop deixará de ser executado.

Você consegue modificar o programa acima para fazer uma sequência decrescente? E para gerar a tabuada de um número dado pelo usuário? Você precisará mexer na expressão lógica do loop e no incremento de numero.

Em expressões onde uma variável aparece de ambos os lados, podemos utilizar uma abreviação. Por exemplo, a expressão x = x + 5 Pode ser reescrita como: x += 5 O mesmo vale para todas as outras expressões aritméticas (subtração, multiplicação, divisão etc).

# Listas

Já fizemos alguns programas para ler 2 ou 3 notas e calcular a média. Inclusive já fomos além e aprendemos a verificar se o aluno passou ou não. Vamos rever um exemplo desses:

nota1 = float(input('Digite a primeira nota: '))

nota2 = float(input('Digite a segunda nota: '))

media = (nota1 + nota2)/2

print(media)

Simples, certo? Mas e se a regra da escola mudasse, e agora cada professor precisasse aplicar 4 provas? Modificaríamos nosso programa:

nota1 = float(input('Digite a primeira nota: '))

nota2 = float(input('Digite a segunda nota: '))

nota3 = float(input('Digite a terceira nota: '))

nota4 = float(input('Digite a quarta nota: '))

media = (nota1 + nota2 + nota3 + nota4)/4

print(media)

Até aqui tudo bem. Mas e se o objetivo fosse testar o quanto o professor conseguiu ensinar? Para isso, poderíamos calcular a média das médias de todos os alunos do professor. Mas e se o professor trabalha em uma faculdade muito grande e suas turmas tem 80 alunos?

aluno1 = float(input('Digite a nota do aluno 1: '))

aluno2 = float(input('Digite a nota do aluno 2: '))

aluno3 = float(input('Digite a nota do aluno 3: '))

aluno4 = float(input('Digite a nota do aluno 4: '))

aluno5 = float(input('Digite a nota do aluno 5: '))

aluno6 = float(input('Digite a nota do aluno 6: '))

aluno7 = float(input('Digite a nota do aluno 7: '))

aluno8 = float(input('Digite a nota do aluno 8: '))

aluno9 = float(input('Digite a nota do aluno 9: '))

aluno10 = float(input('Digite a nota do aluno 10: '))

# ...

```

Nota do programador: eu me demito.

Não ganho bem o suficiente pra ISSO!

```

Para trabalhar com poucos valores, é fácil e conveniente criar uma variável para cada valor e realizar operações individualmente sobre cada uma. Porém, dizemos que esse tipo de solução não é escalável: o programa não está preparado para lidar com variações no tamanho da base de dados, e modificá-lo para comportá-las pode ser difícil, trabalhoso ou mesmo inviável. Imagine se para cada novo perfil em uma rede social o estagiário precisasse criar uma variável nova para o nome, uma para o e-mail, uma para a data de nascimento, e assim sucessivamente... E depois ainda precisasse de linhas novas de código para ler cada um desses valores do novo usuário!

## 1. Listas

É aí que entram as listas. Listas são coleções de objetos em Python. Falando de maneira simplificada, são variáveis que comportam diversos valores ao mesmo tempo. Vejamos alguns jeitos de criar listas em Python:

primeiralista = [] # cria uma lista vazia

segundalista = list() # cria uma lista vazia

terceiralista = [1, 3.14, 5, 7, 9, 'onze'] # lista com valores

Note que podemos misturar tipos de dados. A terceiralista possui 4 int, um float e uma str.

Bom, e agora, como fazemos para acessar cada valor? Podemos imaginar a lista da seguinte maneira: imagine que ao invés de ter uma caixa para guardar cada item, temos uma cômoda com várias gavetas. Cada item está em uma gaveta. Não estamos acostumados a dizer que algo está na terceira gaveta do armário? A ideia é a mesma: a lista é uma coleção indexada, ou seja, podemos acessar cada elemento através de índices, que são números indicando a posição. A indexação é automática e começa a partir do zero:

| elemento | 1 | 3.14 | 5 | 7 | 9 | 11 |
| --- | --- | --- | --- | --- | --- | --- |
| índice | 0 | 1 | 2 | 3 | 4 | 5 |

Portanto, para acessar o elemento "7" da nossa lista, utilizaríamos o índice 3. Informamos o índice entre colchetes:

terceiralista = [1, 3.14, 5, 7, 9, 'onze'] # lista com valores

print(terceiralista[3])

A lista é mutável. Isso significa que podemos modificar os valores já existentes:

terceiralista = [1, 3.14, 5, 7, 9, 'onze'] # lista com valores

terceiralista[3] = 'sete' # troca 7 por 'sete' na lista

print(terceiralista)

# 2. Quebrando listas

É possível pegar subconjuntos de nossas listas utilizando o conceito de slices. Ao invés de passar apenas 1 valor entre colchetes (o índice desejado), podemos passar faixas de valores. Veja o exemplo abaixo:

impares = [1, 3, 5, 7, 9, 11, 13, 15, 17]

meio = impares[3:6]

print(meio) # resultado na tela: [7, 9, 11]

O primeiro valor é o índice inicial da sublista a ser gerada, e o segundo é o índice final (exclusivo). Podemos omitir um desses valores para indicar que será desde o início ou até o final:

impares = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

primeirametade = impares[:5]

segundametade = impares[5:]

print(primeirametade) # resultado: [1, 3, 5, 7, 9]

print(segundametade) # resultado: [11, 13, 15, 17, 19]

Além de índices inicial e final, podemos também passar um passo para os índices. Veja o exemplo abaixo:

numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

# múltiplos de 3 abaixo de 10:

mult3sub10 = numeros[3:10:3]

print(mult3sub10) # resultado: [3, 6, 9]

# 3. Percorrendo listas

Suponha que você queira acessar cada elemento de sua lista individualmente. Digitar todos os índices manualmente cancelaria a escalabilidade do programa, certo? Portanto, podemos usar um loop para gerar os índices:

pares = [0, 2, 4, 6, 8]

tamanho = len(pares)

# tamanho vale 5, logo indice recebe os valores 0, 1, 2, 3 e 4

for indice in range(tamanho):

print(pares[indice])

Porém, tem um jeito ainda mais fácil de percorrer a lista. O for não serve apenas para gerar sequências numéricas junto do range: ele serve para percorrer coleções. Portanto, podemos trocar o range pela própria lista:

pares = [0, 2, 4, 6, 8]

for elemento in pares:

print(elemento)

Assim como no caso das contagens, "elemento" é apenas uma variável que será criada de forma automática e poderia ter qualquer nome. Em cada repetição do loop, um valor diferente da lista será copiado para elemento.

Importante: Como os elementos são copiados, caso você modifique o valor de elemento você não irá modificar o valor na lista, e sim uma cópia dele. Além disso, como este loop serve especificamente para percorrer listas, se dentro dele você fizer operações que alterem o tamanho da lista (append ou remove, por exemplo), o loop poderá executar incorretamente, pulando ou repetindo elementos.

# Funções de listas

As listas possuem diversas funções prontas bastante úteis. Veremos algumas das mais usadas. Não se preocupe em decorar todas elas: sempre podemos consultar nosso material quando precisarmos de um lembrete! Com tempo e prática você irá aos poucos memorizar algumas delas.

## 1. Adicionando elementos

Podemos adicionar novos elementos na lista de duas maneiras. A primeira delas, mais simples, é o append. Ele adiciona um elemento ao final da lista. Veja o exemplo abaixo:

pares = [0, 2, 4, 6, 8]

pares.append(10)

print(pares) # resultado: [0, 2, 4, 6, 8, 10]

Outra maneira é com o insert: além do elemento, ele recebe a posição do novo elemento.

pares = [0, 2, 4, 8, 10]

pares.insert(3, 6)

print(pares) #resultado: [0, 2, 4, 6, 8, 10]

## 2. Removendo elementos

Podemos remover o elemento de 2 jeitos: por valor e por posição. O remove irá remover o primeiro elemento encontrado na lista com um dado valor. Ex:

impares = [1, 3, 3, 5, 7, 9]

impares.remove(3)

print(impares) # resultado: [1, 3, 5, 7, 9]

O pop remove o elemento que estiver em uma dada posição, independentemente de seu valor:

impares = [1, 3, 5, 7, 8, 9]

impares.pop(4)

print(impares) # resultado: [1, 3, 5, 7, 9]

Se nenhum valor for passado no pop, ele irá remover necessariamente o último elemento da lista.

## 3. Ordenando a lista

Podemos ordenar a lista usando o sort.

fibonacci = [8, 1, 0, 5, 13, 1, 3, 2]

fibonacci.sort()

print(fibonacci) # resultado: [0, 1, 1, 2, 3, 5, 8, 13]

Caso desejássemos ordenar em ordem decrescente, podemos passar a opção reverse = True para o sort:

fibonacci = [8, 1, 0, 5, 13, 1, 3, 2]

fibonacci.sort(reverse = True)

print(fibonacci) # resultado: [13, 8, 5, 3, 2, 1, 1, 0]

Importante: o sort só irá funcionar caso sua lista possua apenas elementos que podem ser comparados entre si (apenas strings ou apenas números, por exemplo). Se uma lista contém tanto strings quanto números, o sort não saberá o que vem primeiro.

Falando em ordem decrescente, é possível simplesmente inverter a ordem dos elementos de uma lista utilizando o reverse:

lista = [1, 5, 'dois', 4, 3.14]

lista.reverse()

print(lista) # resultado: [3.14, 4, 'dois', 5, 1]

## 4. Buscando um elemento

Podemos buscar um elemento (descobrir sua posição) utilizando a função index. Ela irá informar a primeira posição onde um elemento for encontrado:

pi = [3, 1, 4, 1, 5, 9, 2, 6, 5]

posicao = pi.index(5)

print(posicao) # resultado: 4

## 5. Informações sobre a lista

Podemos obter alguns dados sobre a nossa lista: seu tamanho atual (len), seu maior valor (max) e seu menor valor (min).

pi = [3, 1, 4, 1, 5, 9, 2, 6]

tamanho = len(pi)

maior = max(pi)

menor = min(pi)

print(tamanho, maior, menor) # resultado: 8 9 1

# Malhas de repetição com contador

No capítulo de malhas de repetição vimos casos onde precisamos contar quantas vezes o loop se repete, e parar quando a contagem atingir um certo valor. Em outras ocasiões, apenas precisamos de algum tipo de sequência numérica. Nestes casos, era normal utilizar uma variável como contador, incrementá-la em cada passo e utilizar seu valor como condição de parada. O exemplo abaixo imprime todos os números pares entre 0 e 100:

contador = 0

while contador < 100:

print(contador)

contador = contador + 2

Existe um meio de automatizar todas as operações envolvidas: atribuir um valor inicial, atribuir um valor final e realizar o incremento.

## 1.1 Loops do tipo "para"

Dizemos que o exemplo acima é um loop do tipo "para": para contador de 0 até 100 com passo 2 faça: imprima contador. Em Python, podemos criar esse tipo de loop utilizando os comandos for e range. O exemplo abaixo imprime os números de 0 até 9 na tela:

for contador in range(10):

print(contador)

A palavra "contador" é apenas uma variável. Ela não precisa ser criada previamente: qualquer nome utilizado nesta construção será automaticamente inicializado pelo for. O programa acima atribui o valor inicial 0 à variável. Em seguida, ele executa tudo que vier dentro do loop, e ao chegar ao final, ele retorna ao início, soma 1 na variável e testa se o seu valor atingiu o número entre parênteses. Caso não tenha atingido, ele repete a execução. Dizemos que aquele número é o valor final exclusivo (pois o loop exclui esse valor).

De forma geral, tudo que vier dentro de um for contador in range(x) irá executar "x" vezes. É o jeito fácil de dizer "repita essas linhas x vezes" em Python.

## 1.2 Parâmetros do range

Foi dito que loops do tipo "para" seguem a forma "para contador de X até Y passo Z faça:". No exemplo acima, os valores inicial (0) e passo (1) foram atribuídos de forma automática. Caso eles sejam omitidos, 0 e 1 são os valores padrão, respectivamente. Porém, podemos determiná-los, se necessário. O exemplo abaixo inicia a impressão dos números em 1 ao invés de 0:

for contador in range(1, 10):

print(contador)

Dizemos que esse loop possui valor inicial 1, valor final (exclusivo) 10 e passo 1. Assim como manipulamos o valor inicial e o final, podemos manipular também o passo. Veja o exemplo abaixo:

for contador in range(0, 100, 2):

print(contador)

Note que o resultado dele na tela é exatamente o mesmo do exemplo com while do início deste capítulo! Valor inicial 0, valor final (exclusivo) 100 e passo 2. Porém, não precisamos nos preocupar em criar o contador, atribuir valor inicial, incrementar e criar uma condição de parada. Apenas colocamos os números dentro do range e ele fez a mágica por nós.

Quantos exercícios de while você fez que podem ser resolvidos de maneira mais fácil com o for?

Dica: é possível utilizar o for para gerar sequências numéricas decrescentes também. Basta adotar valor final menor do que o inicial e incremento negativo.

for contador in range(20, 0, -1):

print(contador)

Qual valor será excluído da sequência: o 20 ou o 0? Tente deduzir e execute o programa para ver se acertou!

# Strings

## 1. Strings

Na primeira aula foi mencionado que temos quatro tipos de dados: inteiro, real, lógico e texto/literal (string). Na verdade, o quarto tipo básico seria um caractere. Uma string é uma coleção de caracteres - como se fosse uma lista, mas aceitando apenas elementos textuais. Vamos verificar algumas propriedades das strings!

### 1.1. Acessando elementos em uma string

No capítulo sobre Listas, vimos que podemos acessar elementos individuais de uma lista utilizando um índice entre colchetes. Vamos testar a mesma lógica com strings?

frase = "Let's Code"

print(frase[0])

print(frase[1])

print(frase[2])

print(frase[3])

print(frase[4])

Note que o programa acima imprime "Let's", com um caractere por linha. Ou seja, ele considerou frase[0] como "L", frase[1] como "e", e assim sucessivamente. Uma string é, de fato, uma coleção de caracteres.

Porém, ao contrário de uma lista, dizemos que uma string é imutável. Isso significa que não podemos alterar um elemento individual da string. O programa abaixo produz um erro:

frase = "let's code"

frase[0] = 'L'

Para alterar uma string, é necessário redefiní-la, de modo que a string original será descartada e a nova (alterada) será escrita por cima da original. Ou, alternativamente, podemos gerar uma cópia da string com alterações. Veremos mais detalhes adiante.

### 1.2. Operações entre strings

Alguns operadores aritméticos funcionam com strings também. Naturalmente, eles não servem para fazer contas, mas nos permitem fazer de forma intuitiva algumas operações bastante úteis.

O operador + serve como um operador de concatenação de strings: unir duas strings. Observe o exemplo abaixo:

string1 = 'Olá'

string2 = 'Mundo'

resultado = string1 + string2

print(resultado) # Na tela: 'OláMundo'

Outro operador que funciona é o operador \*. Este operador não é usado entre duas strings, mas entre uma string e um int. Ele repetirá a string o número de vezes dado pelo int.

string = 'Olá mundo!'

multi = string \* 3

print(multi) # Na tela: 'Olá mundo!Olá mundo!Olá mundo!'

#### 1.2.1. Copiando uma string através de concatenação

Caso você já tenha resolvido problemas de somatório (a essa altura, espera-se que tenha resolvido vários!), você já deve estar acostumado a utilizar um loop onde novos valores são somados em uma mesma variável. Somar os números de uma lista, por exemplo, tem mais ou menos essa carinha:

soma = 0

for numero in lista:

soma = soma + numero

A mesma lógica pode ser aplicada a uma string:

stringInicial = 'Olá Mundo'

stringFinal = '' # cria uma string vazia

for letra in stringInicial:

stringFinal = stringFinal + letra

print(stringFinal)

Isso é útil porque antes de "somar" cada letra à string final podemos fazer alterações (como transformar em maiúscula ou minúscula, acrescentar caracteres entre 2 letras etc). É um jeito de fazer tratamento de strings. Veremos mais sobre tratamento de strings no capítulo de funções de strings.

## 1.3. Transformando uma string em lista

Strings são imutáveis, e isso pode nos dar um pouco de trabalho quando queremos fazer pequenas alterações, como forçar um caractere a ser maiúsculo ou acrescentar um caractere à string. Uma das formas de fazer envolve a "soma cumulativa" apresentada acima. Outra forma envolve transformar a nossa string em lista, que é uma estrutura mutável. Execute o programa abaixo:

string = "let's Code"

lista = list(string)

lista[0] = 'L'

lista.append('!')

print(lista)

# resultado:

# ['L', 'e', 't', "'", 's', ' ', 'C', 'o', 'd', 'e', '!']

Como a lista é mutável, nela conseguimos alterar uma letra e adicionar um símbolo ao final sem dificuldades! Porém, infelizmente nosso resultado é uma lista, o que não ficou muito legível para o usuário. Podemos resolver isso utilizando a função join. Veremos em breve como ele realmente funciona, mas por hora podemos utilizá-lo da seguinte maneira para transformar lista em string:

stringOriginal = "let's Code"

lista = list(stringOriginal)

lista[0] = 'L'

lista.append('!')

stringFinal = ''.join(lista) # antes do . temos uma string vazia

print(stringFinal)

# resultado: "Let's Code!"

Para as modificações mais comuns, temos algumas funções prontas que poderão ser bastante úteis!

## 2. Símbolos especiais

Além de letras, números, sinais de pontuação, símbolos matemáticos etc, uma string pode conter alguns operadores especiais de controle. Esses operadores podem indicar, por exemplo, uma quebra de linha ou uma tabulação. Vejamos os mais comuns:

### 2.1. Quebra de linha

Uma quebra de linha indica que o programa exibindo a string deverá quebrar a linha atual e exibir o restante da string na linha seguinte, e é representada na maioria dos sistemas e na web pelo símbolo \n. Execute o programa abaixo e veja o resultado na tela:

print('Olá\nMundo')

### 2.2. Tabulação

A tabulação indica um recuo equivalente ao da tecla Tab - um recuo de início de parágrafo, ou o recuo que usamos para aninhar linhas de código em Python. Ela é representada pelo símbolo \t. Verifique o resultado do exemplo abaixo:

aprovados = ['Mario', 'Peach', 'Luigi']

reprovados = ['Wario', 'Bowser']

print('Candidatos aprovados:')

for nome in aprovados:

print('\t', nome)

print('Candidatos reprovados:')

for nome in reprovados:

print('\t', nome)

### 2.3. Barra

E se nós quiséssemos representar uma string que explica o significado de \n, por exemplo, como proceder? Afinal, ao ver o símbolo \n o programa entenderá que é uma quebra de linha e fará isso ao invés de escrever \n na tela.

Podemos utilizar 2 barras: \\. Ao fazermos isso, o programa entende que é para representar a barra na tela ao invés de interpretá-la como início de outro símbolo especial.

print('Utilizamos o \\n para quebrar linhas.')

### 2.4 Aspas

Um problema que você deve ter se deparado é que parece impossível representar o símbolo ' em uma string que foi aberta por esse símbolo, já que a segunda ocorrência dele fechará a string. Idem para o símbolo ". Podemos resolver isso da mesma forma que fizemos com a barra: \' irá sempre representar o símbolo ' e \" irá sempre representar o símbolo " ao invés de fechar uma string.

print('Imprimindo uma aspa simples(\') dentro de uma string sem problemas')

print("Imprimindo aspas duplas(\") dntro de uma string sem problemas")

# Funções de strings

## 1. Funções de strings

É possível fazer várias operações com strings utilizando técnicas como concatenação ou converter em listas. Porém, certas operações são muito comuns e podem ser muito trabalhosas de fazer na mão. Por isso, temos diversas funções prontas para nos ajudar.

Note que como strings são imutáveis, nenhuma dessas funções irá alterar a string original. Elas sempre retornarão uma string nova com as modificações desejadas.

Vejamos algumas das mais usadas.

### 1.1. Maiúsculas e minúsculas

Temos algumas funções prontas para alterar a capitalização das letras. Uma delas é a função upper, que transforma todas as letras da string original em maiúsculas:

frase = 'vAmOs PrOgRaMaR'

maiuscula = frase.upper()

print(maiuscula) # resultado: 'VAMOS PROGRAMAR'

Analogamente, temos a função lower para transformar todas as letras em minúsculas:

frase = 'vAmOs PrOgRaMaR'

minuscula = frase.lower()

print(minuscula) # resultado: 'vamos programar'

Também é possível formatar a string inteira como um nome próprio: primeira letra de cada palavra maiúscula, todo o restante minúscula. Para isso temos a função title:

frase = 'vAmOs PrOgRaMaR'

titulo = frase.title()

print(titulo) # resultado: 'Vamos Programar'

E, por fim, é possível tratar nossa string como uma frase gramaticalmente correta: primeira letra maiúscula, todo o resto minúscula. Essa função é a capitalize:

frase = 'vAmOs PrOgRaMaR'

correta = frase.capitalize()

print(correta) # resultado: 'Vamos programar'

Uma utilidade para essas funções é padronizar entrada de usuário. Quando pedimos para o usuário digitar 'sim' caso ele deseje fazer algo, ele pode digitar 'SIM', 'sim', 'Sim', 'sIm', 'siM', 'SIm', 'sIM' ou 'SiM'. Prever todas essas condições em uma condicional pode ser bastante trabalhoso, ou mesmo impossível. Imagine se fosse uma string de várias letras... Porém, podemos forçar um padrão para a entrada do usuário e comparar com esse padrão:

usuario = input('Digite "sim" se aceita os termos de uso: ')

if usuario.upper() == 'SIM':

print('Seja bem-vindo!')

else:

print('Que pena.')

Qualquer forma que o usuário digite a palavra 'sim' será aceita.

### 1.2. Quebrando uma string

É possível separar uma string em uma lista de substrings. Isso pode ser particularmente útil quando precisamos separar um texto em palavras individuais. A função que realiza essa quebra é o split:

texto = 'uma frase qualquer'

palavras = texto.split()

print(palavras) # resultado: ['uma', 'frase', 'qualquer']

O split é mais do que apenas uma função para separar palavras. Podemos opcionalmente passar como parâmetro uma string para ser usada como critério de separação: ao invés de quebrar no espaço em branco, a string principal será quebrada nos pontos onde o parâmetro aparece (e ele será apagado do resultado final).

Uma utilidade interessante para isso seria quando estamos interessados em ler dados formatados do teclado ou de um arquivo e pegar as informações que nos interessam. Imagine, por exemplo, que você queira que o usuário digite uma data no formato 'dd/mm/aaaa' e em seguida você precise separar dia, mês e ano em três variáveis do tipo int. Isso é possível com o split:

data = input('Digite uma data: ')

listaData = data.split('/')

dia = int(listaData[0])

mes = int(listaData[1])

ano = int(listaData[2])

print('Dia: ', dia)

print('Mês: ', mes)

print('Ano: ', ano)

### 1.3. Substituindo elementos na string

Uma das ferramentas mais úteis em qualquer editor de texto é o localizar e substituir, onde podemos buscar por todas as ocorrências de uma expressão no texto e trocar por outra expressão. Em Python temos uma função análoga, o replace. Ele recebe 2 parâmetros: a expressão a ser substituída e a expressão que a substituirá. Veja o exemplo:

frase = 'Python é difícil. Por ser difícil, devemos estudar.'

corrigida = frase.replace('difícil', 'fácil')

print(corrigida)

# resultado: 'Python é fácil. Por ser fácil, devemos estudar.'

Em Python não existe uma função para deletar um pedaço de uma string. Porém, podemos usar o replace para substituir uma expressão por uma string vazia, o que tem o mesmo resultado:

palavra = 'batata'

consoantes = palavra.replace('a', '')

print(consoantes)

# resultado: 'btt'

### 1.4. Concatenando strings em uma coleção

Imagine que você tem uma coleção (por exemplo, uma lista) de strings e precisa unir todas elas utilizando algum símbolo padrão como separador entre elas. Para isso temos o join. Ele soa pouco intuitivo no começo, então convém executar o exemplo e observar com atenção seu resultado:

lista = ['a', 'b', 'c']

separador = '123'

resultado = separador.join(lista)

print(resultado) # resultado: 'a123b123c'

Já vimos o join antes sendo usado para converter uma lista de volta em string. Para isso, utilizávamos uma string vazia como separador. Assim, os elementos da lista eram concatenados sem separador.

Dica: um jeito de memorizar facilmente como o join funciona é pensar que o separador entrará no lugar das vírgulas na visualização da lista.

### 1.5. Formatando strings

Todo mundo que já preencheu um contrato ou uma ficha de cadastro está familiar com textos nesse estilo:

Eu, \_\_, portador do CPF \_\_\_, residente no endereço \_\_\_\_\_ autorizo o procedimento.

Esse é o texto genérico que vale para todos, e cada um de nós em particular entende que deve preencher os campos em brancos com dados específicos (nome, CPF e endereço, no exemplo acima).

Existe uma função em Python para realizar esse tipo de preenchimento de texto: o format. Suponha que você tenha dados em diferentes variáveis e precisa que todos eles apareçam em uma string. Basta criar uma string com os "espaços em branco" para serem preenchidos e passar as variáveis para a função. Os espaços em branco são representados por chaves ({}).

nome = input('Digite seu nome: ')

cpf = input('Digite seu cpf: ')

endereco = input('Digite seu endereço: ')

contrato = 'Eu, {}, portador do CPF {}, residente no endereço {} autorizo o procedimento.'

contratoPreenchido = contrato.format(nome, cpf, endereco)

print(contratoPreenchido)

Porém, o grande charme não está apenas em preencher - isso poderia ser feito concatenando com o operador +. Nós podemos colocar opções de formatação nos nossos dados, como número de casas em um número.

Imagine que você queira exibir uma data no formato dd/mm/aaaa. Em situações normais, dias e meses inferiores a 10 apareceriam com apenas 1 casa (int não é representado com zeros à esquerda). Porém, podemos especificar no format que gostaríamos de representar um inteiro com 2 casas, preenchendo com zero casas em branco.

dia = 1

mes = 2

ano = 2020

data = '{:02d}/{:02d}/{:04d}'.format(dia, mes, ano)

print(data) # resultado: 01/02/2020

Vamos entender o que está dentro das chaves: o símbolo : indica que passaremos opções. O símbolo 'd' indica que estamos representando números inteiros em base decimal (dígitos de 0 a 9). Os símbolos '2' e '4' indicam, respectivamente, 2 casas ou 4 casas. E o símbolo '0' indica que se faltar dígitos, os espaços devem ser preenchidos com zero.

Vejamos outro exemplo, dessa vez com casas decimais. É normal postos de gasolina mostrarem o preço do litro com 3 casas decimais. Mas o preço final a ser cobrado deverá ter 2 casas. Porém, ao multiplicar o preço por litro pelo valor em litros, é provável que o total dê várias casas decimais. Usaremos o format para representar com apenas 2 casas.

precoLitro = 5.234

litros = 29.5

total = precoLitro \* litros

print(total) # resultado: 154.403

precoFinal = 'R$ {:.2f}'.format(total)

print(precoFinal) # resultado: R$ 154.40

Neste caso, o 'f' indica que o número é float. Já o '.2' indica que queremos 2 casas após o ponto decimal. Note que ele não apenas descarta as casas excedentes, e sim arredonda corretamente o número.

O format possui tantas opções diferentes que existe um site inteiro dedicado a explicar e dar exemplos: <https://pyformat.info/>

# Estruturas de dados

Em aulas anteriores, sempre que precisamos trabalhar com múltiplos dados, utilizamos as listas para armazená-los. Porém, o Python possui outras coleções que podem ser úteis em situações diferentes. Vejamos um pouquinho sobre duas estruturas diferentes: tuplas e dicionários.

## 1. Tuplas

Tuplas são estruturas bastante parecidas com listas:

* Podem guardar diversos dados.
* Podem guardar tipos diferentes de dados.
* São indexadas (podemos acessar elementos individuais por índices).
* São iteráveis (podemos percorrer com for).

Porém, elas possuem uma grande diferença: elas são imutáveis. Assim como no caso das strings, após a criação da tupla você não pode alterar elementos individuais, adicionar elementos, remover elementos ou alterar a ordem dos elementos.

Bom, por que usaríamos algo parecido com uma lista mas com menos recursos? Temos 3 bons motivos:

* 1. É um jeito de sinalizar que esses dados não deveriam ser alterados. Apesar de ser possível redefinir a tupla (assim como já fizemos com strings), o programador que estiver mexendo no código alheio e fizer isso estará ciente de que talvez outras partes do código precisem daqueles dados inalterados.
  2. É um meio de garantir que os elementos estarão em uma ordem específica.
  3. Velocidade. Para que uma estrutura seja mutável, seu tamanho é variável, e isso envolve algoritmos complexos para que nenhum dado seja perdido. Para acessar um certo elemento de uma lista, todos os elementos anteriores serão percorridos. Já em uma tupla é possível um salto calculado direto para o elemento em questão. Em programas pequenos isso é pouco perceptível, mas ao trabalhar com bases de dados enormes e/ou realizar muitas operações complexas com os dados, a lista pode ser um gargalo, e a tupla seria a solução.

Vejamos um pequeno exemplo mostrando as diferentes operações que podemos fazer com tupla:

# para criar uma tupla utilizamos parênteses

tupla = (1, 2, 3, 5, 'sete', 11.0)

# note que para acessar um elemento pelo índice ainda usamos colchete

print(tupla[4])

print('Percorrendo a tupla...')

for elemento in tupla:

print(elemento)

# criando uma lista com os elementos da tupla

lista = list(tupla)

# adicionando um elemento na lista - ao contrário da tupla, a lista é mutável!

lista.append(13)

print(lista)

# criando uma tupla a partir de uma lista:

tuplanova = tuple(lista)

print(tuplanova)

## 2. Dicionários

Quando utilizamos um dicionário (o de papel, com definições), não temos o hábito de procurar pela palavra que está em uma determinada posição. Ao invés disso, nós buscamos pela palavra em si, e ao encontrá-la ela contém uma definição.

A estrutura dicionário em Python é uma coleção de dados. Porém, ela não é indexada. Ao adicionarmos elementos em um dicionário, sempre o fazemos aos pares: todo elemento terá uma chave e um valor.

A chave será uma string que utilizaremos como se fosse o índice. É como se fosse a palavra que buscamos em um dicionário de papel.

O valor pode ser qualquer dado: um int, um float, uma str, um bool, uma lista, uma tupla, outro dicionário... Ele é como se fosse a definição que encontramos vinculada à palavra que encontramos no dicionário de papel.

dicionario = {'curso':'Python Pro', 'escola':"Let's Code"}

print(dicionario['curso'])

print(dicionario['escola'])

### 2.1. Adicionando elementos em um dicionário

Para adicionar elementos, não precisamos de uma função pronta (como o append das listas). Basta "acessar" a nova chave e atribuir um novo valor.

dicionario = {'curso':'Python Pro', 'escola':"Let's Code"}

dicionario['unidade'] = 'Faria Lima'

print(dicionario)

### 2.2. Percorrendo um dicionário

Dicionários podem ser percorridos com um for. Ao fazer isso, as chaves serão percorridas, não os valores. Porém, à partir da chave obtém-se o valor:

dicionario = {'curso':'Python Pro', 'escola':"Let's Code"}

dicionario['unidade'] = 'Faria Lima'

for chave in dicionario:

print(chave, '--->', dicionario[chave])

### 2.3. Separando chaves e valores

O Python possui funções para obter, separadamente, todas as chaves ou todos os valores de um dicionário. Elas são, respectivamente, keys e values. Podemos transformar o retorno dessa função em uma lista ou tupla.

dicionario = {'curso':'Python Pro', 'escola':"Let's Code"}

dicionario['unidade'] = 'Faria Lima'

chaves = list(dicionario.keys())

valores = list(dicionario.values())

print('Chaves: ', chaves)

print('Valores:', valores)

### 2.4. Testando a existência de uma chave

Antes de criar uma chave nova em um dicionário, convém testar se ela já existe, para evitar sobrescrever um valor. Podemos fazer isso com o operador in (sim, o mesmo que usamos no for!). Neste contexto, ele retornará True se a chave existir e False caso não exista.

Vamos supor que no exemplo abaixo o valor de 'cursos' seja uma lista com todos os cursos que o usuário está fazendo.

dicionario = {'escola':"Let's Code", 'unidade':'Faria Lima'}

# Neste caso, 'cursos' ainda não existe.

# Cairemos no else e será criada uma lista com a string 'Python'.

if 'cursos' in dicionario:

dicionario['cursos'].append('Python')

else:

dicionario['cursos'] = ['Python']

# Agora a chave já existe.

# Portanto, será adicionado 'Data Science' à lista.

if 'cursos' in dicionario:

dicionario['cursos'].append('Data Science')

else:

dicionario['cursos'] = ['Data Science']

print(dicionario)

Experimente o operador in em uma lista, tupla ou string. Ele pode ser bastante útil!

# Funções

Imagine que você fez um programinha para gerar estatísticas sobre vários dados dos funcionários: média dos salários, média de vendas, média de feedback positivo, média de feedback negativo, média de notas atribuídas pelos gestores... Você tem uma lista com os salários, uma lista com o total de vendas de cada funcionário, e assim sucessivamente. Então você fez o seguinte trecho de código:

soma = 0

for elemento in lista:

soma = elemento

media = soma/len(lista)

Em seguida, você copiou e colou esse trecho de código várias vezes mudando "lista" pelo nome de cada lista individual, e "media" pelo nome do atributo. Trabalhoso, certo? Imagine que agora você percebeu o erro no trecho acima, e terá que sair corrigindo em todos os lugares onde colou o código errado. Imagina que conveniente se você pudesse arrumar apenas uma vez e todas as ocorrências fossem corrigidas automaticamente...

## 1. Funções

Uma função é um pedacinho de programa. Nós podemos dar um nome para nossa função, e toda vez que precisarmos que esse pedacinho de programa seja executado, nós o chamamos pelo nome.

Com isso, podemos evitar repetição de código, tornando nossos códigos mais enxutos e legíveis. Além disso, fica mais fácil corrigir problemas como o ilustrado no início deste capítulo.

### 1.1. Criando funções

Em Python, podemos criar funções com o comando def, e em seguida damos um nome para nossa função.

def minhaPrimeiraFuncao():

print('Olá Mundo')

Se você executar o código acima, o que aparecerá na tela? Nada. Tudo que o código acima faz é definir minhaPrimeiraFuncao, mas ela só será executada quando for chamada pelo nome.

# criando a função

def minhaPrimeiraFuncao():

print('Olá Mundo')

# o programa começa de verdade aqui:

minhaPrimeiraFuncao() # chamada para a função

Quando chamamos uma função, a execução do programa principal é pausada, o fluxo de execução é desviado para a função, e ao final dela ele retornará para o ponto onde parou. Veja o exemplo abaixo:

# criando a função

def minhaPrimeiraFuncao():

print('Olá Mundo')

# o programa começa de verdade aqui:

print('aaa')

minhaPrimeiraFuncao() # chamada para a função

print('bbb')

O resultado na tela será:

aaa

Olá Mundo

bbb

### 1.1.2. Parâmetros de uma função

Nossas funções devem ser tão genéricas quanto possível se quisermos reaproveitá-las ao máximo.

Um dos pontos onde devemos tomar cuidado é na entrada de dados da função: se usarmos um input dentro da função, teremos uma função que resolverá um certo problema desde que o usuário vá digitar os dados do problema. Mas e se quisermos usar a função em um trecho do programa onde o usuário digita os dados e em outro ponto onde os dados são lidos de um arquivo?

Podemos resolver isso fazendo a leitura de dados no programa principal, fora de nossa função, e então passaremos os dados para a função. Dados passados para a função são chamados de parâmetros ou argumentos de uma função. Observe o exemplo abaixo:

def soma(a, b):

resultado = a + b

print(resultado)

soma(3, 2) # resultado na tela: 5

soma(4, 7) # resultado na tela: 11

x = 5

soma(10, x) # resultado na tela: 15

Quando colocamos "a" e "b" entre parênteses na criação da função, estamos especificando que a função recebe 2 parâmetros. O primeiro valor que for passado entre parênteses para nossa função será referenciado por "a" e o segundo será referenciado por "b". É como se "a" e "b" fossem variáveis que vão receber a cópia dos valores passados para a função. Note que podemos passar valores puros ou variáveis (como fizemos com "x" na última linha), e não precisamos criar variáveis "a" e "b" em nosso programa principal para "casar" com os parâmetros da função.

### 1.1.3. Retorno de uma função

Certas funções possuem uma "resposta": elas resolvem um problema (por exemplo, uma equação matemática) e nós estamos interessados no resultado. No exemplo anterior, tínhamos uma soma e nós imprimíamos a soma na tela.

Porém, ainda pensando na questão da função ser genérica: será que nós sempre queremos o resultado na tela? Imagine que você esteja utilizando a fórmula de Bháskara para resolver uma equação de segundo grau. No meio da fórmula existe uma raiz quadrada. Nós não queremos o resultado da raiz quadrada na tela, nós queremos o resultado dentro do nosso programa em uma variável para jogar em outra equação.

Bom, parece fácil: vamos tentar pegar o resultado fora da função... Certo?

def soma(a, b):

resultado = a + b

media = resultado/2

print(media)

Se você executar o programa acima, verá uma mensagem de erro dizendo que "resultado" não existe. Toda variável criada dentro de uma função é privada. Ela só pode ser acessada dentro da função e será destruída ao final da execução da função. Para disponibilizar para o programa um valor que foi gerado dentro da função, utilizamos o comando return.

def soma(a, b):

resultado = a + b

return resultado

s = soma(10, 5)

media = s/2

print(media)

Quando fizemos s = soma(10, 5), a função soma foi chamada, e ao final da execução, s recebeu o valor retornado por ela. Deste ponto em diante podemos utilizar a "resposta" da nossa função em nosso programa principal.

O return, além de disponibilizar um valor, encerra a execução da função. Se a sua função possuir outras linhas após o return, elas serão ignoradas.

## 2. Recursividade

Uma função pode chamar outra função? Sim. Rode o programa abaixo e observe que ele funciona:

def soma(a, b):

resultado = a + b

return resultado

def media(x, y):

s = soma(x, y)

resultado = s/2

return resultado

m = media(10, 5)

print(m)

Mas e se uma função referenciasse ela mesma? Isso também funciona, e chama-se função recursiva, ou recursão.

A ideia vem da matemática. Vejamos um exemplo. Imagine uma função matemática f(x) que seja igual a 2 elevado a x. Podemos reescrevê-la da seguinte maneira:

f(x) =

* 1, se x = 0
* 2 \* f(x-1), se x > 0

Ou seja, imagine que você queira calcular f(4). Como 4 > 0, teremos: f(4) = 2 \* f(3) Precisamos expandir f(3): f(4) = 2 \* (2 \* f(2)) E assim sucessivamente: f(4) = 2 \* (2 \* (2 \* f(1))) f(4) = 2 \* (2 \* (2 \* (2 \* f(0)))) Opa, f(0) nós conhecemos: está definido lá em cima como 1. Portanto: f(4) = 2 \* 2 \* 2 \* 2 \* 1 f(4) = 16

Note que nós decompomos um problema em várias instâncias "menores" do problema. Quebramos a formulação de uma multiplicação enorme por vários casos de 2 x f(n). Chamamos essa estratégia de dividir para conquistar, e ela envolve identificar 2 etapas bastante claras do problema:

* Caso base: é um caso para o qual temos um valor conhecido (no exemplo acima, f(0) = 1)
* Caso geral: é a chamada recursiva, onde faremos referência à própria função.

Note também que esse comportamento tem o comportamento de pilha: se colocamos 3 pratos empilhados sobre a mesa, precisamos tirar primeiro o último que colocamos, certo? Caso contrário, a pilha toda tomba. No caso da recursão, para obter f(4) caímos em f(3), depois f(2), depois f(1), depois f(0) e foi para ele que obtivemos a primeira resposta, que em seguida usamos para calcular f(1), depois calcular f(2), depois f(3) e só então chegamos em f(4). O primeiro passo do problema foi o último a ser resolvido.

Em Python, nossa função ficaria assim:

def potenciadedois(x):

if x == 0:

return 1

else:

return 2\*potenciadedois(x-1)

Se chamarmos potenciadedois(4), o que acontecerá? O programa começará a executar a função, cairá no else e encontrará a função chamada novamente. Neste caso, ele salva x valendo 4 e salva que a execução foi interrompida nessa linha. Então ele cria um novo x valendo 3, cai novamente no else e salva que a execução foi interrompida nessa linha, e assim sucessivamente.

Note que para cada passo recursivo, as variáveis da função são copiadas e também é salvo o ponto onde a execução parou. Ou seja, funções recursivas podem consumir bastante memória, além de tempo de processamento para ficar criando cópias. A vantagem delas é o rigor matemático: podemos transcrever funções matemáticas quase exatamente como elas são, sem criar loops e variáveis para ficar guardando estados.

# Funções com parâmetros variáveis e retornos múltiplos

Quando estudamos funções, aprendemos que elas podem receber dados (parâmetros) e podem fornecer uma resposta (retorno). Porém, o número de parâmetros era fixo para cada função: um dado para cada parâmetro que declaramos na definição da função. Da mesma forma, a função poderia retornar exatamente um resultado.

Em alguns casos, mais flexibilidade seria útil. Utilizando tuplas e dicionários conseguimos essa flexibilidade.

## 1. Funções com retorno múltiplo

Vejamos um caso simples: uma função que retorna os valores máximo e mínimo de uma coleção. Você pode retornar os valores separados por vírgula. Vamos imprimir o resultado e verificar o que acontece.

def maxmin(colecao):

maior = max(colecao)

menor = min(colecao)

return maior, menor

numeros = [3, 1, 4, 1, 5, 9, 2]

resposta = maxmin(numeros)

print(resposta)

print(type(resposta)) # mostra o tipo da variável resposta

Surpresa! Ele tratou o retorno como uma tupla! Quando utilizamos valores separados por vírgula em Python, os valores são agrupados em uma tupla, mesmo que não estejamos utilizando parênteses. Essa informação é relevante porque podemos separar a tupla em varias variáveis usando a mesma sintaxe:

def maxmin(colecao):

maior = max(colecao)

menor = min(colecao)

return maior, menor

numeros = [3, 1, 4, 1, 5, 9, 2]

maiorNum, menorNum = maxmin(numeros)

print(maiorNum)

print(menorNum)

No exemplo acima é mais perceptível a sensação de que a função retornou 2 valores e o programa recebeu esses 2 valores individualmente. Por dentro, tupla. Por fora, retorno múltiplo.

## 2. Funções com quantidade variável de parâmetros

Talvez você já tenha notado que o print é uma função. Se não notou, esse é um bom momento para pensar a respeito. Nós sempre usamos com parênteses, nós passamos informações dentro dos parênteses (os dados a serem impressos) e ele faz um monte de coisa automaticamente: converte todos os dados passados para string, contatena todas as strings com um espaço entre elas e as escreve na tela.

Algo que o print tem que as nossas funções não tinham é a capacidade de receber uma quantidade variável de parâmetros. Nós podemos passar 0 dados (e, neste caso, ele apenas pulará uma linha), 1 dado, 2 dados, 3 dados... Quantos dados quisermos, separados por vírgula, e ele funcionará para todos esses casos. Se temos que declarar todos os parâmetros, como fazer para que múltiplos dados possam ser passados?

### 2.1. Agrupando parâmetros

A solução é utilizar o operador \* - que, neste caso, não será uma multiplicação. Ao colocarmos o \* ao lado do nome de um parâmetro na definição da função, estamos dizendo que aquele argumento será uma coleção. Mais especificamente, uma tupla. Porém, o usuário não irá passar uma tupla. Ele irá passar quantos argumentos ele quiser, e o Python automaticamente criará uma tupla com eles.

O exemplo abaixo cria uma função de somatório que pode receber uma quantidade arbitrária de números.

def somatorio(\*numeros):

# remova o símbolo de comentário das linhas abaixo para entender melhor o parâmetro

# print (numeros)

# print(type(numeros))

soma = 0

for n in numeros:

soma = soma + n

return soma

s1 = somatorio(5, 3, 1)

s2 = somatorio(2, 4, 6, 8, 10)

s3 = somatorio(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

print(s1, s2, s3)

### 2.2. Expandindo uma coleção

O exemplo acima funciona muito bem quando o usuário da função possui vários dados avulsos, pois ele os agrupa em uma coleção. Mas o que acontece quando os dados já estão agrupados?

def somatorio(\*numeros):

print (numeros)

print(type(numeros))

soma = 0

for n in numeros:

soma = soma + n

return soma

lista = [1, 2, 3, 4, 5]

s = somatorio(lista)

print(s)

Note que o programa dará erro, pois como os print dentro da função ilustram, foi criada uma tupla, e na primeira posição da tupla foi armazenada a lista. Isso não funciona com a lógica que projetamos.

Para casos assim, utilizaremos o operador \* na chamada da função também. Na definição, o operador \* indica que devemos agrupar itens avulsos em uma coleção. Na chamada, ele indica que uma coleção deve ser expandida em itens avulsos.

def somatorio(\*numeros):

print (numeros)

print(type(numeros))

soma = 0

for n in numeros:

soma = soma + n

return soma

lista = [1, 2, 3, 4, 5]

s = somatorio(\*lista)

print(s)

No programa acima, a lista é expandida em 5 valores avulsos, e em seguida a função agrupa os 5 itens em uma tupla chamada "numeros".

## 3. Parâmetros opcionais

Outra possibilidade são funções com parâmetros opcionais. Note que isso é diferente de termos quantidade variável de parâmetros.

No caso da quantidade variável, normalmente são diversos parâmetros com a mesma utilidade (números a serem somados, valores a serem exibidos etc).

Já os parâmetros opcionais são informações distintas que podem ou não ser passadas para a função. Como exemplo podemos citar o csv.reader e o csv.writer vistos anteriormente. Os parâmetros que passamos pelo nome (delimiter e lineterminator) são opcionais: se você omiti-los, a função usará valores padrão.

### 3.1. Criando parâmetros opcionais

Para criar parâmetros opcionais, usaremos \*\*, e os parâmetros passados serão agrupados em um dicionário: o nome do parâmetro será uma chave, e o valor será... O valor.

O exemplo abaixo cadastra usuários em uma base de dado. Um usuário pode fornecer seu nome, seu CPF ou ambos.

def cadastro(\*\*usuario):

if not ('nome' in usuario) and not ('cpf') in usuario:

print('Nenhum dado encontrado!')

else:

arquivo = open('usuarios.txt', 'a')

if 'nome' in usuario:

arquivo.write(usuario['nome'] + '\n')

if 'cpf' in usuario:

arquivo.write(str(usuario['cpf']) + '\n')

arquivo.write('-----\n')

arquivo.close()

print('Cadastro realizado com sucesso!')

cadastro(nome = 'João', cpf = 123456789) # tem ambos

cadastro(nome = 'José') # tem apenas nome

cadastro(cpf = 987654321) # tem apenas cpf

cadastro(rg = 192837465) # não tem nome nem cpf

### 3.2. Expandindo um dicionário

Analogamente ao caso dos parâmetros múltiplos, é possível que o usuário da função já tenha os dados organizados em um dicionário. Neste caso, basta usar \*\* na chamada da função para expandir o dicionário em vários parâmetros opcionais.

def cadastro(\*\*usuario):

if not ('nome' in usuario) and not ('cpf') in usuario:

print('Nenhum dado encontrado!')

else:

arquivo = open('usuarios.txt', 'a')

if 'nome' in usuario:

arquivo.write(usuario['nome'] + '\n')

if 'cpf' in usuario:

arquivo.write(str(usuario['cpf']) + '\n')

arquivo.write('-----\n')

arquivo.close()

print('Cadastro realizado com sucesso!')

maria = {'nome':'Maria', 'cpf':2468135790}

cadastro(\*\*maria)

## Bibliotecas (módulos)

Assim como pode ser útil reaproveitar um pedaço de código em vários pontos de um mesmo programa, pode ser útil reaproveitar esse pedaço em vários outros projetos também. Várias linguagens de programação oferecem o que chamamos de bibliotecas, e em Python comumente chamamos de módulos.

### 1. Criando módulos

Você pode criar algumas funções dentro de um arquivo. Ao colocar esse arquivo na mesma pasta que o restante do projeto, você pode usar o comando import para informar ao programa que ele deve buscar funções naquele arquivo. A partir deste momento, você terá acesso às funções daquele módulo.

# Salve este código como contas.py

def soma(a, b):

resultado = a + b

return resultado

def media(x, y):

s = soma(x, y)

resultado = s/2

return resultado

# Salve este outro código como principal.py na mesma pasta do contas.py

# Execute este código, não o contas.py

import contas

x = contas.media(3, 9)

print(x)

### 2. Módulos pré-instalados

A comunidade Python costuma descrevê-lo como uma linguagem com baterias inclusas. Com isso eles querem afirmar que já vem bastante coisa pronta "de fábrica" e você não precisa fazer aquisições adicionais antes de começar a fazer muita coisa.

Esse "bastante coisa" são bibliotecas prontas: The Python Standard Library. Tem biblioteca pra muita coisa: funções matemáticas, estatísticas, manipulação de arquivos, redes, comunicação HTTP, JSON, arquivos ZIP, arquivos CSV, codificação de texto, criptografia, HTML, interface gráfica... A lista vai embora.

Para usar os módulos pré-instalados, basta usar import e o nome do módulo - não é necessário mover nada para nenhuma pasta. O Python saberá aonde encontrá-los! No exemplo abaixo, vamos gerar um número aleatório entre 0 e 100 utilizando a função randrange, do módulo random. Execute o código várias vezes e veja o que acontece!

import random

sorteio = random.randrange(0, 100)

print(sorteio)

Como são vários módulos prontos, é difícil lembrar tudo que temos em mãos. Por isso, a documentação oficial recomenda manter o manual de referência dos módulos "debaixo do travesseiro": <https://docs.python.org/3/library/index.html>

### 3. Instalando módulos

O Python possui um grande repositório de módulos desenvolvidos pela comunidade. São módulos que não vem pré-instalados, mas que alguém fez e disponibilizou. Alguns desses módulos são extremamente populares e amplamente utilizados, sendo quase um consenso em suas respectivas áreas, como o matplotlib (geração de gráficos) e o pandas (manipulação de tabelas). Sempre que precisar instalar um novo módulo, basta abrir o console do seu sistema operacional (Prompt de Comando no Windows) e digitar: pip install nome-do-modulo

Por exemplo, para instalar a matplotlib, usamos: pip install matplotlib

Dica: Você pode navegar pelo repositório onde o pip encontra os módulos: <https://pypi.org/>

# Arquivos

Uma característica que nenhum de nossos programas teve agora é a persistência de dados. Sempre que nossos programas eram executados, eles precisavam que os usuários digitassem todos os dados de entrada novamente, e após exibir os dados de saída na tela, o programa era fechado e esses dados eram perdidos para sempre.

A persistência se dá através de arquivos: estruturas abstratas para armazenar dados em uma memória permanente, como o disco rígido, um drive USB ou um servidor web.

## 1. Arquivos em Python

O Python possui algumas funções prontas para manipular arquivos binários puros (onde, conhecendo a estrutura interna de qualquer formato, podemos salvar qualquer tipo de arquivo) e para manipular arquivos de texto (onde os binários são decodificados como strings).

Focaremos no básico de manipulação de arquivo de texto, pois, na prática, quando formos trabalhar com arquivos mais complexos, é provável que usaremos bibliotecas específicas para lidar com eles, e elas já terão funções próprias para ler e salvar esses arquivos da maneira correta.

### 1.1. Abrindo e fechando arquivos

Podemos criar arquivos novos ou abrir arquivos já existentes utilizando a função open. Ela possui 2 argumentos: uma com o caminho do arquivo e outra com o modo de operação.

| Modo | Símbolo | Descrição |
| --- | --- | --- |
| read | r | lê um arquivo existente |
| write | w | cria um novo arquivo |
| append | a | abre um arquivo existente para adicionar informações ao seu final |
| update | + | ao ser combinado com outros modos, permite alteração de arquivo já existente (ex: r+ abre um arquivo existente e permite modificá-lo) |

Após abrirmos (ou criarmos) um arquivo, podemos realizar diversas operações. Ao final de todas elas, devemos fechar o nosso arquivo usando a função close. Essa etapa é importante por 2 motivos:

* 1. Se alteramos o arquivo mas não o fechamos, as alterações não serão salvas.
  2. Se esquecermos de fechar um arquivo, outros programas podem ter problemas de acesso a ele.

A função open retorna alguns dados que devem ser salvos em uma variável que será responsável por guardar alguns dados de acesso ao arquivo, para uso interno do Python.

### 1.2. Escrevendo arquivos

Para entender melhor o open e o close, façamos um programinha que escreve algo em um arquivo. Além das duas funções que já vimos, também utilizaremos a função write, que escreve um texto em um arquivo. É quase como um print mais simples, mas ele aceita apenas uma string.

arquivo = open('ola.txt', 'w') # cria um arquivo ola.txt

arquivo.write('Olá mundo!') # escreve "Olá mundo" no arquivo

arquivo.close() # fecha e salva o arquivo

Após executar o programa acima, abra a pasta onde seu programa está salvo. Note que apareceu um ola.txt lá. Abra-o e verifique seu conteúdo.

### 1.3. Lendo arquivos

Para ler um arquivo existente, não basta usar o open para abri-lo. É necessário carregar seu conteúdo para uma string, de modo que possamos trabalhar com o texto da mesma forma que sempre trabalhamos. A função read faz o oposto da write: ela retorna o texto existente no arquivo.

Copie e rode o programa abaixo na mesma pasta do programa anterior.

arquivo = open('ola.txt', 'r')

conteudo = arquivo.read()

print(conteudo)

arquivo.close()

Note que recuperamos com sucesso na variável conteudo o texto que estava salvo no arquivo!

# Arquivos CSV

Muitos dados interessantes ou importantes estão disponíveis na forma de tabela. A capacidade de manipular planilhas foi determinante no sucesso dos computadores pessoais, dada sua importância para empresas e indivíduos.

Aprenderemos a manipular dados utilizando um dos formatos de planilha mais amplamente utilizados na web: o formato CSV. Mas antes, como podemos representar tabelas em Python?

## 1. Tabelas em Python

Conforme explicado em capítulos anteriores, temos módulos prontos para realizar muitas tarefas para nós. Um dos módulos mais populares em Python é o pandas, que mesmo não vindo instalado por padrão é provavelmente o módulo mais usado para manipular planilhas. Porém, como este é um curso introdutório, convém entendermos um pouquinho de lógica de como manipular uma tabela para futuramente sermos capazes de trabalhar corretamente com os módulos prontos.

Uma das formas mais simples de se representar uma tabela em Python seria através de uma lista de listas. Nossa lista principal seria a tabela como um todo, e cada lista interna seria uma linha da tabela.

Para acessar um elemento individual, utilizamos 2 índices: o primeiro indica a lista interna (linha) e o segundo o elemento individual na lista (coluna). Para percorrer a tabela inteira, utilizamos 2 for aninhados: o mais externo fixa uma linha e o mais interno percorre cada elemento daquela linha.

tabela = [['Aluno', 'Nota 1', 'Nota 2', 'Presenças'],

['Luke', 7, 9, 15],

['Han', 4, 7, 10],

['Leia', 9, 9, 16]]

print('Imprimindo cada elemento individual da tabela:')

for linha in tabela:

for elemento in linha:

print(elemento)

print('Imprimindo cada "linha" da tabela:')

for linha in tabela:

print(linha)

print('Imprimindo o elemento na linha 2, coluna 0:')

print(tabela[2][0])

## 2. O formato CSV

A sigla CSV significa Comma-Separated Values, ou "valores separados por vírgula". Este formato é uma forma padrão de representar tabelas usando arquivos de texto simples: cada elemento é separado por uma vírgula, e cada linha é separada por uma quebra de linha.

Na prática, nem sempre o padrão é seguido à risca: podemos utilizar outros símbolos para fazer a separação. Um bom motivo é o fato de a vírgula ser utilizada para representar casa decimal em algumas línguas, como a língua portuguesa. O importante é ser coerente: todos os elementos deverão ser separados pelo mesmo símbolo, e todas as linhas deverão ter o mesmo número de elementos.

Cole o texto abaixo em um editor de texto puro (como o Bloco de Notas, no Windows) e salve-o com a extensão .csv.

Aluno;Nota 1;Nota 2;Presenças

Luke;7;9;15

Han;4;7;10

Leia;9;9;16

Caso você tenha um editor de planilha instalado, como o Excel, é provável que o ícone representando o arquivo seja o ícone do editor de planilhas, e não de um arquivo de texto. Abra-o com seu editor de planilha e observe como ele interpreta corretamente os dados!

Devido ao fato de ser um formato aberto (ou seja, não é necessário pagar por propriedade intelectual para usar) e ser muito fácil de manipular, diversos programas diferentes possuem a opção de importar ou exportar dados em CSV, e diversas bases de dados na web fornecem a opção de baixar os dados neste formato.

## 3. O módulo CSV em Python

Devido à facilidade de trabalhar com arquivos CSV, com o que vimos sobre arquivos até o momento já conseguimos facilmente escrever um programinha que escreva uma planilha (representada como lista de listas) em um arquivo CSV. Da mesma forma, utilizando as funções que vimos de strings, conseguimos abrir um arquivo CSV e adequadamente reconhecer seus elementos (dica: split).

Porém, como mencionamos antes, o Python possui muita coisa pronta, então não precisamos constantemente reinventar a roda. Existe um módulo chamado csv que já vem instalado com o Python. Ele já faz sozinho o serviço bruto de transformar nossa lista de listas em um texto separado por símbolos e vice-versa.

### 3.1. Escrevendo um CSV

Para escrever um CSV utilizando o módulo, precisamos ter nossos dados representados como uma lista de listas. Criaremos (ou abriremos) um arquivo usando o open, como já fizemos antes, e utilizaremos um CSV writer - uma estrutura que guardará as regrinhas para escrever nosso CSV. Execute o exemplo abaixo:

import csv

tabela = [['Aluno', 'Nota 1', 'Nota 2', 'Presenças'],

['Luke', 7, 9, 15],

['Han', 4, 7, 10],

['Leia', 9, 9, 16]]

# cria o arquivo CSV

arquivo = open('alunos.csv', 'w')

# definindo as regras do nosso CSV:

# ele será escrito no arquivo apontado pela variável 'arquivo'

# seus elementos serão delimitados (delimiter) pelo símbolo ';'

# suas linhas serão encerradas (lineterminator) por uma quebra de linha

escritor = csv.writer(arquivo, delimiter=';', lineterminator='\n')

# escreve uma lista de listas em formato CSV:

escritor.writerows(tabela)

# fecha e salva o arquivo

arquivo.close()

Após executar o programa acima, deve ter surgido um programa alunos.csv na mesma pasta, e seu editor de planilhas provavelmente o reconhece com sucesso. Se você abrí-lo com um editor de texto puro, verá os dados separados por ; igualzinho ao arquivo que criamos manualmente antes.

### 3.2. Lendo um CSV

O processo para ler o CSV é semelhante: utilizamos um CSV reader, com os mesmos parâmetros utilizados no CSV writer. A função csv.writer já retorna para nós uma estrutura iterável (ou seja, que pode ser percorrida com for) contendo cada linha já organizada como lista.

Coloque o exemplo abaixo na mesma pasta do exemplo anterior para abrirmos

import csv

arquivo = open('alunos.csv', 'r')

planilha = csv.reader(arquivo, delimiter=';', lineterminator='\n')

for linha in planilha:

print(linha)

arquivo.close()

## 4. Trabalhando com CSV de forma ainda mais simples: Pandas

Vimos acima como ler e escrever arquivos csv em Python utilizando a biblioteca csv. No entanto, é possível fazer essas duas tarefas de modo muito mais simples e estruturado, utilizando uma outra biblioteca do Python extremamente útil para tratamento de dados: o Pandas.

Teremos uma introdução a esta biblioteca no Módulo 4, e aí aprenderemos como ler, modificar e salvar arquivos csv de uma forma muito simples, além de outras coisas (como, por exemplo, ler e salvar planilhas do Excel!)

# Programação Orientada a Objeto

## 1. Paradigmas de programação

Ao longo do curso, sempre pensamos nossos programas como sequências de comandos. Uma instrução abaixo da outra, na ordem em que devem ser executadas. Quando muito agrupávamos algumas instruções dentro de uma função e fazíamos desvios de fluxo para essas funções. Dentro de cada uma delas, porém, havia a mesma estrutura sequencial de instruções. Chamamos esse modelo de programação imperativa ou programação procedural.

Porém, essa não é a única forma de organizar um programa. Temos diversas outras formas de pensar em um programa. Chamamos essas formas de paradigmas de programação. Uma linguagem de programação pode suportar exclusivamente um paradigma ou vários paradigmas. A linguagem C, por exemplo, é usada apenas para programação procedural. Já o Python pode ser usado para programação procedural, orientada a objeto, funcional, entre outros.

Vejamos alguns exemplos:

* Programação procedural/imperativa: foco em instruções que modificam o estado do programa e modularização em funções.
* Programação orientada a objeto: foco na modelagem das entidades envolvidas em um problema (objetos) e na interação entre elas.
* Programação funcional: foco na declaração e avaliação de funções, que são definidas como árvores de expressões que sempre retornam valores, ao invés de blocos de instruções.
* Programação orientada a evento: foco na resposta a ação dos usuários (por exemplo, um clique do mouse provoca uma reação do programa).

Estudaremos um pouco sobre o paradigma orientado a objeto.

## 2. Programação Orientada a Objeto

O grande objetivo da programação orientada a objeto é a reutilização de código. Os programas devem ser modularizados, de modo que diferentes pessoas possam implementar módulos diferentes e juntá-los ao final. A integração entre os módulos deve ser relativamente simples, e se possível um módulo deve poder ser reaproveitado em outros sistemas.

Isso é feito através da modelagem das entidades que compõem um problema ou interagirão com o sistema. Utilizamos a ideia de classes e objetos para representar as informações que cada entidade possui e os comportamentos que elas apresentam.

### 2.1. Classes

Uma classe é um modelo genérico para criarmos entidades (objetos). Na classe nós especificamos as informações e os comportamentos que os objetos terão. Todos os objetos pertencentes a uma mesma classe terão características em comum.

### 2.2. Objetos

Podemos dizer que as classes são ideias abstratas, enquanto objetos são instâncias concretas da classe. Por exemplo, podemos dizer que "Pessoa" é uma classe. Quando pensamos em "Pessoa", já imaginamos uma figura com cabeça, olhos, nariz, boca, tronco, mãos e pés que normalmente anda sobre 2 pernas e se comunica com a voz. Você, leitor, e eu, escritor, somos objetos da classe "Pessoa". Somos reais, existimos, temos diversas características em comum, mas cada um de nós tem suas particularidades, como diferenças de altura, forma, cor, voz etc.

### 2.3. Métodos

Para modelar o comportamento dos nossos objetos, podemos criar funções dentro da classe. Elas são funções como qualquer outra, mas tipicamente terão pelo menos um parâmetro: uma referência para o objeto que está realizando a ação. Essas funções também não podem ser executadas arbitrariamente, mas deverão ser chamadas necessariamente pelos objetos. Essas "funções do objeto" são chamadas de métodos.

### 2.4. Atributos

Para armazenar os dados dos nossos objetos, podemos criar variáveis internas nos objetos. Essas variáveis são particulares de cada objeto, ou seja, diferentes objetos de uma mesma classe normalmente terão as mesmas variáveis, mas o valor delas não necessariamente será igual. Chamamos essas "variáveis do objeto" de atributos.

Quatro princípios básicos nos ajudam a fazer uma boa modelagem:

* Encapsulamento: cada classe deve conter todas as informações necessárias para seu funcionamento bem como todos os métodos necessários para alterar essas informações. Idealmente, apenas objetos da própria classe poderão alterar seus atributos, e objetos de outras classes irão interagir com eles através de métodos.
* Abstração: as classes devem apresentar interfaces simples para o uso por outros desenvolvedores e para a interação com outras classes. Todos os detalhes complicados de seu funcionamento devem estar "escondidos" dentro de métodos simples de usar, com parâmetros e retornos bem definidos. Um exemplo disso é um smartphone: por dentro ele possui circuitos extremamente complexos e softwares avançadíssimos, mas para o usuário ele apresenta uma interface simples com poucos botões e vários ícones intuitivos na tela.
* Herança: se várias classes terão atributos e métodos em comum, não devemos ter que redigitá-los várias vezes. Ao invés disso, criamos uma classe com esses atributos comuns e as outras classes irão herdá-los.
* Polimorfismo: objetos de diferentes classes herdeiras de uma mesma classe mãe podem ser tratados genericamente como objetos pertencentes à classe mãe.

# Classes e objetos em Python

No capítulo anterior introduzimos conceitos de programação orientada a objeto. Esses conceitos são gerais e podem ser aplicados a qualquer linguagem orientada a objeto. Vejamos como implementar essas funcionalidades em Python.

## 1. Classes

### 1.1. Criando uma classe

Para definir uma classe em Python utilizamos a palavra class seguida de um nome para a classe.

Normalmente nossas classes terão um método especial conhecido como método construtor. Ele sempre será chamado de \_\_init\_\_ em Python. Sempre que criamos um objeto novo o Python automaticamente executa o método construtor. Apesar de não ser obrigatório, é uma boa prática criar os atributos e atribuir um valor inicial para eles no construtor. Como todos os objetos executam o construtor ao serem criados, é um jeito de garantir que todos eles terão os mesmos atributos.

Como comentamos anteriormente, os métodos devem receber uma referência para o objeto que o chamou. Essa é uma forma do método ter acesso aos atributos do objeto. O primeiro parâmetro de um método sempre será interpretado como sendo essa referência, e convenciona-se chamá-lo de self.

Vejamos um exemplo de uma classe que representa jogadores para um jogo de dados. Cada jogador terá 2 atributos: um nome (que será fornecido na construção do objeto) e uma pontuação (que sempre começará em 0). Eles também terão 2 métodos: o construtor e um método que rola um dado.

class Jogador:

# criando o método construtor

def \_\_init\_\_(self, nome):

# cria uma variável (atributo) dentro do objeto sendo criado

# esse atributo recebeu o nome passado como parâmetro

self.nome = nome

# cria outro atributo no objeto sendo criado

# esse atributo não veio como parâmetro, ele sempre começará valendo 0

self.pontuacao = 0

def rolaDado(self):

# sorteia um número de 1 a 6:

dado = random.randrange(1, 7)

# atualiza o atributo "pontuacao" do objeto que chamou o método:

self.pontuacao = self.pontuacao + dado

## 2. Objetos

### 2.1. Instanciando objetos

Ao rodar o exemplo acima, nada acontece. Isso ocorre porque apenas ensinamos o Python como criar jogadores, mas nunca chegamos a criar um. É exatamente o mesmo caso de quando definimos uma função mas não a chamamos.

Criamos objetos da mesma maneira que criamos variáveis: damos um nome a eles e atribuímos um "valor". No caso, o "valor" atribuido será uma "chamada" para o nome da classe, como se ela fosse uma função. Adicione o seguinte trecho de código fora da classe Jogador (ou seja, após a classe e sem tabs):

player1 = Jogador('Mario')

# vejamos o tipo da "variável" (objeto) que criamos:

print(type(player1))

Quando chamamos a classe dessa maneira, o Python irá automaticamente executar o método construtor (\_\_init\_\_). Algo que pode chamar a atenção é que o método possui 2 parâmetros: self e nome. Porém, nós passamos apenas o nome. O parâmetro self é implícito: nós nunca o passamos, ele é apenas um meio de informarmos ao Python qual será o "apelido" do objeto que está chamando o método para usarmos internamente no método para acessar seus atributos.

Note que player1 é do tipo 'class Jogador'. Ou seja, a nossa classe é considerada um tipo novo de dado, e os objetos são variáveis desse tipo novo.

### 2.2. Acessando atributos

Para acessar um atributo, usamos o nome do objeto, um ponto e o nome do atributo. Vamos acessar o nome do nosso jogador e o seu placar atual:

print(player1.nome)

print(player1.pontuacao)

Os atributos se comportam como variáveis normais. Podemos ler seu valor e escrever valores novos:

player1.nome = 'Luigi'

print(player1.nome)

### 2.3. Executando métodos

Para executar o método, iremos chamá-lo à partir do objeto, da mesma maneira que fizemos com os atributos:

player1.rolaDado()

print(player1.pontuacao)

A notação de um método sendo chamado por um objeto parece familiar? É porque já estamos usando essa notação faz um tempo. Lembra do lista.append() ou do string.upper()? Pois bem: list e str são classes, e todas aquelas funções que vimos em capítulos anteriores são métodos de objetos list e str!

# Métodos mágicos

Uma classe pode conter diversos métodos diferentes para realizar múltiplas tarefas. Porém, o programador que for utilizar uma classe com muitos métodos pode ter bastante trabalho para lembrar de todos esses métodos... Pense em todos os métodos de strings que estudamos, por exemplo. Eles nem sequer são todos os métodos existentes, mas já foram muitos! Para tornar nossa classe mais intuitiva de usar (princípio da abstração), podemos reaproveitar alguns recursos padrão do Python que todo programador já sabe utilizar e adaptá-los para nossa classe.

Considere o código abaixo:

string1 = "Let's "

string2 = "Code"

string3 = string1 + string2

print(string3)

Sabemos que o resultado será a string "Let's Code"... Mas pare para pensar um pouco: o operador + é um operador aritmético, ele serve para fazer conta. E aqui estamos usando-o com strings, e mesmo assim ele funciona. Quem desenvolveu a classe str concluiu que seria mais intuitivo para nós usar o sinal + para concatenar strings do que memorizar um nome como "concat". Podemos aplicar o mesmo conceito em nossas classes: redefinir o comportamento de operadores e funções padrão do Python para nossa classe.

Para isso, basta criar métodos em nossa classe que respeitem certas regrinhas, como nomes padrão já pré-determinados pelo Python. Sempre que ele encontrar esses métodos, ele saberá que é para chamá-los quando o operador for usado entre elementos da classe. Esses são os chamados métodos mágicos.

Para este capítulo, consideremos uma classe chamada Horario. Essa classe servirá para representar horários, dados contendo 3 campos: hora (h), minuto (m) e segundo (s).

class Horario:

def \_\_init\_\_(self, \*\*dados):

if 'h' in dados:

self.h = dados['h']

else:

self.h = 0

if 'm' in dados:

self.m = dados['m']

else:

self.m = 0

if 's' in dados:

self.s = dados['s']

else:

self.s = 0

O que seria esse \*\*dados? E esses if ... in ...? Se não se recorda, revise o capítulo de funções com parâmetros opcionais. Estamos dando liberdade a quem usa essa classe para passar valores para os campos h, m ou s, e iremos adotar o valor 0 para os campos omitidos.

## 1. Representação

Você já experimentou dar print em um objeto de uma classe que criou? Experimente criar um objeto da nossa nova classe Horario e imprimi-lo.

Bom, nosso objeto possui 3 atributos e um método. O Python não sabe qual ou quais dessas informações são relevantes para representar um horário, em qual ordem elas deveriam aparecer ou mesmo se possuem algum tipo de formatação especial. Por exemplo, quando imprimimos uma lista, ele não mostra apenas o seu conteúdo: ele também mostra os símbolos [, ] e , para indicar o início e final da lista, bem como a separação entre os elementos. No caso da nossa classe Horario, seria interessante representar os objetos mostrando os 3 campos com 2 dígitos e separados por dois pontos (o famoso hh:mm:ss).

Ao imprimir um objeto, o print procura por um método chamado \_\_repr\_\_. Caso o objeto tenha esse método, o seu retorno é utilizado pelo print.

Vamos modificar nossa classe adicionando o seguinte método:

def \_\_repr\_\_(self):

representacao = '{:02d}:{:02d}:{:02d}'.format(self.h, self.m, self.s)

return representacao

Tente novamente imprimir o objeto e observe o resultado!

Na realidade, existem 2 métodos mágicos ligados à representação: o \_\_str\_\_ é o ideal para fazer uma representação "bonitinha" para printarmos e o usuário ver na tela, enquanto o \_\_repr\_\_ seria para criar uma representação não ambígua de um objeto (ex: 2 objetos Horario distintos que coincidentemente estejam com os mesmos 3 valores nos parâmetros apareceriam iguais no \_\_str\_\_ e diferentes no \_\_repr\_\_). A utilidade da representação não ambígua seria, por exemplo, gerar logs indicando o que cada objeto está fazendo para encontrarmos bugs.

Porém, quando imprimimos uma coleção de objetos, a impressão da coleção (ex: uma lista de objetos Horario) chamará o \_\_repr\_\_, e não o \_\_str\_\_ dos objetos. Além disso, na ausência do \_\_str\_\_, o \_\_repr\_\_ é usado automaticamente em todos os casos.

Para fins didáticos, podemos usar apenas o \_\_repr\_\_ neste curso. Os alunos interessados em um aprofundamento podem ler mais em <https://stackoverflow.com/questions/1436703/difference-between-str-and-repr>

## 2. Operadores aritméticos

Conforme vimos na introdução deste capítulo, podemos ensinar para o Python como tratar uma soma entre dois objetos de uma certa classe. Isso vale para todas as outras operações aritméticas que conhecemos, como subtração, divisão, resto de divisão etc.

Quando nós fazemos em Python uma operação do tipo c = a + b, o Python "traduz" isso da seguinte maneira: c = a.\_\_add\_\_(b) Ou seja, uma operação aritmética é um método chamado pelo objeto à esquerda do operador que recebe o operador à direita como parâmetro!

Vejamos um exemplo útil: sabemos como somar 2 horários: soma-se os segundos primeiro, em seguida os minutos, e por fim as horas. Se os segundos estourarem 60, subtraimos 60 do campo de segundos e somamos 1 no campo de minutos. Idem para o campo de minutos. Podemos "ensinar" essa regra para o Python dentro do \_\_add\_\_, e dali em diante podemos somar 2 horários.

Adicione o seguinte código à sua classe:

def \_\_add\_\_(self, other):

# soma hora com hora, minuto com minuto e segundo com segundo

h = self.h + other.h

m = self.m + other.m

s = self.s + other.s

# corrige os segundos em excesso

if s >= 60:

s = s - 60

m = m + 1

# corrige os minutos em excesso

if m >= 60:

m = m - 60

h = h + 1

# um Horario + outro Horario = terceiro Horario

# vamos criar um objeto Horario usando o h, m e s calculados!

resultado = Horario(h=h, m=m, s=s)

# retornando o resultado para poder fazer operações como "c = a + b"

return resultado

Fora da classe, no programa principal, faça o teste:

duracaoJogo = Horario (h=1, m=30)

duracaoIntervalo = Horario (m=15)

transmissaoTotal = duracaoJogo + duracaoIntervalo

print(transmissaoTotal)

Temos os seguintes métodos mágicos aritméticos:

* \_\_add\_\_ soma: +
* \_\_sub\_\_ subtração: -
* \_\_mul\_\_ multiplicação: \*
* \_\_truediv\_\_ divisão: /
* \_\_floordiv\_\_ divisão inteira: //
* \_\_mod\_\_ resto de divisão: %
* \_\_pow\_\_ potência: \*\*

## 3. Operadores lógicos

Assim como podemos redefinir os operadores aritméticos, podemos redefinir os operadores lógicos através de métodos mágicos. Da mesma maneira que os operadores aritméticos, os operadores lógicos são traduzidos para o Python em termos de uma chamada para um método do objeto à esquerda recebendo como parâmetro o objeto da direita. Por exemplo, um teste de igualdade do tipo a > b torna-se a.\_\_gt\_\_(b). Os métodos de operadores lógicos devem retornar True ou False.

No caso da nossa classe Horario, sabemos que se o campo h de um objeto for maior do que o do outro, ele é necessariamente maior. Se ambos os campos forem iguais, o campo m deverá ser maior. Se ambos h e m forem iguais, o desempate fica no campo s.

Adicione o método mágico abaixo à sua classe Horario para que ela consiga utilizar o operador >:

def \_\_gt\_\_(self, other):

# hora é maior

if self.h > other.h:

return True

# empate na hora, mas minuto é maior

elif self.h == other.h and self.m > other.m:

return True

# empate na hora e no minuto, mas segundo é maior

elif self.h == other.h and self.m == other.m and self.s > other.s:

return True

# em todos os outros casos, o objeto é definitivamente menor

else:

return False

No programa principal (fora da classe), vamos aproveitar os objetos do exemplo anterior e fazer um teste:

if duracaoJogo > duracaoIntervalo:

print('Temos mais tempo de jogo do que de intervalo')

else:

print('Temos mais tempo de intervalo do que de jogo')

Os métodos lógicos são:

* \_\_gt\_\_ maior que (greater than): >
* \_\_ge\_\_ maior ou igual (greater or equal): >=
* \_\_lt\_\_ menor que (less than): <
* \_\_le\_\_ menor ou igual (less or equal): <=
* \_\_eq\_\_ igual (equal): ==
* \_\_ne\_\_ diferente (not equal): !=

Este capítulo abordou os métodos mágicos mais comuns. Temos vários outros, mas além de ser uma lista muito longa, alguns deles são para funcionalidades avançadas não abordadas no curso. Quem estiver interessado em se aprofundar para fazer projetos mais sofisticados pode conferir esse guia excelente: <https://rszalski.github.io/magicmethods/>

Desafio: complemente a nossa classe Horario implementando métodos mágicos para subtração e para todos os operadores lógicos. Experimente também ajustar o construtor para prevenir horários inválidos: por exemplo, se alguém passar m=90, faça m=30 e some +1 em h.

# Métodos e atributos estáticos

Todos os exemplos até agora mostraram atributos e métodos pertencendo aos objetos: a classe definia quais atributos e métodos o objeto deveria ter, mas cada objeto tem um valor distinto para cada atributo e os métodos devem ser chamados a partir dos objetos.

Porém, há casos onde gostaríamos que uma certa informação fosse coletiva. Pense, por exemplo, na população do Brasil: cada pessoa deve saber seu próprio nome e seu próprio CPF. Porém, é competência da administração federal ter dados como o número total de habitantes e registro de todos os CPFs em circulação. Não faria sentido exigir que cada habitante soubesse todas essas informações.

Da mesma forma, realizar o censo para manter essas informações atualizadas não é de competência de cada cidadão individual, mas da administração federal. Ela deverá ter seus mecanismos para sempre ter informações atualizadas sobre os habitantes.

Esses problemas são resolvidos pelos atributos estáticos e métodos estáticos.

## 1. Atributos estáticos

Um atributo estático nada mais é do que um atributo que é coletivo, pertencente à classe como um todo. Não haverá uma cópia diferente dele para cada objeto, mas sim uma única informação que pode ser acessada diretamente na classe, sem sequer necessitar que um objeto já tenha sido instanciado.

Para criar um atributo estático, basta criar uma variável (atribuindo um valor inicial a ela) dentro da classe, mas fora de qualquer um de seus métodos.

class Pessoa:

# criando um atributo estático

populacao = 0

def \_\_init\_\_(self, nome, cpf):

self.nome = nome

self.cpf = cpf

# o \_\_init\_\_ é chamado quando uma Pessoa nova é criada...

# ...portanto, vamos atualizar nossa populacao sempre que alguém for criado!

Pessoa.populacao = Pessoa.populacao + 1

maisVelho = Pessoa('Mario', 12345678901)

maisNovo = Pessoa('Luigi', 98765432109)

# Note que para acessar o atributo estático, não referenciamos um objeto, e sim a classe!

print('População atual: ', Pessoa.populacao)

## 2. Métodos estáticos

Imagine que você precise de algum método que seja bastante útil para a classe, mas que não necessariamente precise de um objeto. Chamamos esses métodos de métodos estáticos.

Para que um método seja estático, podemos utilizar um decorator (de maneira simplificada, um modificador de comportamento da função). Nós iremos escrever @staticmethod acima da definição do método.

Nosso método não terá um self, apenas parâmetros que sejam realmente necessários. E na hora de usá-lo, usaremos o nome da classe ao invés do nome de um objeto, como fizemos com os atributos estáticos.

Adicione o seguinte método à classe Pessoa:

@staticmethod

def popTotal():

print('População total:', Pessoa.populacao)

Note que ele não possui self, pois ele não precisa de acesso a nenhuma informação de qualquer objeto em particular.

Para chamá-lo, inclua a seguinte linha no programa principal:

Pessoa.popTotal()

# Herança e polimorfismo

Imagine que você tenha várias classes com várias características em comum: os mesmos atributos, os mesmos métodos ou métodos parecidos e com o mesmo nome e mesmos parâmetros. Reescrevê-los várias vezes é um desperdício de tempo e pode induzir ao erro. Além disso, se pecisarmos atualizar um método, precisaremos fazer a modificação múltiplas vezes. A herança e o polimorfismo são a solução para esse problema.

## 1. Herança

Para que diversas classes possuam os mesmos atributos e métodos, podemos criar uma classe com esses atributos e métodos e dizer que as outras classes herdarão dessa classe. As classes filhas possuem os mesmos atributos e métodos que a classe mãe. Para herdar, colocamos o nome da classe mãe entre parênteses na frente do nome da classe filha em sua definição.

Se necessário, podemos redefinir um método na classe filha - assim, temos a garantia de que todas as classes filhas terão métodos com o mesmo nome, mas eles podem ter comportamentos diferentes.

Execute o código abaixo:

class Animal:

def \_\_init\_\_(self, nome):

self.nome = nome

def fala(self):

print(self.nome, 'faz barulho.')

class Cachorro(Animal):

def fala(self):

print(self.nome, 'faz au au.')

class Gato(Animal):

def fala(self):

print(self.nome, 'faz miau.')

class Dinossauro(Animal):

pass

# pass é um comando "vazio": ele não faz nada, apenas evita erro por não colocarmos nada dentro da classe

cachorrinho = Cachorro('Bidu')

gatinho = Gato('Mingau')

dino = Dinossauro('Horácio')

cachorrinho.fala()

gatinho.fala()

dino.fala()

Note que as classes Cachorro e Gato não possuem método construtor, apenas um método "fala", mas mesmo assim conseguimos criar objetos com um atributo "nome". O Python procurou o método construtor nessas classes, e ao não encontrar, procurou em sua classe mãe (Animal) e o usou. Ao chamarmos o método "fala", o Python usou o das próprias classes filhas, já que elas a possuiam.

No caso da classe Dinossauro, fomos ainda mais radicais: não colocamos nada dentro da classe. Mas não só conseguimos criar um objeto com o atributo "nome", como conseguimos usar o método "fala". Assim como no caso do construtor, foi usado o método da classe mãe.

### 1.1. Herdando parcialmente um método

Imagine que você queira que um método de uma classe filha seja muito parecido com um método de uma classe mãe, mas possua algumas linhas a mais de código. Novamente caímos no problema da reutilização de código: se copiarmos e colarmos o método na classe filha e acrescentarmos algumas linhas, caso tenhamos que atualizar ou corrigir o método na classe mãe no futuro, precisaríamos nos lembrar de atualizá-lo também na classe filha.

Podemos utilizar o método super. Ele nos dá acesso dentro de uma classe filha aos métodos originais da classe mãe. Assim, podemos redefinir na classe filha um método, digitar apenas as linhas de código novas/diferentes e em seguida usar o super para chamar o método original.

Atualize a sua classe Cachorro para que ela agora também crie um atributo "raca", mas sem precisarmos novamente criar o atributo "nome":

class Cachorro(Animal):

# redefinindo o \_\_init\_\_

def \_\_init\_\_(self, nome, raca):

# pedaço novo de código

self.raca = raca

# chamando a \_\_init\_\_ da classe mãe para fazer o resto do trabalho

super().\_\_init\_\_(nome)

def fala(self):

print(self.nome, 'faz au au.')

## 2. Polimorfismo

A palavra "polimorfismo" vem do grego antigo e significa "várias formas". A ideia do polimorfismo em programação é que um objeto de uma certa classe pode se comportar como objeto de outras classes. Mais especificamente, objetos de uma classe filha podem também ser tratados como se pertencessem à classe mãe.

O método isinstance recebe 2 parâmetros: um objeto e uma classe. Ele retorna True caso o objeto pertenca à classe, e False caso não pertença. Cole os testes no programa abaixo para verificar que um cachorro não é um gato, um gato não é cachorro, cachorro é cachorro, gato é gato e ambos são animais:

# o cachorrinho é cachorro?

print(isinstance(cachorrinho, Cachorro))

# o gatinho é gato?

print(isinstance(gatinho, Gato))

# o cachorrinho é gato?

print(isinstance(cachorrinho, Gato))

# o gatinho é cachorro?

print(isinstance(gatinho, Cachorro))

# o cachorrinho é animal?

print(isinstance(cachorrinho, Animal))

# o gatinho é animal?

print(isinstance(gatinho, Animal))

Isso é útil porque uma função que seja feita para lidar com Animal será capaz de lidar com qualquer classe herdeira de Animal com a mesma facilidade. Modifique a classe Animal para registrar todos os animais criados e ter uma função que faz todos eles falarem:

class Animal:

# lista estática de animais criados

animais = []

def \_\_init\_\_(self, nome):

self.nome = nome

# criamos o animal e o colocaremos na lista estática

Animal.animais.append(self)

def fala(self):

print(self.nome, 'faz barulho.')

@staticmethod

def falatorio():

for a in Animal.animais:

a.fala()

Agora façamos um teste no programa principal:

Animal.falatorio()

A classe Animal nem sequer sabe quem herda dela. Ela não conhece Cachorro, Gato ou Dinossauro. Mas ela sabe que todo animal possui um atributo nome e um método fala, e por isso não importa a qual subclasse eles pertencem: todos também pertencem à classe Animal.

# Análise de Complexidade

Quando falamos sobre a complexidade de um algoritmo estamos nos referindo à quantidade de recursos (tempo de processamento ou espaço de armazenamento) que precisa ser consumida ou alocada para sua execução.

Essa análise de complexidade é fundamental para que possamos projetar algoritmos eficientes e verificar a eficiência dos algoritmos que utilizamos. Além disso, para escolher a estrutura de dados mais adequada para cada problema que precisamos resolver, é necessário identificar quais operações essa estrutura fornece com máxima eficiência.

## 1. Como analisar?

Definimos T(n), uma função que representa o tempo de execução em função do tamanho da entrada n. O tamanho da entrada aqui é a quantidade de "coisas" que o algoritmo precisa processar. Fazemos a análise em função do tamanho da entrada pois o que nos interessa é identificar como o algoritmo irá escalar. Em outras palavras queremos responder algo do tipo: "Quando dobramos a entrada, o tempo de execução dobra? quadruplica?". Enfim, o que acontece com o tempo de execução a medida que o tamanho da entrada aumenta.

Além disso, queremos uma medida que seja geral o suficiente para capturar o comportamento do algoritmo independentemente do tipo de máquina que o código irá executar. Se usássemos segundos ou minutos para medir o tempo, teríamos que fazer uma análise diferente para cada processador existente. Por esse motivo, o que contamos é a quantidade de operações básicas que são executadas e cada uma destas instruções leva um tempo constante.

Vejamos, por exemplo, a análise do seguinte trecho de código:

a = 0 ... 1 atribuição

for i in range(n): ... executa n vezes o que estiver dentro do for

a = a + i ...... atribuição e soma são executadas n vezes

Logo, T(n) = 1 + noperações.

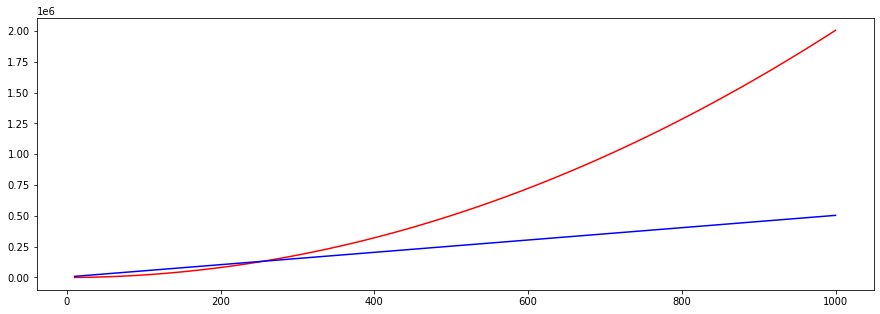
## 2. Comportamento assintótico

Exercício:  
Suponha que você tem algumas listas que precisam ser ordenadas e duas opções de algoritmos com as seguintes complexidades:

* Algoritmo 1: 2n² + 5n operações
* Algoritmo 2: 500n + 4000 operações

Qual deles é mais eficiente para ordenar uma lista com 10 itens? e com 1000?  
Qual você escolheria para utilizar em uma lista arbitrária (da qual você não saberia o tamanho previamente)?

Vejamos abaixo um gráfico do tempo de exeução desses dois algoritmos, Algoritmo 1 em vermelho e Algoritmo 2 em azul:



Pelo gráfico podemos ver que, embora Algoritmo 1 seja mais eficiente que Algoritmo 2 para "valores pequenos" de n, ele rapidamente se torna muito pior a medida que n aumenta. O termo quadrático de Algoritmo 1 (2n²) "domina" a função, ou seja, a contribuição desse termo é maior do que todos os outros de maneira que, para valores grandes de n, a contribuição dos demais termos se torna irrelevante para o comportamento da função.

Dito isso, vemos que não há necessidade de identificar a função T(n) completamente precisa levando em conta cada instrução, o que realmente importa é o comportamento assintótico de T(n), ou seja o que ocorre com T(n) quando n aumenta consideravelmente. Em outras palavras, queremos a ordem da função T(n) de cada algoritmo, o termo dominante da função. Nesse caso, dizemos que Algoritmo 1 é O(n\*\*2) (lê-se "da ordem de n quadrado", ou "o de n quadrado") e Algoritmo 2 é O(n). E é assim que analisaremos e representaremos a complexidade dos algoritmos daqui para frente.

No exemplo da seção anterior: T(n) = 1 + n = O(n). De modo geral, ao fazer uma análise podemos focar nos loops do código.

Mas e quando o loop muda de tamanho?

a = 0

for i in range(n):

for j in range(i + 1, n):

a = a + i

a = a + j

print(a)

Nesse caso vemos que o loop mais interno irá executar n-1 vezes na primeira iteração do loop externo, n-2 vezes na segunda iteração, e assim por diante:

(n-1) + (n-2) + (n-3) +...+ 1

Trata-se da soma de todos os termos de uma [Progressão Aritmética](https://pt.wikipedia.org/wiki/Progressão_aritmética), e portanto é da ordem de n\*\*2, O(n\*\*2)

Ou seja, o for interno irá executar n - 1 vezes para cada n do for externo. O que leva a uma função de complexidade T(n) = n(n - 1).

Podemos simplificar em T(n) = n² - n, o que dá uma complexidade assintótica de O(n²).

E qual a complexidade desse código?

i = n

while i > 1:

i = i / 2

print(i)

A princípio podemos pensar que esse loop executaria n vezes, porém se olharmos com mais atenção o termo i está sendo cortado pela metade a cada iteração. Ou seja, esse loop será executado a quantidade de vezes que for possível dividir n por 2 e obter um resultado maior do que 1. Por exemplo, se n = 8 o loop executa 3 vezes, se n = 32 o loop executa 5 vezes. Para um n qualquer, o loop executa O(log2(n)).

Complexidades mais comuns em ordem de eficiência (do pior para o melhor):

- O(n!) -> fatorial

- O(2\*\*n) -> exponencial

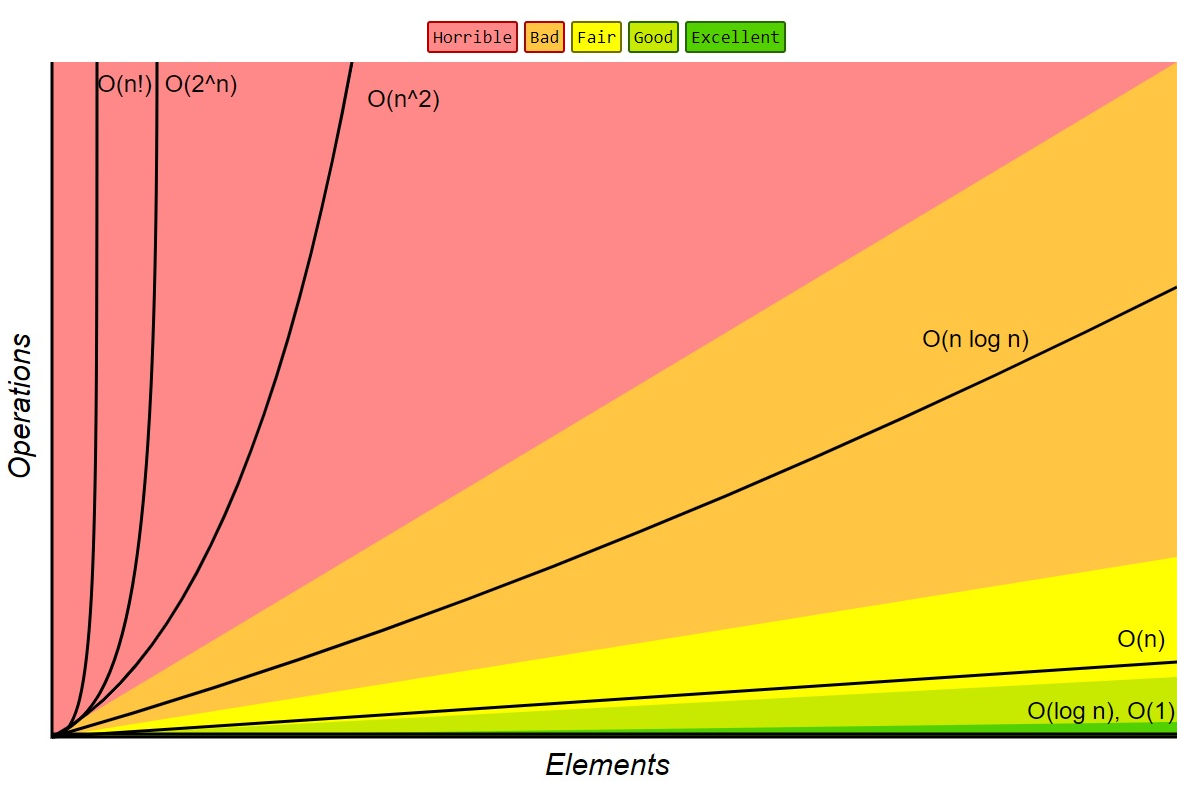
- O(n\*\*2) -> polinomial/quadrático

- O(n\*log2(n)) -> logaritmica

- O(n) -> linear

- O(log2(n)) -> logaritmica

- O(1) -> constante



## 3. Exemplo prático: busca binária

Agora que sabemos analisar algoritmos, vejamos um exemplo mais prático.  
Dada uma lista ordenada, defina uma função def find(lista, item) que retorna True caso item esteja na lista e False caso contrário.

A resolução mais direta desse exercício seria, talvez, iterar por cada elemento da lista e comparar com o item buscado, se for igual retorna verdadeiro, se chegar ao fim da lista retorna falso.  
Implemente e analise a complexidade desse algoritmo. O que ocorre se buscamos um elemento que não está na lista e a lista for muito grande?

Será que podemos fazer algo melhor?  
O algoritmo descrito acima não faz uso da informação de que a lista está ordenada, podemos usar essa ordenação a nosso favor.  
O código abaixo é uma implementação do algoritmo conhecido como busca binária: Começamos comparando o item com o elemento no meio da lista, se o item que queremos for menor que o meio procuramos na metade esquerda da lista, se for maior procuramos na metade direita.

(Lembra da aula de recursão? Qual o caso base desse código recursivo? Será que é mais fácil fazer a busca binária iterativamente ou recursivamente?)

def binary\_search(l, item):

if len(l) == 0:

return False

meio = len(l)//2

if l[meio] == item:

return True

if item > l[meio]:

return binary\_search(l[meio+1:], item)

else:

return binary\_search(l[:meio], item)

Qual é a complexidade desse código? Os exemplos da seção anterior podem ajudar na análise. Meça também o tempo de execução dos dois códigos, o que ocorre em cada um quando dobramos o tamanho da lista?

# Complexidade de estruturas de dados

Tratar da complexidade de algoritmos também possibilita analisar a complexidade das estruturas de dados. Com isso, cada estrutura de dados (listas, dicionários, árvores) possui sua própria complexidade de memória e tempo de execução.

Para analisar essas estruturas podemos nos apoiar nas seguintes operações: acesso, busca, inserção e remoção. De maneira simplificada, podemos anotar os custos médios de complexidade de listas, dicionários, árvores e grafos com a seguinte tabela.

| Estrutura de dados | Acesso | Busca | Inserção | Remoção |
| --- | --- | --- | --- | --- |
| Listas | O(1) | O(N) | O(N) | O(N) |
| Dicionários | O(1) | O(1) | O(1) | O(1) |
| Árvores | O(log2(N)) | O(log2(N)) | O(log2(N)) | O(log2(N)) |