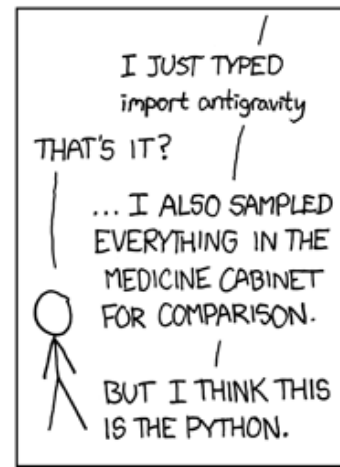
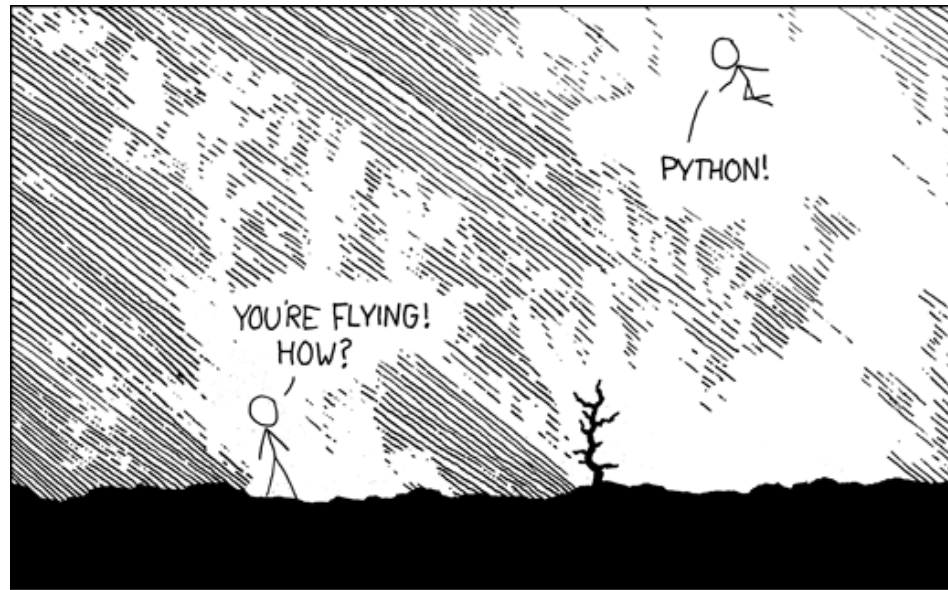




PYTHON BÁSICO
alkeller@unisinos.br

A linguagem Python



PYTHON BÁSICO

Prof. Armando Leopoldo Keller

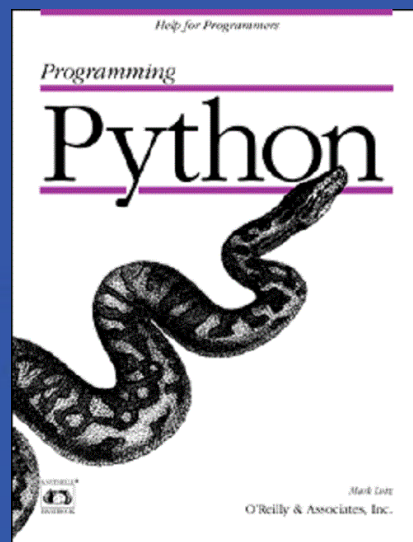
“By the way, the language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!”

(python.org)





MONTY PYTHON'S SPAMALOT™



Algumas empresas que usam Python



Language Rank	Types	Spectrum Ranking
1. Python	🌐 🖥️	100.0
2. C	📱 🖥️	99.7
3. Java	🌐 📱 🖥️	99.5
4. C++	📱 🖥️	97.1
5. C#	🌐 📱 🖥️	97.7
6. R	🖥️	87.7
7. JavaScript	🌐 📱	85.6
8. PHP	🌐	81.2
9. Go	🌐 🖥️	75.1
10. Swift	📱 🖥️	73.7

2017

Language Rank	Types	Custom Ranking
1. C	📱 🖥️	100.0
2. Java	🌐 📱 🖥️	98.1
3. Python	🌐 🖥️	98.0
4. C++	📱 🖥️	97.9
5. R	🖥️	87.7
6. C#	🌐 📱 🖥️	86.5
7. PHP	🌐	82.5
8. JavaScript	🌐 📱	81.8
9. Ruby	🌐 🖥️	74.2
10. Go	🌐 🖥️	71.5
























2016

Language Rank	Types	Custom Ranking
1. Java	🌐 📱 🖥️	100.0
2. C	📱 🖥️	99.3
3. C++	📱 🖥️	95.5
4. Python	🌐 🖥️	93.5
5. C#	🌐 📱 🖥️	92.3
6. JavaScript	🌐 📱	85.1
7. PHP	🌐	84.8
8. Ruby	🌐 🖥️	78.9
9. R	🖥️	74.9
10. Matlab	🖥️	73.9

2014

Language Rank	Types	Custom Ranking
1. Java	🌐 📱 🖥️	100.0
2. C	📱 🖥️	99.7
3. C++	📱 🖥️	99.1
4. Python	🌐 🖥️	96.5
5. C#	🌐 📱 🖥️	91.2
6. PHP	🌐	84.2
7. R	🖥️	83.6
8. JavaScript	🌐 📱	82.7
9. Ruby	🌐 🖥️	75.4
10. Matlab	🖥️	74.0

2015

Language Rank	Types	Spectrum Ranking
1. Python	  	100.0
2. C++	  	99.7
3. Java	  	97.5
4. C	  	96.7
5. C#	  	89.4
6. PHP		84.9
7. R		82.9
8. JavaScript	 	82.6
9. Go	 	76.4
10. Assembly		74.1
11. Matlab		72.8

2018

Motivos da popularidade do Python

PEP 20 - The Zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.

Motivos da popularidade do Python

PEP 20 - The Zen of Python

- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

Como instalar o Python

- Linux e Mac OS (instalado nativo)
- Windows:
 - www.python.org/downloads (Lembrar de deixar habilitado a opção para alterar as variáveis de ambiente)
 - <http://winpython.github.io> (Versão portátil)

Operadores aritméticos

Operador	Nome	Aplicação	Exemplo	Resposta
+	Adição	$x + y$	5+4	10
-	Subtração	$x - y$	10-4	6
*	Multiplicação	$x * y$	2*5	10
/	Divisão	x / y	10/3	3,333333
%	Módulo	$x \% y$	10%3	1
**	Exponenciação	$x ** y$	3**2	9
//	Divisão arredondada para baixo	$x // y$	10//3	3

Obs: Números complexos também são aceitos.

Exemplo: $(10+3j)*5$

Operadores aritméticos

Ordem de execução PEMDAS:

Parênteses

Exponenciação

Multiplicação

Divisão

Adição

Subtração

Mais operações matemáticas

Bibliotecas

- math: <https://docs.python.org/3/library/math.html>
- cmath: <https://docs.python.org/3/library/cmath.html>
- numpy: <https://www.numpy.org/>

Operadores de comparação

As tomadas de decisões são baseadas em comparações, onde dois valores são comparados e como resultado temos uma resposta booleana (Verdadeiro ou Falso).

Operadores de comparação

Por exemplo, na escolha entre dois produtos semelhantes onde o critério de escolha será o menor preço, algumas comparações que podem ser realizadas são:

- O valor do produto A é menor do que o valor do produto B?
- O valor do produto B é maior do que o valor do produto A?

Operadores de comparação

Para utilizar as comparações em python vamos utilizar os seguintes operadores:

Operador	Nome	Aplicação	Exemplo	Resposta
>	Maior que	$x > y$	$5 > 4$	True
<	Menor que	$x < y$	$5 < 4$	False
==	Igual a	$x == y$	$10 == 15$	False
!=	Não é igual a (diferente de)	$x != y$	$10 != 15$	True
>=	Maior ou igual a	$x >= y$	$10 >= 15$	False
<=	Menor ou igual a	$x <= y$	$10 <= 15$	True

Operadores de comparação

Para testar os operadores, abra a IDE Spyder e insira o seguinte código:

```
x = int(input("Entre com um valor numérico para x: "))  
y = int(input("Entre com um valor numérico para y: "))
```

```
print("x > y", x > y)  
print("x < y", x < y)  
print("x == y", x == y)  
print("x != y", x != y)  
print("x >= y", x >= y)  
print("x <= y", x <= y)
```

Operadores lógicos

As comparações ainda podem ser combinadas para formar uma lógica a ser seguida pelo programa, para realizar estas combinações temos os operadores lógicos.

Operador	Nome	Aplicação	Resposta
and	e	x and y	True se x e y forem verdadeiros
or	OU	x or y	True se qualquer um dos operadores for verdadeiro
not	não	not x	False se x for verdadeiro, ou True se x for falso

Operadores lógicos

Exemplo: Para verificar se um número está dentro de uma faixa de valores.

```
x = int(input("Entre com um valor numérico para x: "))
```

```
print(x, " é um valor entre 1 e 10? ", (x>=1)and(x<=10))
```

Testes condicionais

As respostas destas comparações podem ser utilizadas para alterar o fluxo de um programa, para isto serão utilizados os comandos de testes condicionais:

- **if** condição: Executa o bloco de código se a condição for verdadeira;
- **else** : Executa o bloco de código caso a condição do if for falsa;
- **elif** condição: Executa o bloco de código caso a condição do if for falsa e a do elif for verdadeira.

Testes condicionais

Os blocos de código em python são separados por indentação;

A indentação além de separar os blocos de códigos ainda fornece uma melhor leitura do código;

Os blocos são identados por 4 espaços ou 1 tab, alguns editores convertem o tab em 4 espaços.

Testes condicionais

Exemplo:

x = 2

y = 5

if x>y:

Indentação



print("X é maior que Y")

elif x==y:

print("X não é maior que Y, mas sim igual a Y")

else:

print("X não é nem maior, e nem igual a Y")

Laços de repetição

Muitos algoritmos exigem que um determinado bloco de código seja repetido até que uma determinada condição seja satisfeita, ou que uma certa quantidade de iterações seja realizada.

Para isto temos algumas opções de laços de repetição:

- **for**
- **while**

Laços de repetição

O comando **for** normalmente é executado quando se deseja realizar uma certa quantidade de iterações, por exemplo imprimir os números de 0 a 10 na tela.

Sintaxe:

for condição:

bloco de código

Laços de repetição

Exemplo:

```
for x in range(0,11):  
    print(x)
```



Sintaxe do comando range:
range(início, parada, passo)

Laços de repetição

O comando **while** executará o bloco de código enquanto a sua condição for verdadeira;

É importante definir um critério de parada, ou o código pode ficar trancado dentro de um while.

Laços de repetição

Sintaxe:

while condição:
 bloco de código



Laços de repetição

Exemplo:

```
x=0
while x<10:
    print(x)
    x = x+1
```

Laços aninhados

Os laços podem ser utilizados de forma aninhada, ou seja, um laço dentro do bloco de código de outro laço.

Exemplo:

```
for x in range(a):  
    print(x)  
    for y range(b):  
        print(x,y)
```



Funções

Funções

Para que um determinado bloco de código possa ser reutilizado diversas vezes dentro de um programa podemos organizar estes blocos em funções;

Algumas funções nativas do python que já foram utilizadas:

- `print()`
- `int()`
- `input()`

Funções

As funções são declaradas utilizando o comando **def**.

Exemplo:

```
def ola():  
    print("Olá eu sou uma função")
```

Chamando a função:

```
ola()
```

Funções

As funções podem receber argumentos que serão utilizados dentro do seus blocos de códigos.

Estes são passados dentro dos parênteses, separados por virgulas.

Funções

Exemplo:

```
def ola(nome):  
    print("Olá ", nome, "!")
```

```
ola("pessoa1")
```

```
ola("pessoa2")
```

Funções

Os argumentos seguem a ordem em que foram declarados na definição da função, mas podem ser nomeados e receber um valor padrão, o qual será utilizado caso não for passado nenhum parâmetro na chamada da função.

Funções

Exemplo:

```
def ola(nome, tratamento="Sr. "):  
    print("Olá ", tratamento, nome, "!")
```

```
ola("pessoa1")  
ola("pessoa2", "Sra.")  
ola(tratamento="Dr.", nome="Pessoa 3")
```

Funções

Outro recurso importante das funções é o retorno de valores, que é dado pelo comando **return**. Por exemplo uma função que soma dois valores:

```
def soma(a, b):  
    return a+b
```

```
print(soma(5, 3))
```

Funções

Uma mesma função pode ser chamada dentro do seu próprio bloco de código, o que é chamado de recursão.

Exemplo:

```
def fatorial(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        return n*fatorial(n-1)
```

Funções

As variáveis declaradas dentro de um bloco de código de uma função, possuem seu próprio contexto, e não alteram variáveis externas com o mesmo nome.

No entanto é altamente recomendável nomear as variáveis com nomes diferentes e que sejam facilmente entendidos.

Funções

Exemplo:

```
x=10
def quadrado(n):
    x=n**2;
    print(n, x)
quadrado(1)
quadrado(2)
print(x)
```

Funções

Para documentar uma função, utiliza-se um recurso chamado docstrings, que é um comentário de múltiplas linhas (aspas triplas) logo após os dois pontos e antes do bloco de código.

Esta documentação é apresentada como dica em alguns ambientes de desenvolvimento, ou quando o comando help é utilizado.

Tipos de dados

Inteiros

Os números inteiros em python pertencem a classe **int**

Exemplo:

X=10

```
print(type(x))
```

Números não inteiros

Os números não inteiros em python pertencem a classe **float**

Exemplo:

```
X=10.0
```

```
print(type(x))
```

Números não inteiros

Alguns métodos dos floats:

- `float.as_integer_ratio()`: retorna um par de inteiros onde a razão entre eles resulta no float original.

Exemplo:

`(3.5).as_integer_ratio()` # saída (7, 2)

Números não inteiros

- `float.is_integer()`: Retorna True se o valor for inteiro e False se o valor não for inteiro.

Exemplo:

```
(10.1).is_integer() # False
```

```
(10.0).is_integer() # True
```

Números complexos

Os números complexos pertencem a classe **complex**

Exemplo:

```
X = 10+5j
```

```
Y = complex(6, 8)
```

```
type(X)
```

```
type(Y)
```


Números complexos

Funções e atributos dos números complexos:

- `complex.conjugate()`: Retorna o conjugado de um número complexo
- `complex.real` : Retorna o valor da parte real de um número complexo
- `complex.imag` : Retorna o valor da parte imaginária de um número complexo

Listas

Listas

As listas são vetores de dados, onde cada dado assume uma posição dentro do vetor e pode ser acessado utilizando a sua posição.

Para a criação de uma lista, pode-se inicializar a variável com o comando `list()` ou com colchetes `[]`:

```
X= list()
```

```
Y = []
```

Listas

Exemplos de listas e operações, considere as listas X e Y para os exemplos:

X= [0, 1, 2]

Y = [3, 4, 5]

Listas

Para obter a quantidade de itens de uma lista, utiliza-se o comando **len(lista)**.

Exemplo:

```
len(X) # Resposta: 3
```

Listas

Para obter o menor valor presente em uma lista, utiliza-se o comando **min(lista)**.

Exemplo:

`min(X)` # Resposta: 0

`min(Y)` # Resposta: 3

Listas

Para obter o maior valor presente em uma lista, utiliza-se o comando **max(lista)**.

Exemplo:

`max(X)` # Resposta: 2

`max(Y)` # Resposta: 5

Listas

Para obter a soma de todos os valores presentes em uma lista, utiliza-se o comando **sum(lista)**.

Exemplo:

sum(X) # Resposta: 3

sum(Y) # Resposta: 12

Listas

Para verificar se um determinado valor está presente em uma lista, pode-se utilizar o operador **in**.

Exemplo:

2 in X # Resposta: True

6 in X # Resposta: False

O mesmo é válido para verificar se ele não está presente, utilizando o operador **not in**

Listas

Para concatenar duas listas, basta utilizar o operador **+**:

Exemplo:

`Z = X+Y`

`print(Z)` # Resposta [0, 1, 2, 3, 4, 5]

Listas

Para replicar a lista n vezes utiliza-se o operador $*$, esta função é bastante útil para inicializar listas com um determinado tamanho e valor inicial igual para todas as posições.

Exemplo:

```
K = [0]*5
```

```
print(K) # Resposta [0, 0, 0, 0, 0]
```

Listas

Para acessar o elemento que está em uma posição com índice n na lista utiliza-se os colchetes após o nome da variável, com o índice desejado. Em python este índice inicia em 0

Exemplo:

`X[2]` # retorno 2

`Y[0]` # retorno 3

Listas

O índice também funciona com valores negativos, onde o índice -1 é o último valor da lista, o -2 o penúltimo e assim por diante.

Exemplo:

`X[-1]` # retorno 2

`X[-2]` # retorno 1

Listas

Os colchetes também podem ser utilizados para acessar uma faixa de valores, para receber uma nova lista com os itens contidos nos índices de i a j, utiliza-se o comando

`variavel[i : j+1]`

Exemplo:

```
Z = X+Y      # [ 0, 1, 2, 3, 4, 5]  
print( Z[1:3] ) # [1,2]
```

Listas

Funções das listas:

- `list.append(valor)` : Adiciona o valor ao final de uma lista.

Exemplo: `Y.append(6)` # Agora `Y=[3, 4, 5, 6]`

- `list.index(valor)` : Retorna o índice onde o valor se encontra na lista. Caso o valor não esteja na lista, retorna um erro.

Listas

Funções das listas:

- `list.insert(índice,valor)` : Adiciona o valor na posição indicada pelo índice, deslocando os valores posteriores para a direita.

Exemplo: `x.insert(2, 9)` # Agora `x=[0, 1, 9, 2]`

Listas

Funções das listas:

- `list.remove(valor)` : Remove da lista a primeira ocorrência do valor, e retorna um erro caso o valor não esteja presente na lista.

Exemplo: `x.remove(1)` # Agora `x=[0, 2]`

Listas

Funções das listas:

- `list.reverse()` : Inverte a ordem dos elementos de uma lista

Exemplo: `x.reverse()` # Agora `x=[2,1,0]`

Listas

Funções das listas:

- `list.sort()` : Ordena os valores de uma lista, podendo receber o parâmetro `reverse=True` para ordenar no sentido decrescente.

Exemplo:

`[9,8,10].sort()` # [8, 9, 10]

`[9,8,10].sort(reverse=True)` # [10, 9, 8]

Listas

Os elementos dentro das listas não precisam ser do mesmo tipo, inclusive é possível ter listas dentro de listas.

```
X = ["Olá", 1, 4.5, [0,1,2]]
```

Listas

Utilizando o for para percorrer todos os elementos de uma lista:

```
X = ["Olá", 1, 4.5, [0,1,2]]
```

```
for elemento in X:
```

```
    print(elemento)
```

Listas

As listas podem ter n dimensões, uma vez que uma lista pode conter outras listas.

Exemplo:

```
X = [ [0, 1], [2, 3] ]
```

```
print(x[0][1])
```

Tuplas

Tuplas

As tuplas são um caso especial das listas, elas são listas imutáveis, ou seja, não aceitam alterações;

Para a criação de uma tupla, pode-se inicializar a variável com o comando `tuple()` ou com parênteses `()`:

```
X= tuple(10, 5)
```

```
Y = (10, 5)
```


Tuplas

Como as tuplas são imutáveis, elas possuem menos funções, sendo estas:

- index
- count

Estas funções são as mesmas das listas, e o acesso aos elementos também é realizado da mesma maneira.

Strings (Textos)

Strings

Em python os textos são chamados de “strings”, muitas vezes abreviado como str.

Tudo que estiver entre aspas simples ou duplas será interpretado como uma string.

```
type('isto é uma string')
```

```
type("Isto também")
```

Strings

Caso a string possua mais de uma linha, pode-se utilizar três aspas para iniciar e terminar a string. Exatamente como no comentário de múltiplas linhas, a interpretação dependerá da localização do texto.

Strings

Uma string é um vetor de caracteres, logo seus caracteres podem ser acessados exatamente como um vetor.

```
texto="Olá Mundo"  
print(texto[0:3])
```

Strings

Para saber a quantidade de caracteres em uma string, utiliza-se a função **len()** assim como nos vetores.

```
texto = "Olá Mundo"  
print(len(texto))
```

Strings

Para contar a quantidade de ocorrências de um determinado valor em uma string, utiliza-se a função **count()**

```
texto = "Olá Mundo"  
print( texto.count(u) )
```

Strings

É possível verificar se uma determinada string possui um sufixo específico utilizando a função `endswith(sufixo)`

```
texto = "Olá Mundo"
```

```
print( texto.endswith("ndo") )
```


Strings

Para buscar a posição de um caractere ou sequência de caracteres (substring) em uma string, pode-se utilizar a função **find()** que recebe como argumento a substring a ser buscada.

```
texto = "Olá Mundo"  
print( texto.find("Mundo") )
```

Strings

A formatação de uma string pode ser feita com a função **format()** que recebe como argumento os valores a serem inseridos nos espaços marcados com chaves na string.

```
texto = "Teste de {} de {}".format("formatação","string")  
print(texto)
```

Strings

A conversão de uma string para letras minúsculas pode ser realizada através da função **lower()**

```
teste = "Olá Mundo"  
print( teste.lower() )
```

Strings

Para substituir uma substring por outra substring, pode-se utilizar a função **replace(antigo,novo)** onde os argumentos são a substring que será removida, e a substring que será inserida no lugar.

```
teste = "Olá Mundo"  
teste = teste.replace("Mundo","pessoas")  
print(teste)
```

Strings

Muitas vezes os textos que são importados de arquivos estão separados por algum separador, como espaço, virgula, ponto e virgula. Para criar um vetor com estes valores separados, pode-se utilizar o comando **split(separador)** onde o argumento separador é o tipo de separador utilizado.

```
teste = "5,2,3,4,6"  
print(teste.split(","))
```

Strings

Para verificar se um texto inicia com um determinado prefixo, utiliza-se o comando **startswith(prefixo)**, onde o argumento é o prefixo a ser buscado e a resposta será um valor booleano.

```
teste = "www.google.com"  
print(teste.startswith("www"))
```

Strings

Os textos podem vir com espaços em branco no início ou no final, para remove-los pode-se utilizar o comando `strip()`

```
teste = "  texto  "  
print(teste.strip())
```

Strings

Para colocar o texto todo em letras maiúsculas, utiliza-se o comando **upper()**

```
teste = "Olá Mundo"  
print(teste.upper())
```


Sets (conjuntos)

Sets

Em python um set é uma coleção não ordenada de valores onde estes não se repetem. Para inicializar um set, pode-se utilizar o comando **set()**, ou utilizando chaves **{}** com os valores separados por virgulas.

```
teste = set()
```

```
teste2 = {1,2,3}
```

Sets

Uma grande função do set é separar os valores que estão presentes em uma lista ou string sem estes se repitam.

Por exemplo, para saber quais são as letras que compõem a palavra “abacate”:

```
letras = set("abacate")  
print(letras)
```

Sets

Para verificar se dois sets não compartilham os mesmos elementos, podemos utilizar o comando **isdisjoint(set)**. Caso não exista nenhum elemento em comum, retornará verdadeiro.

```
x = {1,2,3}
```

```
y = {4,5,6}
```

```
z = {2,3,4}
```

```
x.isdisjoint(y)
```

```
y.isdisjoint(z)
```

Sets

Para adicionar mais elementos a um conjunto, pode-se utilizar a função **add()** que recebe como argumento o novo elemento a ser inserido. Ou a função **update()** que recebe uma lista ou conjunto como argumento.

```
x=set()
x.add(3)
x.update([1,2,5,6])
print(x)
```

Sets

A remoção de itens pode ser feita com o comando **discard()** que recebe o elemento a ser removido, ou com o comando **clear()** que removerá todos os elementos do conjunto.

```
x={1,2,3,4}
x.discard(3)
print(x)
x.clear()
print(x)
```

Operações com conjuntos

Operações com conjuntos

Para os exemplos de operações com conjuntos serão considerados os seguintes conjuntos:

$$A = \{1, 2, 3, 4, 5\}$$

$$B = \{4, 5, 6, 7, 8\}$$

Operações com conjuntos

A união de conjuntos pode ser realizada com o operador `|` ou através da função **`union()`**.

```
C = A | B
```

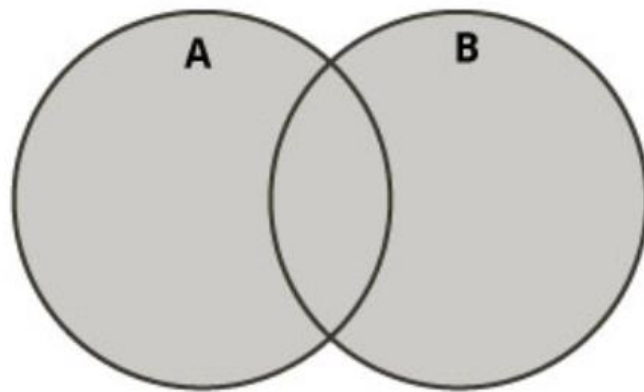
```
print(C)    # {1, 2, 3, 4, 5, 6, 7, 8}
```

```
#método alternativo:
```

```
C = A.union(B)
```

```
# ou
```

```
C = B.union(A)
```



Operações com conjuntos

A interseção de conjuntos pode ser realizada com o operador **&** ou através da função **intersection()**.

```
C = A & B
```

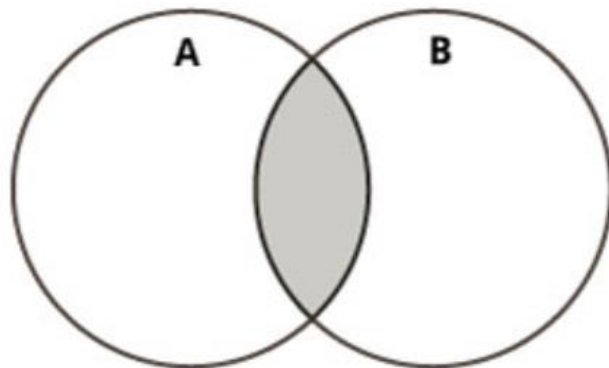
```
print(C)    # {4, 5}
```

```
#método alternativo:
```

```
C = A.intersection(B)
```

```
# ou
```

```
C = B.intersection(A)
```



Operações com conjuntos

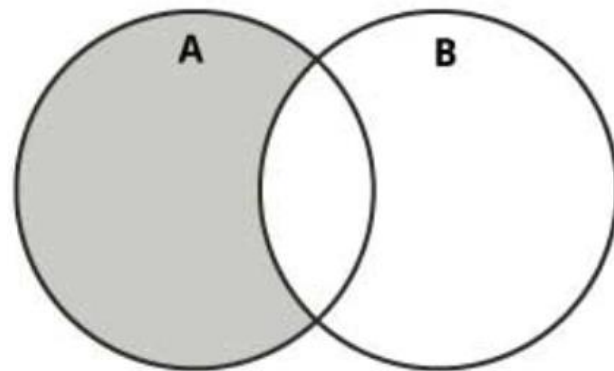
A diferença de conjuntos pode ser realizada com o operador - ou através da função **difference()**.

```
C = A - B
```

```
print(C)    # {1, 2, 3}
```

```
#método alternativo:
```

```
C = A.difference(B)
```



Operações com conjuntos

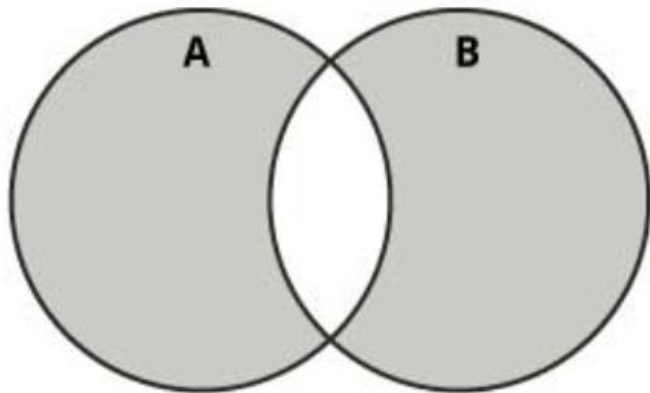
A diferença simétrica de conjuntos pode ser realizada com o operador `^` ou através da função `symmetric_difference()`.

```
C = A ^ B
```

```
print(C)    # {1, 2, 3, 6, 7, 8}
```

```
#método alternativo:
```

```
C = A.symmetric_difference(B)
```



Dicionários

Dicionários

Um dicionário é uma coleção não ordenada, onde os itens são compostos por um par de chave e valor.

A inicialização de um dicionário pode ser feita de duas maneiras:

```
x = dict()  
y = {}
```

Dicionários

Exemplo de dicionário:

```
peessoa = { "nome": "Fulano de tal",  
            "idade": 30 }
```

Neste caso a variável `peessoa` é um dicionário com as chaves “nome” e “idade”, com os valores “Fulano de tal” e 30.

Dicionários

O acesso aos itens de um dicionário é semelhante ao acesso aos itens de uma lista. Utiliza-se os colchetes após o nome da variável. A diferença está no índice, que agora é a chave do elemento que se deseja acessar.

Exemplo:

```
pessoa = { "nome": "Fulano de tal", "idade": 30 }
```

```
print(pessoa["nome"], "tem", pessoa["idade"], "anos")
```


Dicionários

Para remover todos os itens de um dicionário utiliza-se o comando `dicionario.clear()`.

Exemplo:

```
pessoa = { "nome": "Fulano de tal", "idade": 30 }
```

```
pessoa.clear() # {}
```

Dicionários

O comando `dicionario.keys()` retorna uma lista com todas as chaves presentes no dicionário.

Exemplo:

```
pessoa = { "nome": "Fulano de tal", "idade": 30 }  
chaves = pessoa.keys() # ["nome", "idade"]
```

Dicionários

O comando `dicionario.values()` retorna uma lista com todos os valores presentes no dicionário.

Exemplo:

```
pessoa = { "nome": "Fulano de tal", "idade": 30 }  
valores = pessoa.values() # ["Fulano de tal", 30]
```

Dicionários

O comando `dicionario.pop(chave)` remove o item com a chave passada como argumento.

Exemplo:

```
pessoa = { "nome": "Fulano de tal", "idade": 30 }  
pessoa.pop("idade")  
print(pessoa) # { "nome": "Fulano de tal" }
```

Dicionários

O comando `dicionario.items()` retorna uma lista com as tuplas (chave,valor) para cada elemento do dicionário

Exemplo:

```
pessoa = { "nome": "Fulano de tal", "idade": 30 }
```

```
print(pessoa.items())
```

```
# [ ("nome", "Fulano de tal"), ("idade", 30) ]
```

Compreensão de listas

Compreensão de listas

A linguagem Python oferece um recurso bastante prático para trabalhar com listas e iteráveis, que é a compreensão de listas.

Compreensão de listas

A compreensão de listas pode ser entendida como uma forma reduzida e mais rápida de executar um comando **for** com um teste condicional e uma determinada expressão que será aplicada para cada elemento da lista caso o teste condicional retorne verdadeiro.

Tudo isto em uma única operação.

Compreensão de listas

Equivalente sem compreensão de listas:

```
for elemento in lista:  
    if condição:  
        expressão
```

Com compreensão de listas:

```
[expressão for elemento in lista if condição]
```

Compreensão de listas

Exemplo: Selecione somente os elementos pares de uma determinada lista em uma nova lista.

```
listaOriginal = [1,2,3,4,5,6,7,8]
```

```
pares = [ x for x in listaOriginal if (x%2)==0]
```

```
print(pares)
```

Compreensão de listas

Assim como existe a compreensão de listas, existe a compreensão de dicionários, que possui exatamente o mesmo funcionamento

{chave:valor for (chave,valor) in dicionario.items() if condição}

Compreensão de listas

Exercícios:

- 1 – Crie uma lista com dicionários que contenham as informações: nome do aluno e média.
- 2 - Com a lista criada no exercício 1 utilize compreensão de listas para gerar uma nova lista que contenha somente os alunos com média maior ou igual a 6
- 3 – Imprima na tela as informações dos alunos selecionados no exercício 2, em ordem alfabética.

Operações com pastas do sistema operacional

Operações com pastas do sistema operacional

O interpretador do python é executado em um determinado diretório do computador onde ele foi chamado, logo, o caminho para os arquivos pode ser absoluto (“C:\\pasta1\\pasta2\\arquivo1.txt”) ou relativo ao diretório atual (“arquivo1.txt”, considerando que o arquivo está na mesma pasta)

Operações com pastas do sistema operacional

É possível listar os diretórios e arquivos, além de navegar entre diferentes diretórios em tempo de execução.

Para isto será utilizada a biblioteca **os** que é nativa do python (não precisa ser instalada usando o pip).

Operações com pastas do sistema operacional

Para que a biblioteca esteja disponível para o nosso programa é necessário importar ela, isto normalmente é realizado nas primeiras linhas do código.

Exemplo:

```
import os
```

agora a biblioteca os está disponível para ser usada

Operações com pastas do sistema operacional

Principais comandos da biblioteca os:

os.getcwd(): Retorna o caminho do diretório atual

os.chdir(caminho): Navega até o caminho passado como argumento.

os.listdir(caminho): Lista todos os arquivos e pastas do diretório*

*Para recursos mais avançados, utilizar os.scandir()

Acesso a arquivos

Acesso a arquivos

Uma forma bastante comum de armazenar e compartilhar informações é através de arquivos de textos, os quais muitas vezes precisam ser manipulados para se extrair as informações desejadas.

Estes arquivos muitas vezes podem conter uma grande quantidade de informações, e o processamento manual demoraria muito tempo, para isto podemos escrever um programa para processar os arquivos.

Acesso a arquivos

Em python para abrir um arquivo utiliza-se o comando **open()** que retorna um objeto que será utilizado para manipular o arquivo.

O comando open pode receber um ou dois argumentos, sendo estes o nome do arquivo a ser aberto, e o outro o modo de abertura do arquivo.

Acesso a arquivos

Os modos de abertura disponíveis são:

Modo	Significado
r (valor padrão)	Modo de leitura (retorna um erro caso o arquivo não exista), cursor no início do arquivo
r+	Modo de escrita e leitura, cursor no início do arquivo
a	Modo de acrescentação (Abre o arquivo para acrescentar dados, ou cria o arquivo caso este não exista), cursor no final do arquivo
a+	Modo de acrescentação e leitura, cursor no final do arquivo
w	Modo de escrita (Cria o arquivo caso este não exista)
w+	Abre o arquivo para escrita e leitura, sobrescrevendo o arquivo caso já exista

Acesso a arquivos

Exemplo:

```
arquivo = open("meuarquivo.txt","r+")
```

Acesso a arquivos

É necessário fechar um arquivo depois da sua utilização, para que outros processos do computador possam acessar este arquivo.

Para fechar um arquivo, utilizamos o comando `arquivo.close()`

Exemplo:

```
arquivo = open("meuarquivo.txt","r+")  
arquivo.close()
```

Acesso a arquivos

Na manipulação de um arquivo possuímos um cursor, assim como nos editores de texto, para navegar entre diferentes posições do arquivo.

Para saber a posição do cursor utiliza-se o comando `arquivo.tell()`

Acesso a arquivos

Para navegar até uma determinada posição, utiliza-se o comando `arquivo.seek(posição,referencia)`

Os argumentos do comando `seek` são:

- Posição: Nova posição do cursor
- Referencia: 0 (Referência absoluta), 1 (referente a posição atual do cursor) ou 2 (referente ao final do arquivo)

Acesso a arquivos

Para ler uma determinada quantidade de caracteres de um arquivo utiliza-se o comando `arquivo.read(quantidade)` que retornará o conteúdo do arquivo referente a quantidade de caracteres solicitada. Caso a quantidade seja omitida, ele retorna todo o conteúdo do texto.

Acesso a arquivos

Para ler uma única linha de um arquivo utiliza-se o comando `arquivo.readline()` que retornará o conteúdo da linha e moverá o cursor para o início da próxima linha.

Exemplo:

```
arquivo = open("meuarquivo.txt","r")  
print(arquivo.readline())  
arquivo.close()
```

Acesso a arquivos

Para ler uma todas as linhas de um arquivo utiliza-se o comando `arquivo.readlines()` que retornará uma lista com todas as linhas do arquivo.

Exemplo:

```
arquivo = open("meuarquivo.txt","r")  
print(arquivo.readlines())  
arquivo.close()
```

Acesso a arquivos

Para ler uma todas as linhas de um arquivo utiliza-se o comando `arquivo.readlines()` que retornará uma lista com todas as linhas do arquivo.

Exemplo:

```
arquivo = open("meuarquivo.txt","r")  
print(arquivo.readlines())  
arquivo.close()
```

Acesso a arquivos

Para escrever em um arquivo é utilizado o comando `arquivo.write(conteúdo)` onde o argumento são os dados a serem escritos no arquivo.

Exemplo:

```
arquivo = open("meuarquivo.txt","r")
arquivo.write("Texto escrito pelo programa")
arquivo.close()
```

Orientação a objetos

Orientação a objetos

Um dos recursos que torna o uso de Python mais simples e interessante é a possibilidade de trabalhar com a orientação a objetos.

Antes de ver como utilizar isto no código, é necessário entender o que são objetos, classes e suas propriedades.

Orientação a objetos

Classe: Podemos classificar objetos por suas semelhanças de propriedades e funcionalidades.

Vamos comparar as característica de alguns animais e ver como conseguimos classificá-los.

Orientação a objetos

Tanto um cachorro quanto um gato possuem atributos como:

- Nome
- Raça
- Peso

E pode realizar algumas ações como:

- Emitir sons (latir ou miar)
- Andar
- Dormir



Orientação a objetos

Então é possível dizer os animais com estes atributos e ações podem ser modelados em uma **classe** **AnimaisDomesticos**.

Lembrando que os valores destes atributos vão ser diferentes para cada animal pertencente a classe AnimaisDomesticos. Cada animal diferente pertencente a mesma classe é dito como uma **instância** deste animal.

Orientação a objetos

Exemplo de instâncias de animais domésticos:

- Nome: Rex
- Raça: Rottweiler
- Peso: 60 kg

Ações:

- Emitir sons (Latido)
- Andar
- Dormir



Orientação a objetos

Exemplo de instâncias de animais domésticos:

- Nome: Frajola
- Raça: Scottish Fold
- Peso: 3 kg

Ações:

- Emitir sons (Miado)
- Andar
- Dormir



Orientação a objetos

Para transformar isto em código, cria-se uma nova classe, onde as propriedades serão armazenadas em forma de variáveis e as ações, agora chamadas de **métodos**, serão implementadas em forma de funções.

Orientação a objetos

Para criar uma classe utiliza-se o comando **class** seguido do nome da classe, normalmente inicializado por letras maiúsculas e dois pontos, para inicializar o bloco de código.

Exemplo:

```
class AnimaisDomesticos:  
    # Aqui vai o código da classe
```

Orientação a objetos

As classes possuem uma função que é chamada toda vez que uma nova instância é criada, e ela que vai receber os parâmetros que serão passados para esta nova instância.

Esta função é chamada de **construtor** da classe.

O construtor de uma classe em python é implementado na função `__init__` (dois `_` em cada lado)

Orientação a objetos

Exemplo:

```
class AnimaisDomesticos:  
    def __init__(self):  
        print("Animal criado")
```

Em python todas as funções dentro de uma classe devem receber como parâmetro a variável **self** como primeiro parâmetro, este é utilizado para referência de escopo.

Orientação a objetos

Para criar uma instância de uma classe, basta utilizar o nome da classe como se fosse uma função.

Exemplo:

```
Animal1 = AnimaisDomesticos()
```

```
Animal2 = AnimaisDomesticos()
```

Orientação a objetos

Os atributos da classe serão variáveis declaradas dentro da função construtora precedidas por “self.”, neste caso podemos descrever as propriedades dos animais.

Orientação a objetos

```
class AnimaisDomesticos:  
    def __init__(self):  
        print("Animal criado")  
        self.nome = None  
        self.raca = None  
        self.peso = None
```

Obs: None (Valor nulo)

Orientação a objetos

Estas propriedades podem ser acessadas pelas instâncias dos nossos objetos:

```
animal1 = AnimaisDomesticos()
```

```
animal2 = AnimaisDomesticos()
```

```
animal1.nome = "Rex"
```

```
animal2.nome = "Frajola"
```

```
print(animal1.nome, animal2.nome)
```

Orientação a objetos

Os valores dos atributos podem ser passados como parâmetro na função construtora.

```
class AnimaisDomesticos:  
    def __init__(self, nome=None, raca=None, peso=None ):  
        print("Animal criado")  
        self.nome = nome  
        self.raca = raca  
        self.peso = peso
```

Orientação a objetos

Estas propriedades podem ser acessadas pelas instâncias dos nossos objetos:

```
animal1 = AnimaisDomesticos("Rex","Rottweiler",60)
animal2 = AnimaisDomesticos("Frajola","Scottish Fold", 3)

print(animal1.nome, animal1.raca, animal1.peso)
print(animal2.nome, animal2.raca, animal2.peso)
```

Orientação a objetos

Os métodos (ou as ações dos animais) serão implementados como funções, que devem receber a variável self como parâmetro, e podem receber ou não outras variáveis como parâmetro.

Orientação a objetos

```
class AnimaisDomesticos:
```

```
    def __init__(self, nome=None, raca=None, peso=None ):
        print("Animal criado")
        self.nome = nome
        self.raca = raca
        self.peso = peso
    def emitir_som(self, som):
        print("{} diz:{}".format(self.nome, som))
    def andar(self):
        print("{} andou".format(self.nome))
    def dormir(self):
        print("{}: zZzZzZ".format(self.nome))
```

Orientação a objetos

Agora é possível chamar os métodos para as instâncias

```
animal1 = AnimaisDomesticos("Rex","Rottweiler",60)
```

```
animal2 = AnimaisDomesticos("Frajola","Scottish Fold", 3)
```

```
animal1.emitir_som("Au Au")
```

```
animal1.andar()
```

```
animal1.dormir()
```

```
animal2.emitir_som("miau miau")
```

```
animal2.andar()
```

```
animal2.dormir()
```

Herança

Herança

Apesar de os gatos e cachorros possuírem algumas características em comum, que classificamos na classe `AnimaisDomesticos` cada um possui características que os tornam animais diferentes.

Então podemos criar uma classe para cada tipo de animal, no caso a classe `Cachorro` e a classe `Gato`, que vão herdar as propriedades e métodos da classe `Animais domésticos`.

Herança

Para criar uma classe com herança, passa-se o nome da classe pai entre parênteses após a definição do nome da nova classe.

Exemplo:

```
class Cachorro(AnimaisDomesticos):  
    # bloco de código da classe cachorro
```

```
class Gato(AnimaisDomesticos):  
    #bloco de código da classe Gato
```

Herança

Agora podemos modificar os métodos e criar novas propriedades, ou métodos para cada tipo de animal.

Exemplo:

```
class Cachorro(AnimaisDomesticos):  
    def emitir_som(self):  
        print("{}: Au Au Au".format(self.nome))  
  
class Gato(AnimaisDomesticos):  
    def emitir_som(self):  
        print("{}: Miau Miau".format(self.nome))  
  
animal1 = Cachorro("Rex", "Rottweiler", 60)  
animal2 = Gato("Frajola", "Scottish Fold", 3)  
animal1.emitir_som()  
animal2.emitir_som()
```

Herança

Caso deseje passar mais parâmetros para o construtor de uma classe que herdou o construtor de outra classe, não é necessário repetir o código do construtor, pode-se fazer referência a classe pai utilizando o comando **super()**

Herança

Exemplo:

```
class Cachorro(AnimaisDomesticos):  
    def __init__(self,nome=None,raca=None,peso=None,porte="Grande"):  
        super().__init__(nome,raca,peso)  
        self.porte = porte  
  
    def emitir_som(self):  
        print("{}: Au Au Au".format(self.nome))  
  
class Gato(AnimaisDomesticos):  
    def emitir_som(self):  
        print("{}: Miau Miau".format(self.nome))
```