print()

- Saída de dados
- Imprime o código

In [1]:

```
# Textos
print ('Olá Mundo!')
```

```
Olá Mundo!
```

In [2]:

```
# Números e operações matemáticas
print (2020)
print (5 + 8)
```

```
2020
13
```

In [ ]:

```
# Variáveis
nome = 'Cosmo'
print (nome)
```

Criando variáveis

Variáveis são contêineres para armazenar valores de dados.

Diferente de outras linguagens de programação, o Python não tem comando para declarar uma variável.

Uma variável é criada no momento em que você atribui um valor a ela.

In [5]:

```
# nome da variável = tipo da variável
curso = 'Python'
ano = 2020
preco = 50.00
ativo = True

print (curso)
print (ano)
print (preco)
print (ativo)
```

```
Python
2020
50.0
True
```

Tipos de dados

Na programação, o tipo de dados é um conceito importante.

Variáveis podem armazenar dados de tipos diferentes e tipos diferentes podem fazer coisas diferentes.

- **int** -> Números inteiros
- **float** -> Números reais
- **str** -> String - coleção de caracteres - Textos
- **bool** -> Expressão booleanas - True ou False
- **type** ( )
- Mostra o tipo da variável

In [6]:

```
nome = 'Cosmo' # int
print (type(nome))
print (nome)

idade = 35 # int
print (type(idade))
print (idade)

altura = 1.75 # float
print (type(altura))
print (altura)

casado = True
print (type(casado))
print (casado)
```

```
<class 'str'>
Cosmo
<class 'int'>
35
<class 'float'>
1.75
<class 'bool'>
True
```

Python Strings

- string em python são cercados por aspas simples ou aspas duplas -> 'Python' , "Python" .

Strings multilinhas

- Você pode atribuir uma sequência multilinha a uma variável usando três aspas:

In [7]:

```
print ('Python')

print ("Python")

print (''' Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua ''')
```

```
Python
Python
 Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua
```

Fatiamento

Você pode retornar um intervalo de caracteres usando a sintaxe da fatia.

Especifique o índice inicial e o índice final, separados por dois pontos, para retornar uma parte da sequência.

- **len** ( ) -> Mostra o tamanho da string

In [8]:

```
# 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
# C u r s o   d e   P  y t  h  o  n

curso = 'Curso de Python'
print (len(curso))
```

```
15
```

In [15]:

```
print (curso[9])
print (curso[0:5])
print (curso[6:15])
```

```
P
Curso
de Python
```

Métodos de String

- O Python possui um conjunto de métodos internos que você pode usar em strings.
- O **strip** () método remove qualquer espaço em branco do começo ou do fim:
- O **lower** ()método retorna a string em minúsculas:
- O **upper** ()método retorna a string em maiúsculas:
- O **replace** ()método substitui uma string por outra string:
- O **split** ()método divide a string em substrings se encontrar instâncias do separador:
- Para verificar se uma determinada frase ou caractere está presente em uma string, podemos usar as palavras-chave **in** ou **not in**.

In [18]:

```
# strip ()
curso = '       Curso de Python          '
print (curso)
print (curso.strip())
```

```
       Curso de Python
Curso de Python
```

In [19]:

```
#lower ()
curso = 'Curso de Python'
print (curso.lower())
```

```
curso de python
```

In [20]:

```
# upper ()
curso = 'Curso de Python'
print (curso.upper())
```

```
CURSO DE PYTHON
```

In [21]:

```
# replace ()
curso = 'Curso de Python'
print(curso.replace('e','3'))
```

```
Curso d3 Python
```

In [22]:

```
#  split()
curso = 'Curso de Python'
print (curso.split())
```

```
['Curso', 'de', 'Python']
```

In [23]:

```
# in ou not in
curso = 'Curso de Python'
x = 'de' in curso
z = 'de' not in curso
print (x)
print (z)
```

```
True
False
```

Operadores Python

Operadores são usados para executar operações em variáveis e valores.

Python divide os operadores nos seguintes grupos:

- Operadores aritméticos
- Operadores de atribuição
- Operadores de comparação
- Operadores lógicos
- Operadores de identidade
- Operadores de associação
- Operadores bit a bit

Operadores aritméticos de Python

- Operadores aritméticos são usados com valores numéricos para executar operações matemáticas comuns:
- **+** Adição x + y
- **-** Subtração x - y
- * Multiplicação x * y
- **/** Divisão x / y
- **%** Modulo x % y
- ** Exponenciação x ** y
- **//** Resto da divisão x // y

In [24]:

```
x = 10
y = 3
```

In [25]:

```
print (x + y)
```

```
13
```

In [26]:

```
print (x - y)
```

```
7
```

In [27]:

```
print (x * y)
```

```
30
```

In [28]:

```
print (x / y)
```

```
3.3333333333333335
```

In [29]:

```
print (x % y)
```

```
1
```

In [30]:

```
print (x // y)
```

```
3
```

In [33]:

```
print (x ** y)
```

```
1000
```

Operadores de atribuição do Python

- Operadores de atribuição são usados para atribuir valores a variáveis:

In [34]:

```
# =          x = 5
# +=         x+= 3   x = x + 3
# -=         x-= 3   x = x - 3
```

In [35]:

```
x = 5
x += 3

print(x)
```

```
8
```

In [36]:

```
x = 5
x -= 3

print(x)
```

```
2
```

Operadores de comparação Python

Operadores de comparação são usados para comparar dois valores:

In [ ]:

```
Operador        Nome                    Examplo
==              igual                    x == y
!=              Diferente                   x != y
>               Maior que                   x > y
<               Menos que                   x < y
>=              Maior ou iguall          x >= y
<=              Menor ou igual           x <= y
```

In [38]:

```
x = 5
y = 3

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

```
False
True
True
False
True
False
```

Operadores lógicos Python

Operadores lógicos são usados para combinar instruções condicionais:

In [ ]:

```
Operador            Descrição
and              Retorna True se as duas instruções forem verdadeiras
or               Retorna True se uma das instruções for verdadeira
not              inverter o resultado, retorna False se o resultado for verdadeiro
```

In [40]:

```
x = 5

print(x > 3 and x < 10)
```

```
True
```

In [41]:

```
x = 5

print(x > 3 or x < 4)
```

```
True
```

In [42]:

```
x = 5

print(not(x > 3 and x < 10))
```

```
False
```

Operadores de identidade Python

Os operadores de identidade são usados para comparar os objetos, não se forem iguais, mas se forem realmente o mesmo objeto, com o mesmo local de memória:

In [ ]:

```
is       Retorna True se as duas variáveis forem o mesmo objeto
is not   Retorna True se as duas variáveis não forem o mesmo objeto
```

In [43]:

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)
```

```
True
```

In [44]:

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is not z)
```

```
False
```

Entrada de dados

input ()

In [45]:

```
nome = input ('Digite seu nome: ')
print (nome)
```

```
Digite seu nome: Cosmo
Cosmo
```

Listas de Python

Lista [ ]

- Uma lista é uma coleção que é ordenada e mutável. No Python, as listas são escritas com colchetes.

In [46]:

```
lista = ["maçã", "banana", "cereja"]
print (lista)
```

```
['maçã', 'banana', 'cereja']
```

In [47]:

```
# Você acessa os itens da lista consultando o número do índice:

lista = ["maçã", "banana", "cereja"]
print (lista[1])
```

```
banana
```

In [48]:

```
# Intervalo de índices
# Você pode especificar um intervalo de índices especificando por onde começar e
# por onde terminar o intervalo.

# Ao especificar um intervalo, o valor retornado será uma nova lista com os itens spec
ificados.

frutas = ["maçã", "banana", "cereja", "laranja", "kiwi", "melão", "manga"]
print (frutas[2:5])
```

```
['cereja', 'laranja', 'kiwi']
```

In [49]:

```
print (frutas[:4])
```

```
['maçã', 'banana', 'cereja', 'laranja']
```

In [50]:

```
print (frutas[2:])
```

```
['cereja', 'laranja', 'kiwi', 'melão', 'manga']
```

In [51]:

```
# Alterar valor do item
# Para alterar o valor de um item específico, consulte o número do índice:

lista = ['maçã', 'banana', 'cereja']
lista [1] = 'uva'
print (lista)
```

```
['maçã', 'uva', 'cereja']
```

Adicionar itens

- Para adicionar um item ao final da lista, use o método **append**() :

In [52]:

```
lista = ['maçã', 'banana', 'cereja']
lista.append ('abacaxi')
print (lista)
```

```
['maçã', 'banana', 'cereja', 'abacaxi']
```

Para adicionar um item ao índice especificado

- use o método **insert** () :

In [54]:

```
frutas = ['maçã', 'banana', 'cereja', 'abacaxi']
frutas.insert (2, 'laranja')
print (frutas)
```

```
['maçã', 'banana', 'laranja', 'cereja', 'abacaxi']
```

Remover item

- Existem vários métodos para remover itens de uma lista:
- O **remove** () método remove o item especificado:

In [55]:

```
frutas = ['maçã', 'banana', 'cereja', 'abacaxi']
frutas.remove ('cereja')
print (frutas)
```

```
['maçã', 'banana', 'abacaxi']
```

- O **pop** () método remove o índice especificado (ou o último item se o índice não for especificado):

In [57]:

```
frutas = ['maçã', 'banana', 'abacaxi' ]
frutas.pop ()
print (frutas)
```

```
['maçã', 'banana']
```

- A **del** palavra-chave remove o índice especificado:

In [59]:

```
frutas = ['maçã', 'banana', 'abacaxi' ]
del frutas[0]
print (frutas)
```

```
['banana', 'abacaxi']
```

- O **clear** () método esvazia a lista:

In [60]:

```
frutas = ['maçã', 'banana', 'abacaxi' ]
frutas.clear()
print (frutas)
```

```
[]
```

O construtor list ()

- Também é possível usar o construtor **list** () para criar uma nova lista.
- Usando o list() construtor para fazer uma lista:

In [63]:

```
cores = list (('verde','azul', 'roxo'))
print (cores)
```

```
['verde', 'azul', 'roxo']
```

Tuplas

- Uma tupla é uma coleção ordenada e imutável .
- No Python, as tuplas são escritas com parênteses ( ) .

In [64]:

```
cores = ('verde', 'azul', 'roxo')
print (cores)
```

```
('verde', 'azul', 'roxo')
```

Acessar itens da tupla

- Você pode acessar itens da tupla consultando o número do índice, entre colchetes:

In [65]:

```
cores = ('verde', 'azul', 'roxo')
print (cores[0])
```

```
verde
```

Remover itens

- Nota: Você não pode remover itens em uma tupla.
- As tuplas são imutáveis , portanto você não pode remover itens, mas é possível excluir a tupla completamente:
- A **del** palavra-chave pode excluir completamente a tupla:

In [67]:

```
cores = ('verde', 'azul', 'roxo')
del cores
```

In [68]:

```
print (cores)
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
<ipython-input-68-724034ae01c7> in <module>
----> 1 print (cores)

NameError: name 'cores' is not defined
```

Conjuntos Python - Sets

- Um conjunto é uma coleção não ordenada e não indexada. No Python, os conjuntos são escritos com chaves { }.
- Nota: Os conjuntos não são ordenados, portanto, você não pode ter certeza de qual ordem os itens serão exibidos

In [69]:

```
cores = {'verde', 'azul', 'laranja'}
print (cores)
```

```
{'verde', 'laranja', 'azul'}
```

Alterar itens

- Depois que um conjunto é criado, você não pode alterar seus itens, mas pode adicionar novos itens.
- Adicionar itens
- Para adicionar um item a um conjunto, use o método **add** ()
- Para adicionar mais de um item a um conjunto, use o método **update** ()

In [70]:

```
# Adicione um item a um conjunto, usando o add() método:
cores = {'verde', 'azul', 'laranja'}
cores.add('roxo')
print (cores)
```

```
{'roxo', 'verde', 'laranja', 'azul'}
```

In [71]:

```
# Adicione vários itens a um conjunto, usando o update() método:
cores = {'verde', 'azul', 'laranja'}
cores.update(['vermelho','preto','amarelo'])
print (cores)
```

```
{'preto', 'amarelo', 'verde', 'laranja', 'azul', 'vermelho'}
```

Remover item

- Para remover um item de um conjunto, use o método **remove** ( ), ou o **discard** ()

In [72]:

```
cores = {'verde', 'azul', 'laranja'}
cores.remove('azul')
print (cores)
```

```
{'verde', 'laranja'}
```

Dicionários em Python

- Um dicionário é uma coleção desordenada, mutável e indexada. No Python, os dicionários são escritos com chaves e possuem chaves e valores.

In [73]:

```
cidades = {
    'SP': 'São Paulo',
    'DF': 'Distrito Federal',
    'RJ': 'Rio de Janeiro'
}

print (cidades)
```

```
{'SP': 'São Paulo', 'DF': 'Distrito Federal', 'RJ': 'Rio de Janeiro'}
```

Acessando itens

- Você pode acessar os itens de um dicionário consultando o nome da chave, entre colchetes:

In [74]:

```
print (cidades['DF'])
```

```
Distrito Federal
```

- Também existe um método chamado **get**( ) que fornecerá o mesmo resultado:

In [75]:

```
print (cidades.get('SP'))
```

```
São Paulo
```

Mudar valores

- Você pode alterar o valor de um item específico consultando o nome da chave:

In [76]:

```
dic = {1 : 'A', 2: 'B', 3 : 'C'}
print (dic)
dic[1] = 'D'
print (dic)
```

```
{1: 'A', 2: 'B', 3: 'C'}
{1: 'D', 2: 'B', 3: 'C'}
```

- Você também pode usar a **values** ()função para retornar valores de um dicionário:

In [77]:

```
dic = {1 : 'A', 2: 'B', 3 : 'C'}
print (dic.values())
```

```
dict_values(['A', 'B', 'C'])
```

- Você também pode usar a **keys** ()função para retornar as chaves de um dicionário:

In [78]:

```
dic = {1 : 'A', 2: 'B', 3 : 'C'}
print (dic.keys())
```

```
dict_keys([1, 2, 3])
```

- Você também pode usar o **items** () função para retornar os itens de um dicionário:

In [79]:

```
dic = {1 : 'A', 2: 'B', 3 : 'C'}
print (dic.items())
```

```
dict_items([(1, 'A'), (2, 'B'), (3, 'C')])
```

Removendo itens

- O **pop** () método remove o item com o nome da chave especificado:

In [81]:

```
dic = {1 : 'A', 2: 'B', 3 : 'C'}
print (dic.pop(1))
print (dic)
```

```
A
{2: 'B', 3: 'C'}
```

Python If ... Else

In [82]:

```
idade = 18
if idade >= 18:
    print ('Maior')
else:
    print ('Menor')
```

```
Maior
```

Elif

- A palavra-chave **elif** é uma forma em python de dizer "se as condições anteriores não fossem verdadeiras, tente esta condição".

In [91]:

```
idade = 18
if idade < 12:
    print('crianca')
elif idade < 18:
    print('adolescente')
elif idade < 60:
    print('adulto')
else:
    print('idoso')
```

```
adulto
```

Python Loops

- O Python possui dois comandos primitivos de loop:
- **while** loops
- **for** loops

O loop while

- Com o **while** podemos executar um conjunto de instruções enquanto uma condição for verdadeira.

In [92]:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

```
1
2
3
4
5
```

A declaração break

- Com a instrução **break** , podemos parar o loop mesmo se a condição while for verdadeira:

In [93]:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

```
1
2
3
```

A declaração continue

- Com a instrução **continue** , podemos parar a iteração atual e continuar com a próxima:

In [94]:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
1
2
4
5
6
```

A declaração else

- Com a instrução **else** , podemos executar um bloco de código uma vez quando a condição não for mais verdadeira:

In [95]:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print('FIM !')
```

```
1
2
3
4
5
FIM !
```

Loop For

- Um laço **for** é utilizado para a iteração através de uma sequência (isto é, quer uma lista, um tuplo, um dicionário, de um conjunto, ou uma cadeia).
- Isso é menos parecido com a palavra-chave for em outras linguagens de programação e funciona mais como um método iterador, como encontrado em outras linguagens de programação orientadas a objetos.
- Com o loop for , podemos executar um conjunto de instruções, uma vez para cada item de uma lista, tupla, conjunto etc.

In [96]:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

```
apple
banana
cherry
```

Loop através de strings

- Mesmo seqüências de caracteres são objetos iteráveis, elas contêm uma sequência de caracteres:

In [97]:

```
curso = 'Curso de python'
for x in curso:
    print (x)
```

```
C
u
r
s
o

d
e

p
y
t
h
o
n
```

A função range ()

- Para percorrer um conjunto de códigos um número especificado de vezes, podemos usar a função **range** () ,
- A função range () retorna uma sequência de números, iniciando em 0 por padrão e incrementando em 1 (por padrão), e termina em um número especificado.

In [98]:

```
for x in range(10):
  print(x)
```

```
0
1
2
3
4
5
6
7
8
9
```

- A função range () é padronizada como 0 como valor inicial, no entanto, é possível especificar o valor inicial adicionando um parâmetro: range (2, 6) , que significa valores de 2 a 6 (mas não incluindo 6):

In [99]:

```
for x in range(2, 6):
  print(x)
```

```
2
3
4
5
```

- A função range () é padronizada para incrementar a sequência em 1, no entanto, é possível especificar o valor do incremento adicionando um terceiro parâmetro: range (2, 30, 3) :

In [100]:

```
for x in range(2, 30, 3):
  print(x)
```

```
2
5
8
11
14
17
20
23
26
29
```

Funções em Python

- Uma função é um bloco de código que só é executado quando é chamado.
- Você pode passar dados, conhecidos como parâmetros, para uma função.
- Uma função pode retornar dados como resultado.
- Criando uma Função
- Em Python, uma função é definida usando a palavra-chave **def** :

In [ ]:

```
def minha_func():
    print ('Esta é minha função')
```

- Chamando uma função
- Para chamar uma função, use o nome da função seguido por parênteses:

In [101]:

```
def minha_func():
    print ('Esta é minha função')

minha_func ()
```

```
Esta é minha função
```

Argumentos

- As informações podem ser passadas para funções como argumentos.
- Os argumentos são especificados após o nome da função, entre parênteses. Você pode adicionar quantos argumentos quiser, basta separá-los com uma vírgula.

In [102]:

```
def minha_func(saudacao):
    print (saudacao)

minha_func('Olá!')
minha_func('Olá tudo bem!')
minha_func('Olá como vai!')
```

```
Olá!
Olá tudo bem!
Olá como vai!
```

Argumentos arbitrários, * args

- Se você não souber quantos argumentos serão passados para sua função, adicione a *antes do nome do parâmetro na definição da função.

In [104]:

```
def minha_func(*args):
    print (args)

minha_func('azul', 'verde','roxo', 'amarelo')
```

```
('azul', 'verde', 'roxo', 'amarelo')
```

Argumentos arbitrários de palavras-chave, ** kwargs

- Se você não souber quantos argumentos de palavra-chave serão passados para sua função, adicione dois asterisco: ** antes do nome do parâmetro na definição da função.
- Dessa forma, a função receberá um dicionário de argumentos e poderá acessar os itens de acordo:

In [105]:

```
def minha_func(**kwargs):
    print (kwargs)

minha_func(nome='Cosmo',idade=35)
```

```
{'nome': 'Cosmo', 'idade': 35}
```

Retornar valores

- Para permitir que uma função retorne um valor, use a instrução **return** :

In [106]:

```
def minha_func(num1 , num2):
    return num1 + num2

print (minha_func(5,8))
```

```
13
```

Python Lambda

- Uma função lambda é uma pequena função anônima.
- Uma função lambda pode receber qualquer número de argumentos, mas pode ter apenas uma expressão.

Sintaxe

- **lambda arguments : expression**
- A expressão é executada e o resultado é retornado:

In [1]:

```
# Exemplo
# Uma função lambda que adiciona 10 ao número passado como argumento e imprime o result
ado:

x = lambda a : a + 10
print(x(5))
```

```
15
```

In [2]:

```
#As funções do Lambda podem receber qualquer número de argumentos:

#Exemplo
#Uma função lambda que multiplica o argumento a pelo argumento be imprime o resultado:

x = lambda a, b : a * b
print(x(5, 6))
```

```
30
```

In [3]:

```
# Use essa definição de função para criar uma função que sempre dobre o número que você envia:

#Exemplo
def myfunc(n):
  return lambda a : a * n

dobro = myfunc(2)

print(dobro(11))
```

```
22
```

Classes / objetos Python

- Python é uma linguagem de programação orientada a objetos.
- Quase tudo no Python é um objeto, com suas propriedades e métodos.
- Uma classe é como um construtor de objetos ou um "blueprint" para criar objetos.

Criar uma classe

- Para criar uma classe, use a palavra chave - **class**:
- Exemplo
- Crie uma classe chamada MyClass, com uma propriedade chamada x:

In [4]:

```
class MyClass:
  x = 5

print(MyClass)
```

```
<class '__main__.MyClass'>
```

Criar Objeto

- Agora podemos usar a classe chamada MyClass para criar objetos:
- Exemplo
- Crie um objeto chamado p1 e imprima o valor de x:

In [5]:

```
class MyClass:
  x = 5

p1 = MyClass()
print(p1.x)
```

```
5
```

A função init ()

- Os exemplos acima são classes e objetos em sua forma mais simples e não são realmente úteis em aplicativos da vida real.
- Para entender o significado das classes, precisamos entender a função interna **init** ().
- Todas as classes têm uma função chamada **init** (), que é sempre executada quando a classe está sendo iniciada.
- Use a função **init** () para atribuir valores às propriedades do objeto ou outras operações necessárias quando o objeto está sendo criado:

In [7]:

```
# - Exemplo
# - Crie uma classe chamada Pessoa, use a função __init __ () para atribuir valores
# para nome e idade:

class Pessoa:
  def __init__(self, nome,idade):
    self.nome = nome
    self.idade = idade

p1 = Pessoa("Lucas", 14)

print(p1.nome)
print(p1.idade)
```

```
Lucas
14
```

- Nota: A **init()** função é chamada automaticamente toda vez que a classe está sendo usada para criar um novo objeto.

Métodos de objeto

- Objetos também podem conter métodos. Métodos em objetos são funções que pertencem ao objeto.
- Vamos criar um método na classe Pessoa:
- Exemplo
- Insira uma função que imprima uma saudação e execute-a no objeto p1:

In [8]:

```
class Pessoa:
  def __init__(self, nome,idade):
    self.nome = nome
    self.idade = idade

  def saudacao(self):
    print ('Olá meu nome é ', self.nome)

p1 = Pessoa("Lucas", 14)
print (p1.saudacao())
```

```
Olá meu nome é  Lucas
None
```

- **Nota** : O parâmetro **self** é uma referência à instância atual da classe e é usado para acessar variáveis que pertencem à classe.

In [ ]:

```

```