Programação em Python

Curso de Especialização Tecnológica (CET) 01/2025 - Técnico/a Especialista em Automação, Robótica e Manutenção Industrial

UC00606 - Desenvolver programas em linguagem estruturada

FORMADOR/A
Bruno Silva

DATA
Junho de 2026

PESSOAS 2030
PORTUGAL 2030
Cofinanciado pela União Europeia
Os Fundos Europeus mais próximos de si.

Índice

Capítulo 1 - Introdução ao Python

1.1 – Python

- *Python* é uma linguagem de programação popular de alto nível;
- Foi criada por Guido van Rossum e lançada em 1991;
- Hoje em dia, o Python é muito utilizado nas empresas de programação para desenvolver websites, componentes de software e aplicativos ou para trabalhar com tecnologias de ciência de dados, Inteligência Artificial AI) e Aprendizagem máquina (ML).

Pode ser usado para:

- Desenvolvimento de software;
- Desenvolvimento web;
- Matemática e Ciência de dados;
- Script de sistemas;
- Cibersegurança;
- Entre outros ...

1.2 – Vantagens da utilização de Python

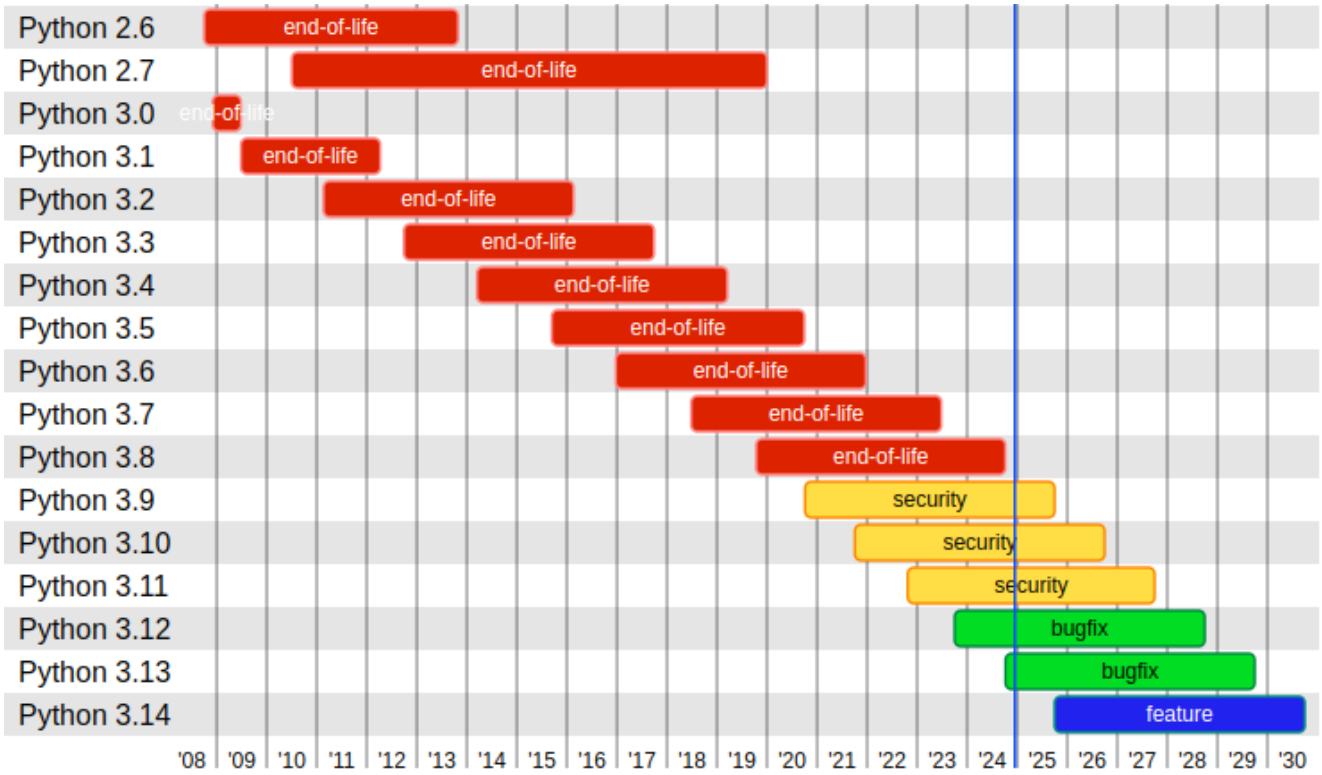
- Funciona em diferentes plataformas (Windows, Mac, Linux, etc);
- Tem uma sintaxe simples semelhante a linguagem do quotidiano;
- Funciona num sistema interpretador, o que significa que o código pode ser executado assim que é escrito;
- Python pode ser tratado de forma estruturada, orientada a objetos ou funcional;

1.3 – Versões

Existem diversas versões do Python, como 2.6, 2.7, 3.0, 3.1, 3.2, 3.3 e assim sucessivamente, conforme novas versões vão surgindo.

Cada uma das versões possui algumas diferenças significativas. No geral, recomenda-se instalar a última versão disponível para garantir as atualizações e melhorias mais recentes.

Atualmente, estamos a versão estável 3.14 (lançado a 7 de Outubro de 2025).



1.4 – Configuração e instalação

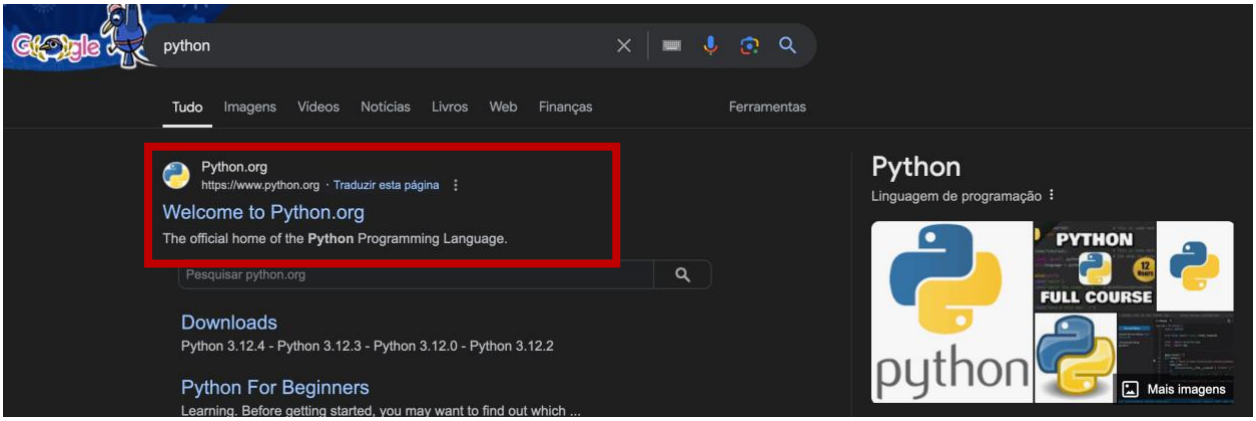
Verificar se o sistema operativo tem Python instalado. Abra um terminal do S.O. e digite o seguinte código:

Windows: python --version

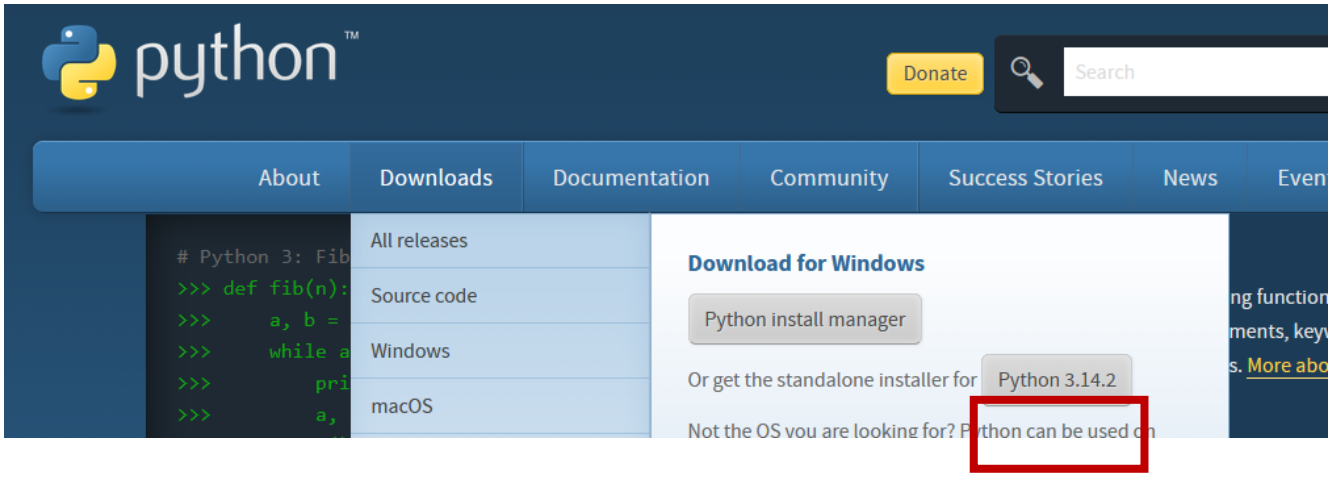
Linux, MacOS: python3 –version

Se ainda não tiver o Python instalado, vamos seguir os próximos passos:

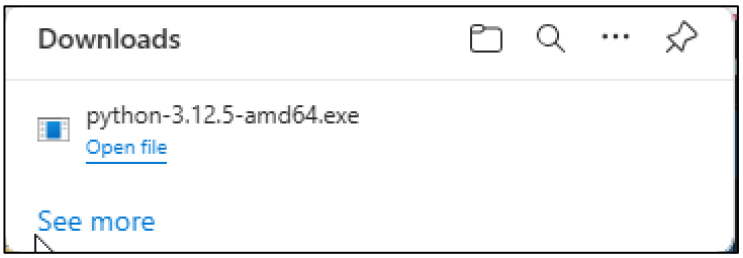
Passo 1 – Ir ao motor de busca e digitar Python e entrar na página oficial ou clicar nesta hiperligação: *https://www.python.org/*



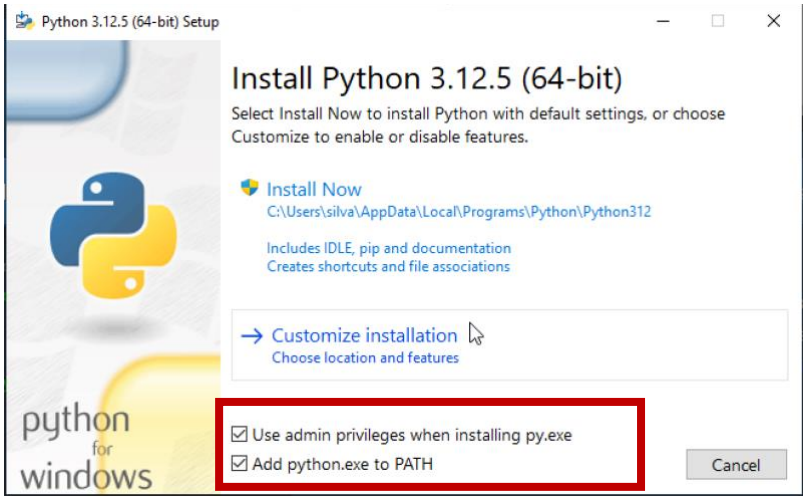
Passo 3 – No meu de navegação, ***escolher a opção Downloads*** e escolher o S.O. mais apropriado (normalmente, este já deteta automaticamente):



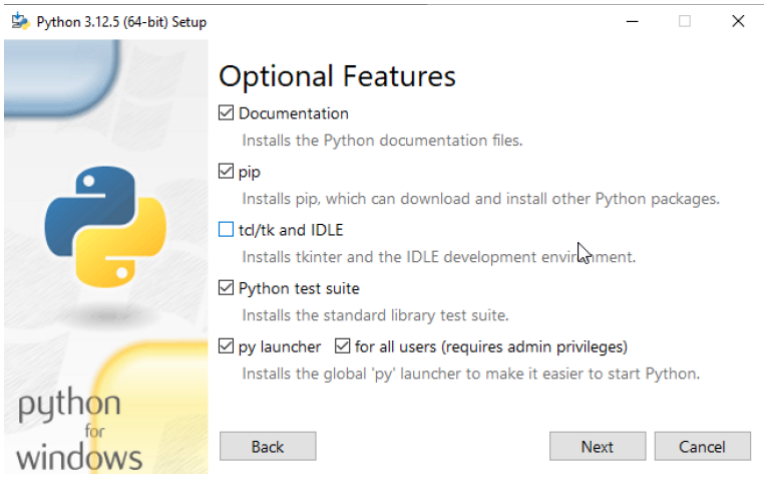
Passo 4 – Ao clicar no botão da versão Python, este vai perguntar onde deseja guardar o ficheiro da instalação e clique no ficheiro para executar:



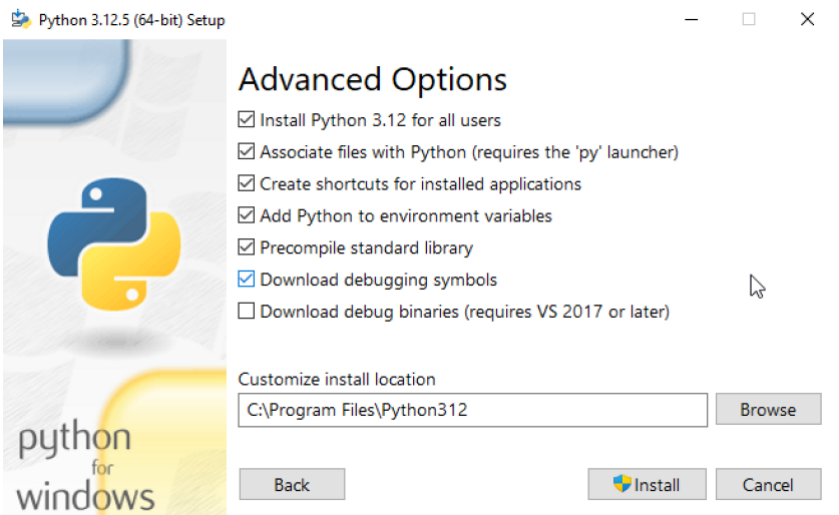
Passo 5 – Selecione as duas opções no fundo da janela e depois clique na opção "Customize installation" (pois vai permitir instalar o python em modo de administrador e colocar a localização para executar os programas nas variáveis do sistema):



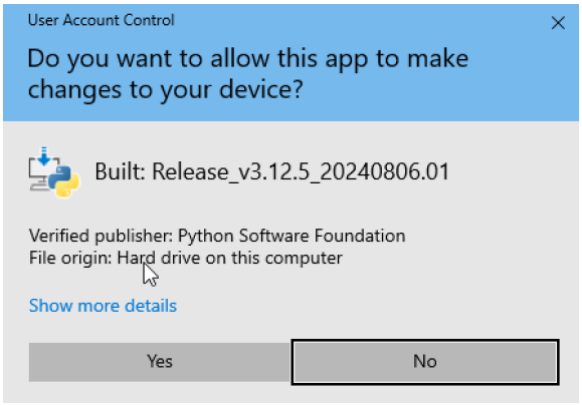
Passo 6 – Verifique se tem todas as opções selecionadas (com exceção da 3ª opção):



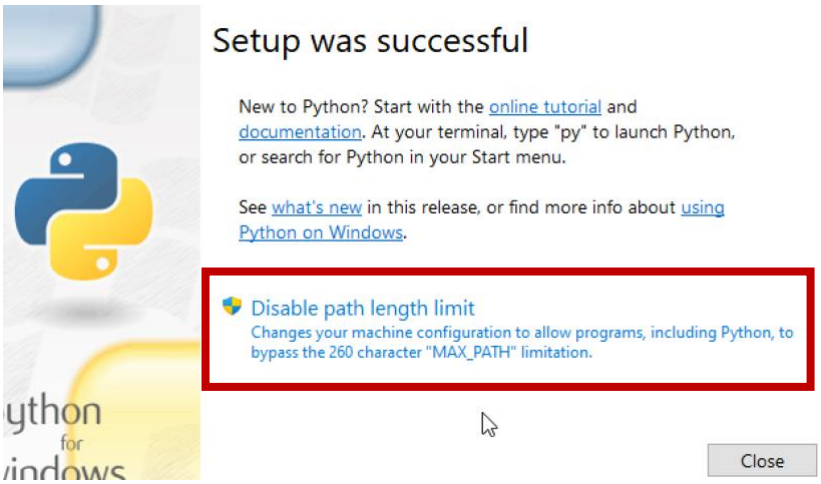
Passo 7 – Verifique se tem todas as opções selecionadas (com exceção da última) e verificar o local da instalação:



Se aparecer o aviso da confirmação de instalação, deve aceitar o mesmo:

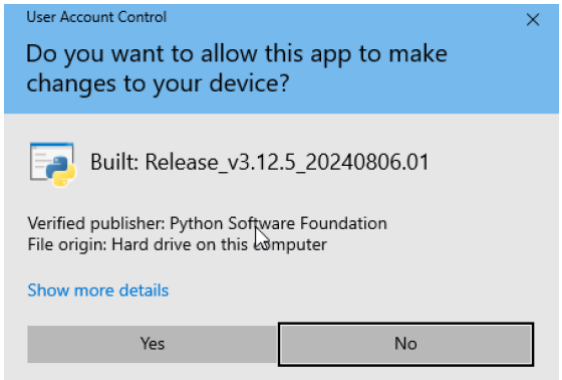


Passo 8 – Após a instalação, poderá aparecer esta mensagem:

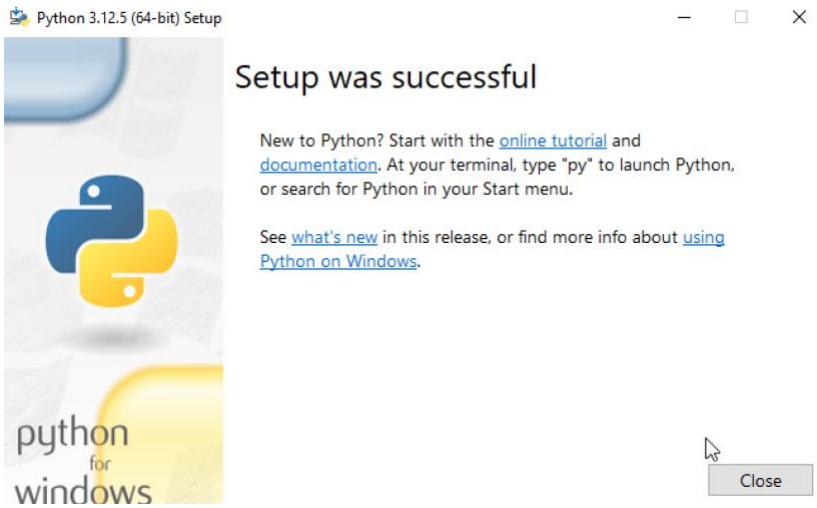


Caos positivo, deve clicar na opção "Disable path lentgh limit", para permitir que a variável do sistema que identifica o caminho onde o Python está instalado tenha todas as instruções e não dar problemas.

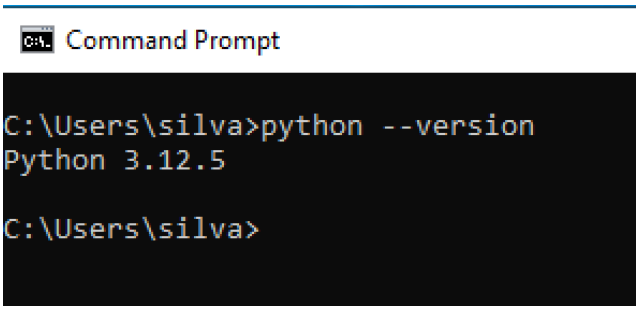
Passo 9 – Se aparecer o aviso da confirmação de instalação, deve aceitar o mesmo:



Passo 10 – No final, será exibida a caixa com o sucesso da instalação:



Passo 11 – Para confirmar a instalação, vamos abrir uma linha de comandos e digitar a instrução ***python --version***:



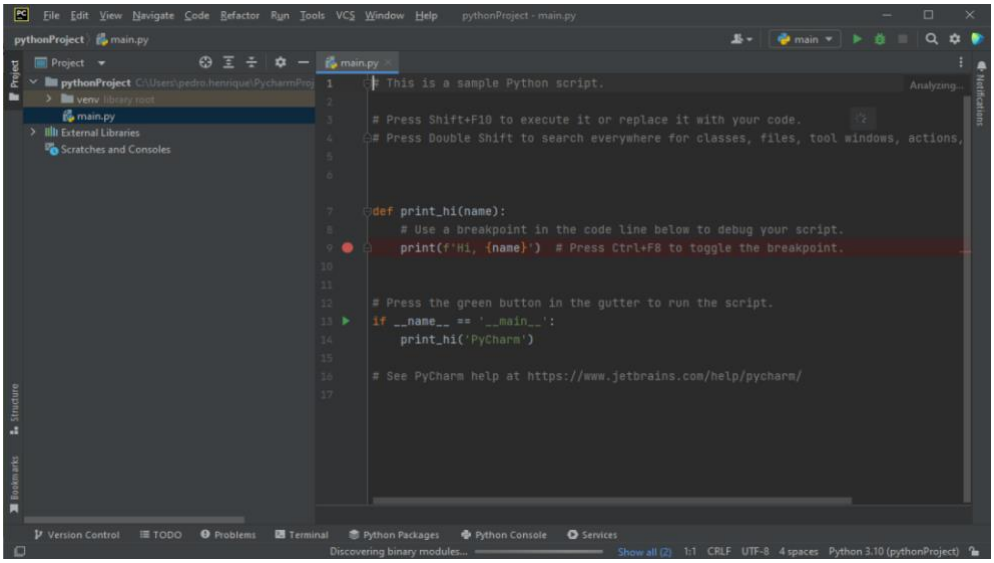
1.5 – Configuração e instalação

Existem vários editores de programação para python, tais como:

- PyCharm
- Visual Studio Code
- Sublime Text
- Atom
- Spyder

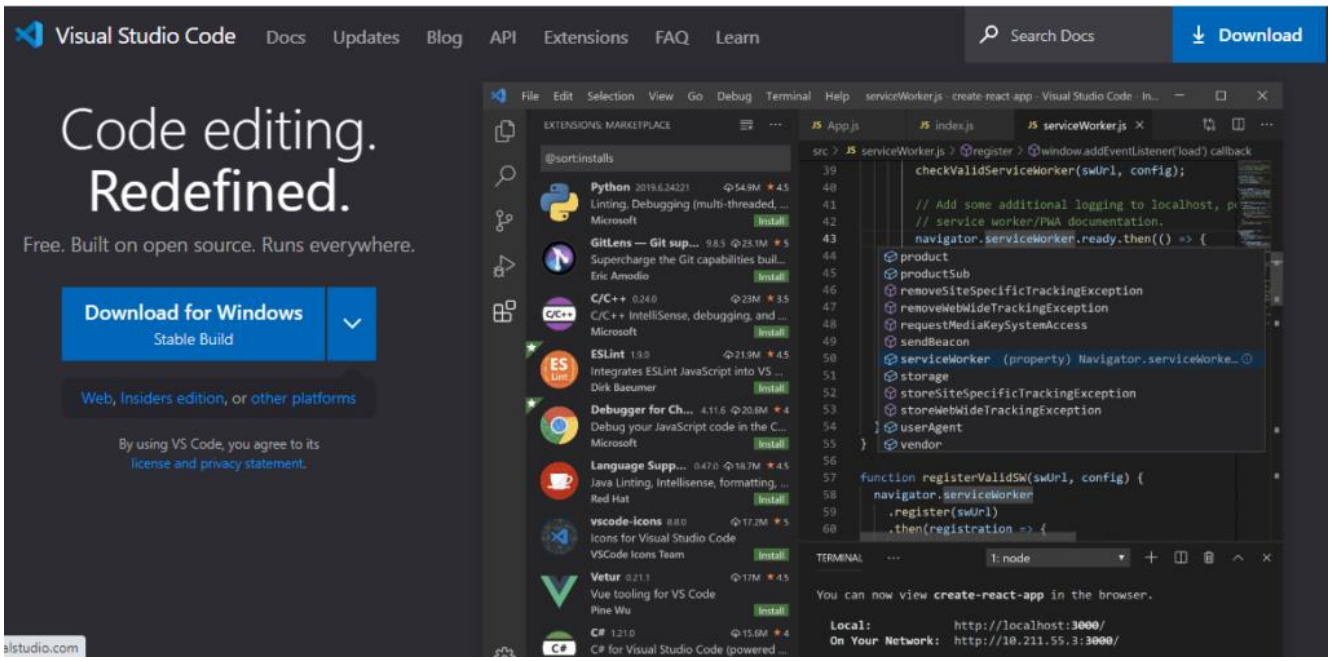
PyCharm

- O PyCharm é um excelente editor de código e é reconhecido pela qualidade e recursos avançados. Este oferece várias ferramentas de desenvolvimento e está disponível em 2 versões: uma gratuita e outra paga, que inclui recursos adicionais.



Visual Studio Code

Um editor de código multi-plataforma, desenvolvido pela Microsoft, com suporte para Python e muitas extensões disponíveis na loja de extensões. Tem sido um programa de excelência no que toca a programação de conteúdos.

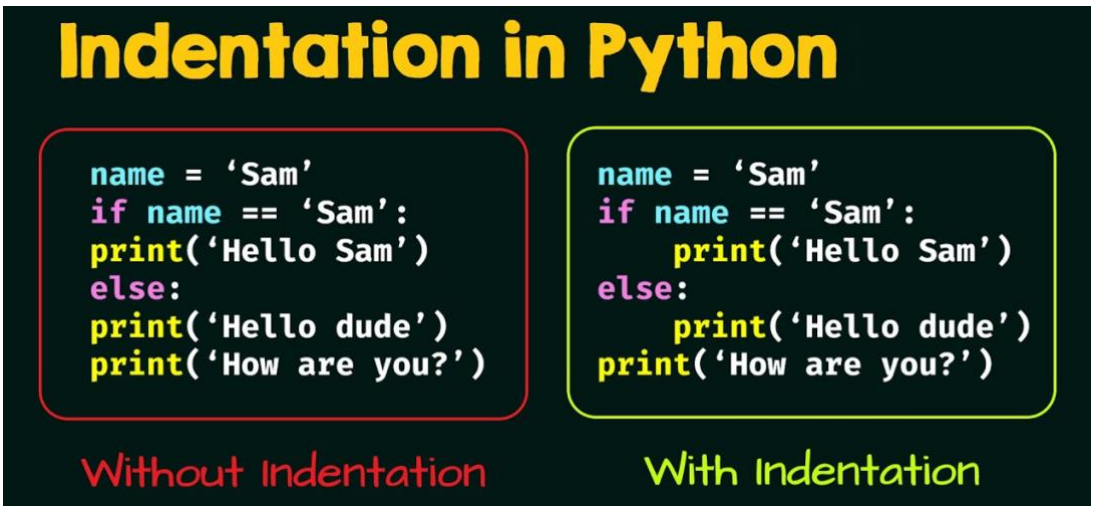


Capítulo 2 – Conceitos genéricos de programação em Python

2.1 – Indentação de código

Nesta linguagem de programação, somos obrigados a utilizar o conceito de ***indentação***, ou seja, refere-se aos espaços no início de uma linha de código para ter o código mais organizado.

Este é um bom princípio de programação, pois a organização do código permite detetar erros e oferecer uma leitura de código mais personalizada.

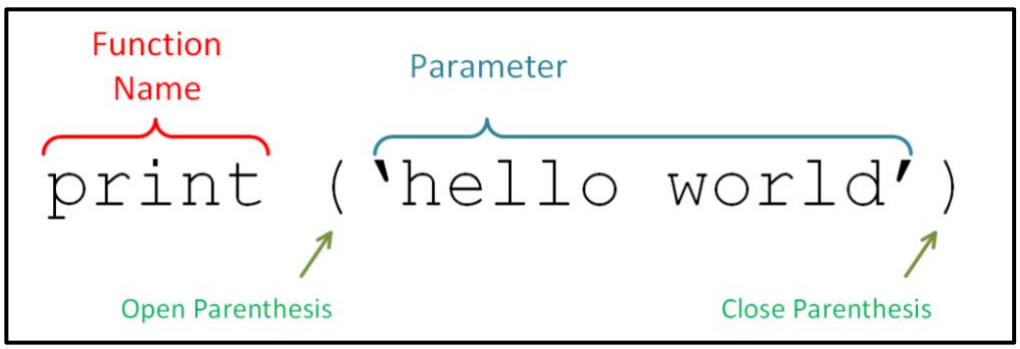


2.2 – Função Printf (escrever mensagens no ecrã)

A função ***print*** é uma das primeiras ferramentas que os iniciantes aprendem ao começar a programar.

A função ***print*** serve para exibir mensagens, valores de variáveis, resultados de cálculos e outros tipos de dados na consola durante a execução de um programa.

A função ***print*** deve utilizar parênteses (para indicar o início da instrução e fim da instrução) e dentro destes colocar o texto dentro de aspas:



Exemplo: para exibir a mensagem "Olá, Mundo!" na consola:

```
1 print("Olá, Mundo!")
2
3 # output:
4 # Olá, Mundo!
```

A função print pode aceitar praticamente qualquer tipo de dado, incluindo texto, números, resultados de operações ou qualquer outro objeto.

2.3 – Comentários no código

Para colocar ***comentários***, basta ***inicializar*** a frase com o ***símbolo #*** e escrever o texto do comentário logo a seguir:

```
#Isto é um exmeplo de um comentário
#Os comentários não são executados
print("Isto foi um comentário :) ")
```

Nota Importante: Comentários nunca são interpretados pelo código, ou seja, são descartados pela depuração.

Também podem escrever os comentários a frente do código:

```
print("Outro comentario!") #Outro exemplo de comentário
```

Mas também podemos fazer comentários de multilinhas com vários cardinais ou então com 3 aspas em cima e em baixo:

```
#Isto foi
#um comentário de
#multi-linhas :)
print("Várias linhas XD")
```

```
"""
Isto foi
um comentário de
multi-linhas :)
"""
print("Várias linhas XD")
```

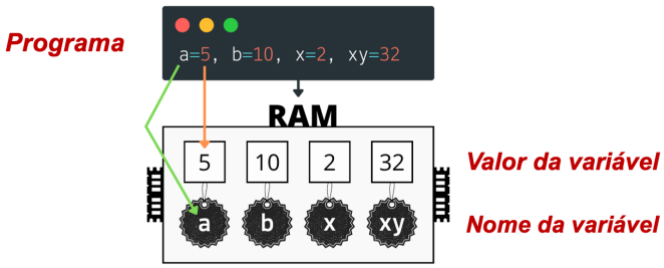
2.4 – Variáveis

Uma variável corresponde a uma posição da memória onde vamos guardar a informação, cujo conteúdo, pode ser alterado na execução de um programa.

***Nota importante*: Embora uma variável possa assumir diferentes tipos de dados, ela só pode armazenar um valor a cada instante.**

O que acontece na memória RAM?

O programa armazena as variáveis e informação na memória RAM, para que o programador não perca a informação:



Regras dos Nomes Variáveis e Constantes

O nome de uma variável/constante não pode:

- conter espaços e operadores (+, -, +, /, %) (ex: soma+dois);
- começar por um número (ex: 1numero);
- conter acentos (ex: preparação);
- ser igual a uma palavra reservada da linguagem de programação, como por exemplo:

Keywords

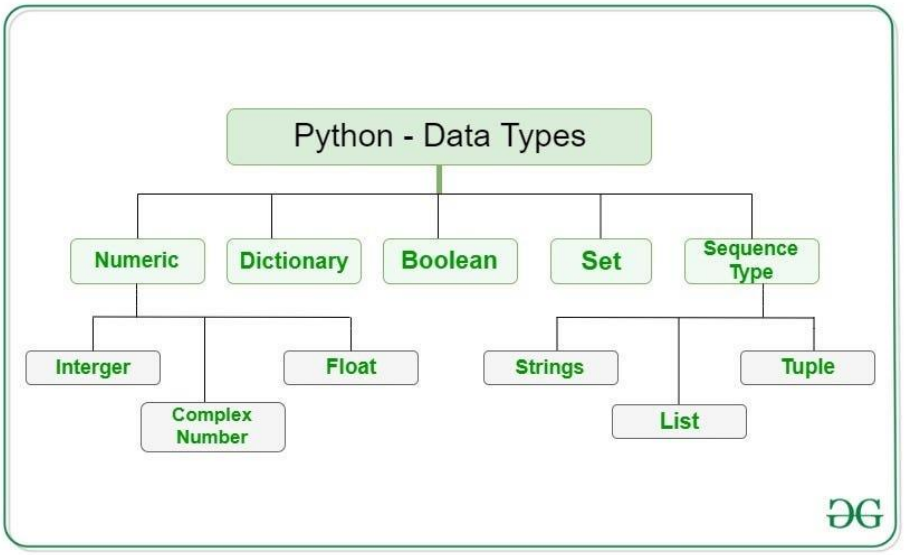
```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with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

Soft Keywords

```
match      case       _
```

Real Python

2.5 – Variáveis e Tipos de Dados



Tipos de dados elementares

- ***Numéricos (Inteiro, Decimais)***

  Existem dados que são do tipo numéricos e podem ser de vários tipos:

  int → Inteiros (Exemplo: 10, -5, 0, 5, 10...)

  Float / Double → Decimais (Exemplo: 1.5, 0, 2.5, - 4.1...)

  Exemplos: numero1 = 123 ***ou*** numero1 = 123.12

  Nota Importante: na programação é utilizado o "." como ***sinal decimal***

- ***Strings (caracter ou cadeia de carateres (texto))***

  Existem dados que são do tipo caracteres (char - de character em inglês);

  - O tipo char serve para ***armazenar UM***, e somente UM caracter. Para declarar, usamos a seguinte sintaxe:

    texto1 = 'b'

  - Se for preciso armazenar ***vários caracteres*** (texto), temos que usar os dados que são do tipo texto ou cadeias de caracteres (strings). Existem dados que são do tipo texto ou cadeias de caracteres (strings);

    texto2 = "Hoje é dia 14"

- ***Booleanos (apenas contém um de dois valores lógicos): (true (verdadeiro) ou false (falso))***

  Existem dados que são do tipo lógicos ou booleanos. São utilizados com muita frequência em algoritmia e programação. Este tipo de dados caracteriza-se por admitir de cada vez um de dois resultados: Verdadeiro(true, 1) ***ou*** Falso(false, 0)

estado = true ***ou*** estado = false

Nota importante:

O Python é *case sensitive*, o que significa que:

a variável ***nome***

é diferente

da variável ***Nome***

```
a = 4
A = "Sally"

print(a)
print(A)
```

```
4
Sally
```

2.6 – Funções Print ... mas com variáveis

Exemplo: Se desejar mostrar o conteúdo de variáveis com a função print() (seja qual o tipo de dados), use as chavetas curvas e coloque a variável a representar:

```
1  nome = "Alice"
2  idade = 30
3
4  print("Nome: " + nome)
5  print(f"Idade: {idade}")
```

2.7 – Operadores aritméticos e funções matemáticas

Funções matemáticas <u>sem</u> a necessidade da biblioteca Math:

- min() e max()
- abs()
- pow()
- round()

Funções matemáticas <u>com</u> a necessidade da biblioteca Math

- sqrt()
- Pi
- Entre outras ...
- Mais fórmulas: https://www.w3schools.com/python/module_math.asp

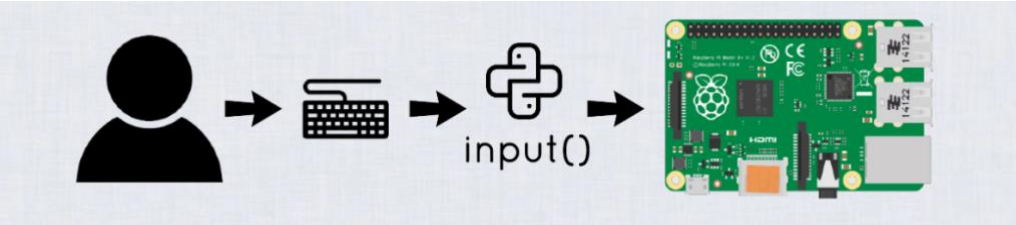
```
1  import math
2
3  #Introduzir numero com várias casas decimais
4  numero = float(input("Insira número decimal: "))
5
6  print(f"Como número inteiro: {int(numero)}")
7  print(f"Arredondado às unidades: {round(numero)}")
8  print(f"Duplo de numero: {numero * 2}")
9  print(f"Elevado ao quadrado: {numero ** 2} ou com a função pow(): {pow(numero,2)}")
10 print(f"Raiz quadrada: {math.sqrt(numero)}")
```

```
Run   Share   Command Line Arguments

Insira número decimal:
12.12
Como número inteiro: 12
Arredondado às unidades: 12
Duplo de numero: 24.24
Elevado ao quadrado: 146.8944 ou com a função pow(): 146.8944
Raiz quadrada: 3.481379037105842
```

2.8 – Função input (receber valores do teclado)

A função ***input*** serve para receber valores digitados do teclado para dentro do programa através da consola durante a execução de um programa.

Graças a isso, conseguimos com que o utilizador consiga definir os próprios dados para dentro do programa.



- Na ***versão python 2.7*** utilizavam a função ***raw_input()***
- A partir da ***versão 3.6*** usa-se a função ***input()***

Quando invocamos a função input() temos de ***colocar aspas dentro dos parenteses*** para exibir a mensagem na consola:

```
1   #criar uma variável nome para guardar o valor inserido do teclado (input)
2   nome = input("Qual é o seu nome? ")
3
4   #mostrar o que foi obtido para a variável nome
5   print(f"O seu nome é: {nome}")
```

```
Qual é o seu nome? Ana
O seu nome é: Ana
```

Outro exemplo com mais do que uma variável (utilizando o operador + para juntar texto e variáveis):

```
primeiro = input("Primeiro Nome: ")
ultimo = input("Ultimo Nome: ")

print(f"Primeiro Nome: {primeiro} e o Ultimo Nome: {ultimo}")
```

```
Primeiro Nome: Rui
Ultimo Nome: Silva
Primeiro Nome: Rui e o Ultimo Nome: Silva
```

No entanto, devemos ter cuidado com um ponto importante.

Quando escrevemos algo do teclado, a função input() ***recebe a informação no formato textual***.

Se eventualmente necessitar de pedir dados em formato numérico, temos de ***converter o texto para inteiros*** (ou decimais), ***desde que o que introduza coincide com a mesma temática***.

- Para ***converter o formato textual para números inteiros***, utilizamos a função ***int()*** ***antes*** ***de utilizar*** a função input:

int(input(".....") **)**

```
#utilizar a função int() atrás do input para converter texto em inteiro
idade = int(input('Qual a idade: '))
```

Exemplo: Após ler a idade e converter para inteiro, se desejar mostrar o conteúdo com a função print(), deve colocar a ***letra f*** seguido do conteúdo do texto dentro de aspas. Quando precisar de mostrar o valor de uma variável (seja qual o tipo de dados), use as chavetas curvas e coloque a variável a representar:

```
12 #como alternativa, mostrar outra forma de apresnetação de variáveis
13 print(f'Nome {nome} e idade {idade}')
```

Exemplo completo com as 2 formas

```
1 #função input devolve o valor no fomrato textual
2 nome = input('Insira o seu nome ')
3
4 #utilizar a função int() atrás do input para converter texto em inteiro
5 idade = int(input('Qual a idade: '))
6
7 #como alternativa, mostrar outra forma de apresnetação de variáveis
8 print(f'Nome {nome} e idade {idade}')
```

```
Qual a idade:
15
Nome: ana e idade: 15
Nome ana e idade 15
```

Exemplo de conversão de dados

Function	Conversion
int()	string,float->int
float()	string,int->float
str()	int, float, list, tuple, dictionary -> string
list()	string, tuple, dictionary, set -> list
tuple()	string, list, set ->tuple
set()	string, list, tuple -> set
complex()	int, float -> complex

2.9 – Operadores de comparação

Operador	Descrição	Exemplo	Resultado
>	Maior	5 > 3	5 é **maior que** 3?
>=	maior ou igual	5 >= 4	5 é **maior ou igual que** 4?
<	menor	2 < 9	2 **é menor que** 9?
<=	menor ou igual	9 <= 9	9 **é menor ou igual que** 9?
==	igual / comparação	4 == 4	4 **é igual a** 4?
!=	diferença	2 != 2	2 **é diferente qu**e 2?

2.10 – Operadores lógicos

Operador	Descrição	Exemplo	Resultado
&&	**E** lógico	(2 > 1 **&&** 3 > 1)	**Verdadeiro** (porque satisfez as duas condições)
||	**Ou** lógico	(2 > 1 **||** 3 < 1)	**Verdadeiro** (porque satisfez as uma das duas condições)
!	negação	**!**true	O contrário de **Verdadeiro** é **Falso**

Negação

A	~A
V	F
F	V

Operador "E"

A	B	A ∧ B
V	V	V
F	V	F
V	F	F
F	F	F

Operador "OU"

A	B	A ∨ B
V	V	V
F	V	V
V	F	V
F	F	F

2.11 – Estruturas de controlo

A parte operativa de um programa é composto por um conjunto de instruções.

Numa linguagem de programação as instruções correspondem à codificação dum algoritmo (pseudolinguagem ou fluxograma), de acordo com a sintaxe específica da linguagem.

Contudo, existe a possibilidade de utilizar estruturas de controlo para decidir ou repetir ações do código.

Tipos de Instruções de um programa:

- ***Leitura e Escrita*** (como já analisado)
- ***Atribuição*** (como já analisado)
- ***Estruturas de controlo*** (decisão ou repetição)

2.11.1 – Estruturas de Decisão

4 formas de estruturas de decisão:

- ***Estrutura de decisão simples (if)***
- ***Estrutura de decisão dupla (if ... else)***
- ***Estrutura de decisão dupla (if ... elif ... else)***
- ***Estrutura de decisão seletiva (switch)***

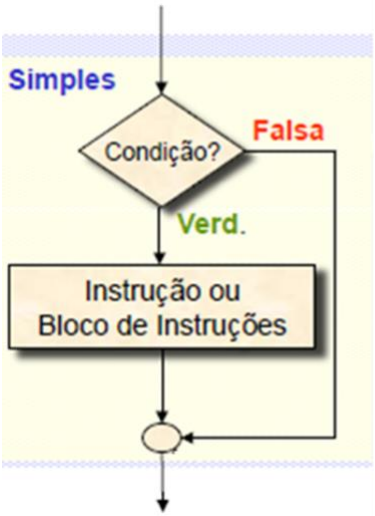
2.11.1.1 – Estrutura de decisão simples (if)

Na programação, utilizamos a ***estrutura de decisão simples (if)*** para executar um bloco de código ***mediante a condição*** seja ***verdadeira***.

Por exemplo, atribuir notas (A, B, C) com base na percentagem obtida por um aluno.

- **Se** o percentagem for superior a 90 , **atribuir** nota A
- **Se** o percentagem for superior a 70 , **atribuir** nota B
- **Se** o percentagem for superior a 50 , **atribuir** nota C

Na programação, utilizamos a ***estrutura de decisão simples (if)*** para executar um bloco de código ***mediante a condição*** é verificada é verdadeira:



Decisão Simples

SE condição **ENTÃO**
(*Condição é Verdadeira*)
Instrução ou Bloco de Instruções

```
if condição:
    # instruções ou bloco de instruções
```

Condition is True

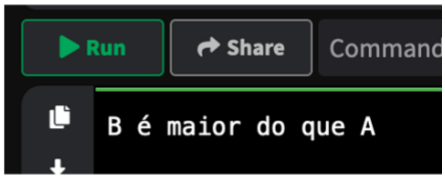
```
number = 10
if number > 0:
    # code

# code after if
```

Condition is False

```
number = -5
if number > 0:
    # code

# code after if
```

Em Python, usamos a estrutura de decisão simples (if) com a seguinte estrutura:

```
1  #declaração das variáveis com valores
2  a = 12
3  b = 15
4
5  #Se b for maior do que a
6  if b > a:
7    #se for verdade, mostra a seguinte mensagem
8    print("B é maior do que A")
```



Reparem que este ***apenas verifica se*** b é maior do que a. ***Caso isto não aconteça***, a ***condição será falsa*** e não vai mostrar a mensagem do if.

Nota importante: devem ter sempre em consideração a indentação e organização do código. Nas ***estruturas de controlo*** e ***métodos*** utilizamos esta regra, caso contrário, o Python vai dar erro ao executar o código:

Com indentação

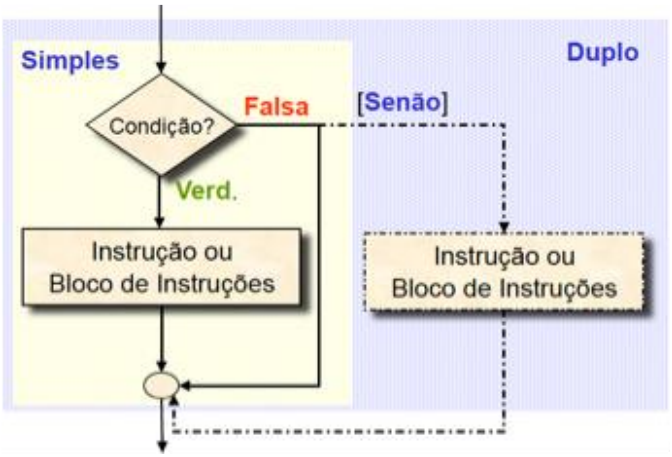
```
1  x = 1
2  total = 0
3
4  # Inicio da estrutura de decisão simples
5  if x != 0:
6      total += x
7      print(total)
8  # Fim da estrutura de decisão simples
9
10 print("Isto foi um teste.")
```

Sem indentação

```
1  x = 1
2  total = 0
3
4  # Inicio da estrutura de decisão simples
5  if x != 0:
6  total += x
7  print(total)
8  # Fim da estrutura de decisão simples
9
10 print("Isto foi um teste.")
```

2.11.1.2 – Estrutura de decisão dupla (if ... else)

Utilizamos a ***estrutura de decisão dupla (if ... else)*** para executar um bloco de código ***mediante a condição*** seja ***verdadeira*** ou ***falsa***, ou seja, ***podemos testar quando a condição é verdadeira e*** quando a ***condição é falsa:***



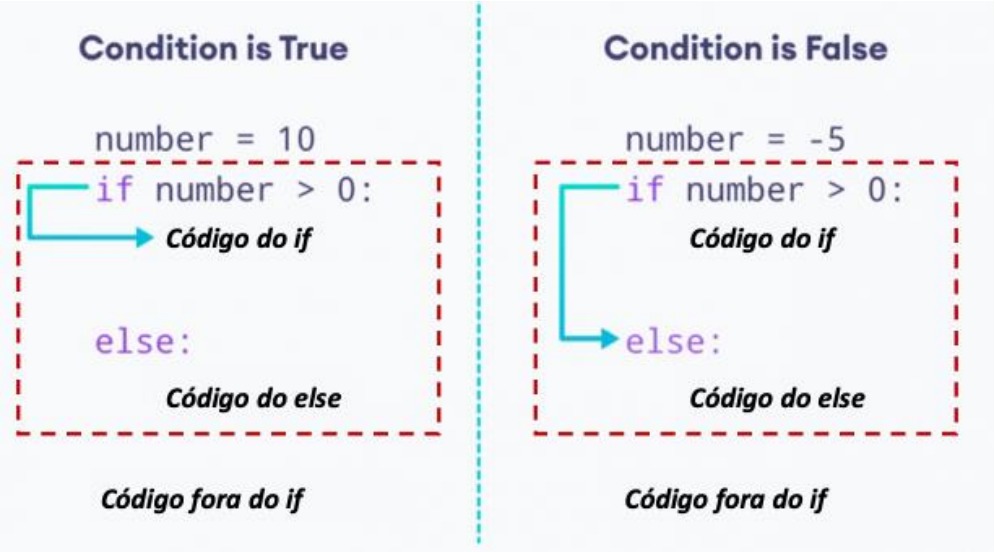
Na decisão dupla, ***ou*** corre o código do bloco do if ou do else. Nunca vai correr os dois ao mesmo tempo, senão perdia a lógica da decisão dupla. Logo:

- **Se a condição for *verdadeira***, executa o código do ***if*** e ***descarta*** o else;
- **Se a condição for *falsa***, executa o código do ***else*** e ***descarta*** o if;

Decisão Dupla

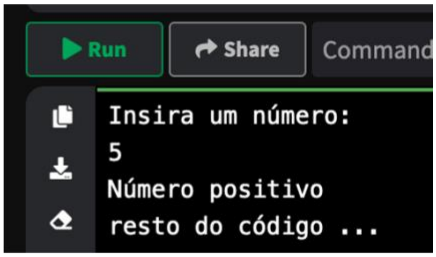
SE condição **ENTÃO**
(*Condição é Verdadeira*)
Instrução ou Bloco de Instruções
SENÃO
(*Condição é Falsa*)
Instrução ou Bloco de Instruções

```
if condição:
    # instruções do bloco if
else:
    # instruções do bloco else
```



Em Python, usamos a estrutura de decisão dupla (if ... else) com a seguinte estrutura:

```
1  number = int(input("Insira um número: "))
2
3  #Se número for maior ou igual que 0
4  if number >= 0:
5      #Se a condição for verdadeira
6      print('Número nulo ou positivo')
7  else:
8      #Se a condição for falsa
9      print('Número negativo')
10
11 #resto do código após a estrutura de decisão dupla
12 print('resto do código ...')
```



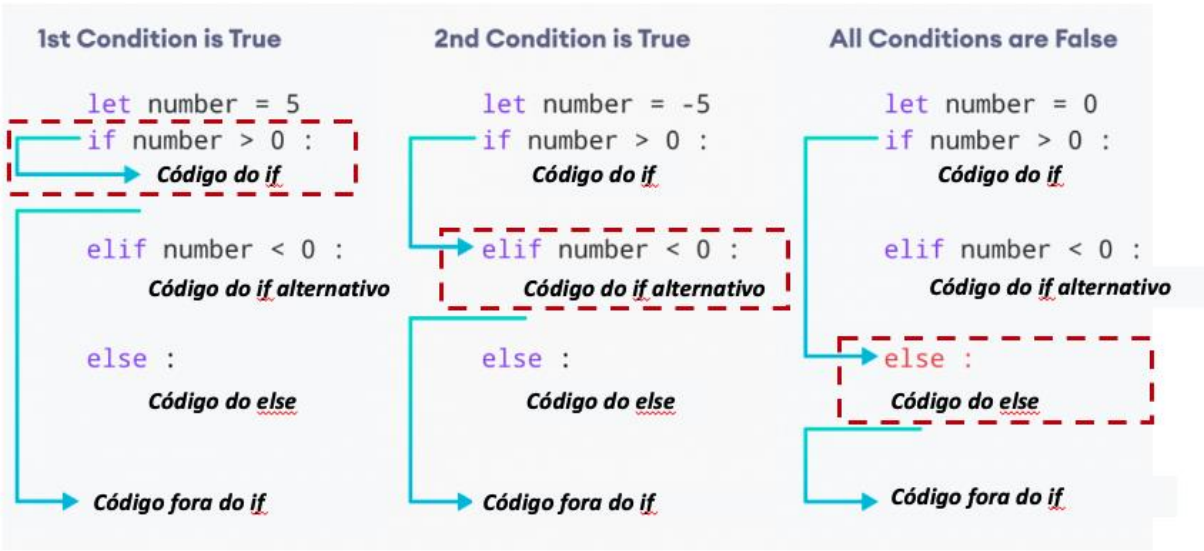
2.11.1.3 – Estrutura de decisão dupla encadeada (if ... Elif .. else)

Em Python, podemos ter ***várias instruções if encadeadas***, para executar um bloco de código entre várias alternativas.

Com base ***na análise de uma condição*** (geralmente, o resultado duma expressão booleana), podemos decidir sobre a execução ou não de determinada instrução ou bloco de instruções, ou, neste caso, optar ***entre duas ou mais instruções alternativas.***

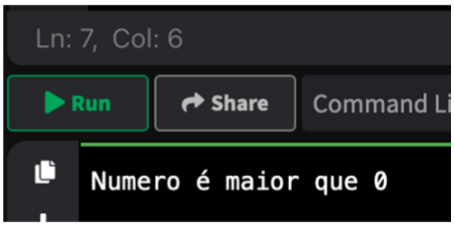
Resumindo, podemos testar para várias condições antes de chegar ao último bloco do else:

```
let number = 5
if number > 0 :
    Código do if

elif number < 0 :
    Código do if alternativo

else :
    Código do else

Código fora do if
```



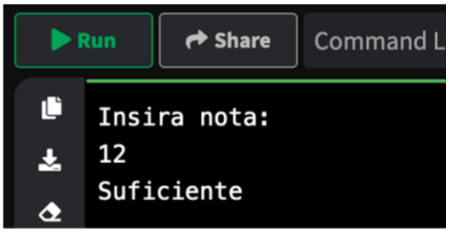
Exemplo 1:

```
1  numero = 5
2
3  if numero > 0:
4      print("Numero é maior que 0")
5  elif numero < 0:
6      print("Numero é menor que 0")
7  else:
8      print("São iguais")
```



Exemplo 2:

```
1  nota = 0
2
3  #pedir um número ao utilizador
4  nota = int(input("Insira nota: "))
5
6  if nota >= 0 and nota < 9.5:
7      print("Negativa")
8  elif nota >= 9.5 and nota < 14:
9      print("Suficiente")
10 elif nota >= 14 and nota < 18:
11     print("Bom")
12 elif nota >= 18 and nota < 20:
13     print("Excelente")
14 else:
15     print("Nota fora do intervalo entre 0 e 20")
```



2.11.1.4 – Estrutura de decisão seletiva (switch)

Em Python, usamos a estrutura de decisão seletiva quando é necessário decidir entre ***múltiplas opções possíveis***, tornando-se ideal quando queremos definir várias escolhas.

Nota importante: Até a versão 3.10, o Python não tinha a implementação da instrução switch, algo que já existe em muitas outras linguagens de programação.

Em muitas linguagens de programação, a estrutura seletiva usa o próprio nome switch.

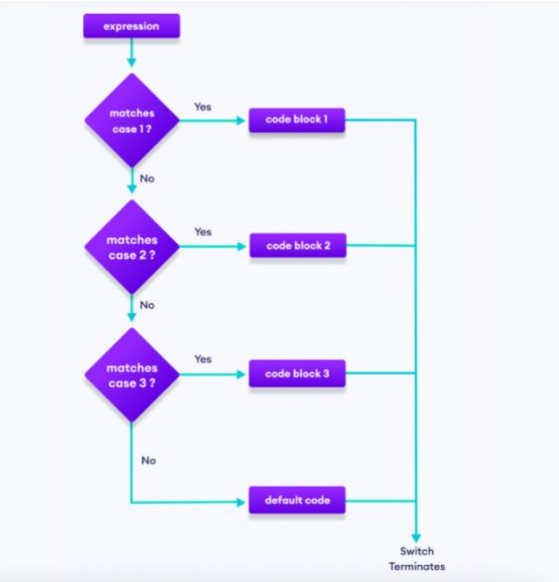
Em Python, usamos a estrutura de decisão seletiva (if ... else) com a seguinte estrutura:

```
match variável:
   case 1 : #bloco do instruções do caso 1
   case 2 : #bloco do instruções do caso 2
   ...
   case _ :  #bloco do instruções por defeito
```

```
match variável:
   case 'um' : #bloco do instruções do caso 1
   case 'dois' : #bloco do instruções do caso 2
   ...
   case _ :  #bloco do instruções por defeito
```

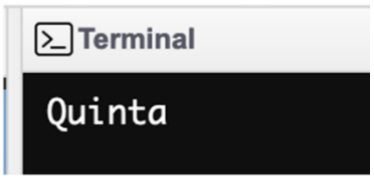
Exemplo 1 – Converter número inteiro em texto:

```
numero = 3
texto = ""

#avaliar o valor da variável
match numero:
    #colocar todos os casos a serem considerados
    case 1: texto = "Número 1"
    case 2: texto = "Número 2"
    case 3: texto = "Número 3"
    case 4: texto = "Número 4"
    case 5: texto = "Número 5"

    #caso por defeito
    case _: texto = "Fora do intervalo entre 1 e 5"

#mostrar o resultado
print(texto)
```



Exemplo 2 – Converter dia da semana (inteiro) em texto:

```
dia_semana = 4

match dia_semana:
    case 1: print("Segunda")
    case 2: print("Terça")
    case 3: print("Quarta")
    case 4: print("Quinta")
    case 5: print("Sexta")
    case 6: print("Sábado")
    case 7: print("Domingo")
    case _: print("Dia da semana inválido")
```



Exemplo 3 – Estrutura seletiva com vários casos:

```
1   mes = 5
2
3   match mes:
4     case 1 | 3 | 5 | 7 | 8 | 10 | 12: print("Mês com 31 dias")
5     case 4 | 6 | 9 | 11 : print("Mês com 30 dias")
6     case 2: print("Mês com 28 ou 29 se bisexto")
7
8     case _: print("Mês invalido")
```

```
Mês com 31 dias
```

2.11.2 – Estruturas de Repetição

As linguagens de programação também estruturas de repetição, quando o objetivo é repetir a mesma operação várias vezes.

Como tal, o python tem as seguintes estruturas de repetição:

- ***WHILE*** (estrutura de repetição manual)
- ***FOR*** (estrutura de repetição automática)

2.11.2.1 – Estrutura de repetição (while)

Nesta estrutura de repetição, o teste à condição é realizado ***antes de executar as instruções dentro estrutura de repetição*** e ***enquanto*** a condição for verdadeira, volta a executar a instrução ou bloco de instruções, ***caso contrário sai do ciclo***.

Neste ciclo, caso a condição – que é avaliada logo no início do ciclo – ***não é verificada, as ações indicadas não serão executadas nem uma vez***.

Exemplo:

- Vamos simular o processo de repetição do cálculo da soma dos números de 1 a 5;
  - Quando começamos o nosso programa, vamos ter 2 valores a ter em conta e que serão determinantes para o ciclo do programa:
    - ✓ ***Variável da soma (acumulador)***
    - ✓ ***Variável do contador***
- Vamos simular o processo de repetição do cálculo da ***soma dos números de 1 a 5***. ***Quando começamos o nosso programa***, vamos ter 2 valores a ter em conta e que serão determinantes para o ciclo do programa:
  - ***Variável da soma*** (pois será o nosso acumulador das somas)
  - ***Variável do contador*** (pois devemos dar +1 ao contador de cada vez que percorre a estrutura de repetição, ou seja, ***de cada vez que corre o ciclo da estrutura de repetição***).
  - De cada vez que percorre a estrutura de repetição (***um ciclo***), vamos ***adicionar +1 ao valor do contador de forma automática***;
  - Uma estrutura de repetição deve ser ***repetida***, ***tantas vezes quantas necessárias***, até que ***chegue a condição de saída*** da estrutura de repetição;

```python
# variaveis para guardar o acumulador da soma e contador
soma = 0
contador = 1

#enquanto o contador for inferior ou igual a 10
while contador <= 5:
    #substitui o valor da soma antiga pelo o novo cálculo
    soma = soma + contador
    print(f"Soma do ciclo {contador}: " + str(soma))

    #incrementa +1 ao contador de cada vez que repete o while
    contador = contador + 1
```

Vamos simular o processo de repetição do cálculo da soma dos números de 1 a 5:

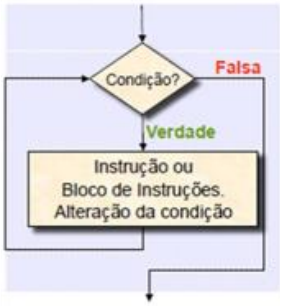
- ***ENQUANTO*** contador <= 5 ***FAZ***
  - ***soma = soma + contador;***
  - ***contador = contador + 1***
- ***FIM PARA***

(voltar ao inicio do WHILE)

Ciclos	Valor do Contador	Valor da Soma
Valores iniciais	1	0
1º Ciclo	2	1
2º Ciclo	3	3
3º Ciclo	4	6
4º Ciclo	5	10
5º Ciclo	6	15

2.11.2.2 – Estrutura de repetição automática (FOR) e a função range (intervalo)

Esta estrutura de repetição permite a repetição duma determinada instrução ou bloco de instruções, ***durante um número de vezes predefinido (sequência)***, e de forma automática. Cada repetição condicional é designada por ***Ciclo***.




```python
for valor in sequencia:
    # bloco de instruções a repetir
```

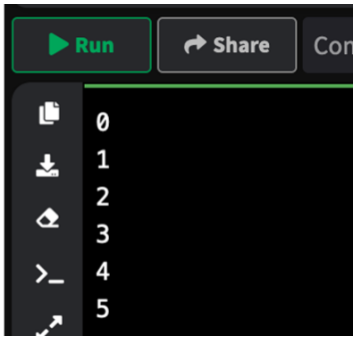
Esta estrutura de repetição permite percorrer coleções de dados, indicar um intervalo de valores, entre outras funcionalidades.

Podemos combinar funções com as repetições, tal como, uma das funções mais utilizadas, a ***função range()***.

A ***função range()*** permite gerar uma sequência de números automática ou de forma manual (indicando o valor mínimo, máximo e quanto é que salta de ciclo em ciclo).

Por padrão, a sequência da ***função range() começa em 0***, ***incrementa +1 a cada ciclo de repetição*** e ***para antes do número especificado*** nos parênteses da função. Exemplo:

```
1  # criar a sequência de 0 até 5 (contar a partir de 0)
2  lista_numeros = range(6)
3
4  # percorrer a lista dos numeros
5  for valor in lista_numeros:
6      print(valor)
```



A estrutura de repetição ***termina*** quando ***chega ao último item da sequência***.

Vejamos o que foi por detrás:

Interação	Valor de `valor`	`print(valor)`	Último valor da sequência?
1	0	Prints 0	Não
2	1	Prints 1	Não
3	2	Prints 2	Não
4	3	Prints 3	Não
5	4	Prints 4	Não
6	5	Prints 5	Sim Terminou a repetição do FOR

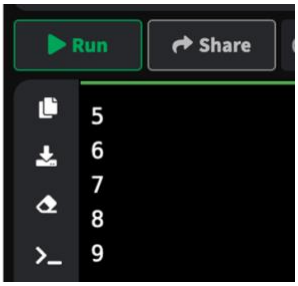
Mas, também podem indicar um intervalo de valores dentro da função range(). Como tal, em vez de indicar apenas o máximo, pode colocar 3 parâmetros:

range (mínimo, máximo, salto)

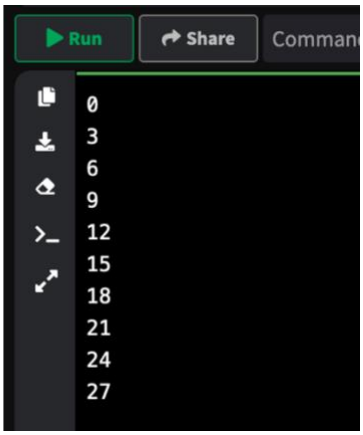
- ***Mínimo***: qual o ***valor mínimo*** a considerar na sequência;

- ***Máximo***: qual o ***valor máximo*** a considerar na sequência;
- ***Salto***: quantos ***saltos deve realizar a cada ciclo*** de repetição;

```python
# criar a sequência de 5 até 10 (com saltos de 1 em 1)
lista_numeros = range(5, 10, 1)

# percorrer a lista dos numeros
for valor in lista_numeros:
    print(valor)
```




```python
# criar a sequência de 0 até 30 (com saltos de 3 em 3)
lista_numeros = range(0, 30, 3)

# percorrer a lista dos numeros
for valor in lista_numeros:
    print(valor)
```



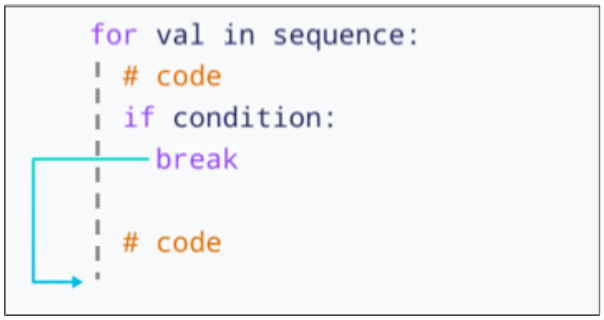
Somar todos os números de 1 até 5 (inclusive)

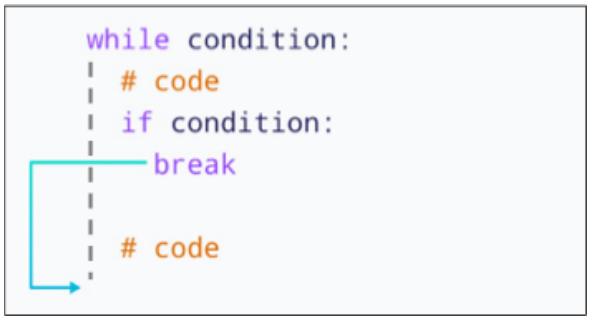
```python
#declaração da variável para guardar a soma
soma = 0

#criar sequência de números de 1 até 5 (inclusive)
numeros = range(1,6,1)

#percorrer e somar todos os números da sequência
for valor in numeros:
    soma = soma + valor

#mostrar o resultado final
print(f"Soma de 1 até 5: {soma}")
```

2.11.3 – Condições break e continue

Na programação, as instruções ***break*** e ***continue*** são utilizadas para alterar o fluxo das estruturas de repetição, ou seja:

- ***break*** – sai da estrutura de controlo por completo;
- ***continue*** - salta o ciclo atual e prossegue para o próximo ciclo;



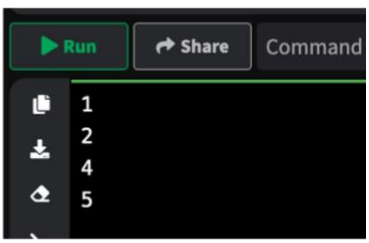


Exemplo 1: Percorrer sequencia de 1 até 5 e quebrar a estrutura se o contador entrar o número 3

```
1  #percorrer sequência de 1 até 5
2  for i in range(6):
3      #se o número que percorre for igual a 3
4      if i == 3:
5          #quebra a estrutura que sestá a percorrer
6          break
7
8      #mostra o valor que está a percorrer
9      print(i)
10
11 print(f"Parou no número: {i-1}")
```

Exemplo 2: Percorrer sequencia de 1 até 5 e salta da estrutura se o contador entrar o número 3

```
1  #percorrer sequência de 1 até 5
2  for i in range(6):
3      #se o número que percorre for igual a 3
4      if i == 3:
5          #salta o ciclo atual e prossegue para o próximo ciclo
6          continue
7
8      #mostra o valor que está a percorrer
9      print(i)
```



2.12 – Listas (lists)

Quando queremos que uma variável, tenha mais do que uma informação, utilizamos estruturas de dados.

Embora que existam várias estruturas, na maioria das linguagens de programação, uma das mais famosas estruturas é a utilização dos vetores (arrays).

Em python, vamos utilizar o conceito de listas, também conhecido como lists.

As listas servem para permitem agrupar e armazenar vários itens na mesma variável/objeto.

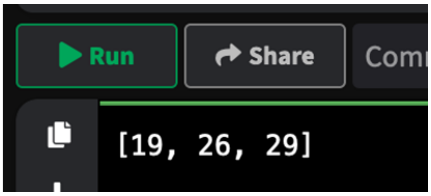
Por exemplo, podemos usar uma lista em Python para armazenar uma lista de itens (números, nomes, músicas, etc...), para que seja possível adicionar, remover, atualizar e listar facilmente as músicas conforme seja necessário.

2.12.1 – Criar Lista

Para criar uma lista e colocar elementos, temos de dar o nome a lista (tal como as variáveis), e de seguida, devemos colocar os conteúdos dentro de chavetas retas com os itens separados por virgulas:

nome_variavel = [valor_1, valor_2, valor_3]

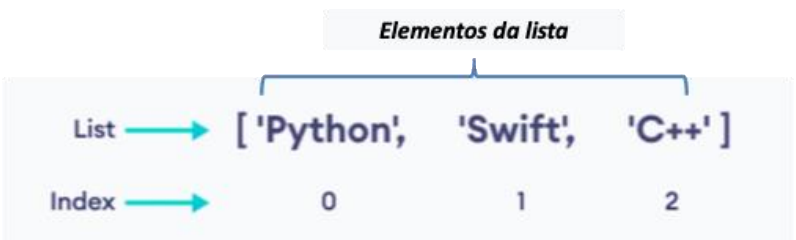
```
1  #criar lista com 3 elementos inteiros
2  numeros = [19, 26, 29]
3  print(numeros)
4
```



2.12.2 – Aceder aos elementos da lista

Quando criam uma lista de valores (podem ser numéricos, textuais, booleanos), cada elemento da lista está associado a um identificador, mais conhecido como ***índice***.

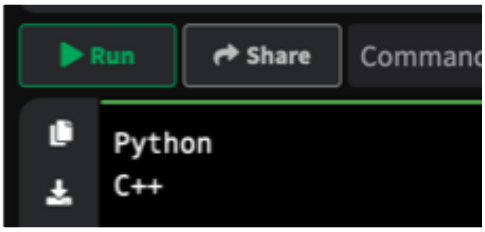
O índice do primeiro item começa sempre a contar a partir 0 e assim em diante:



Usamos os números dos índices para aceder os itens da lista.

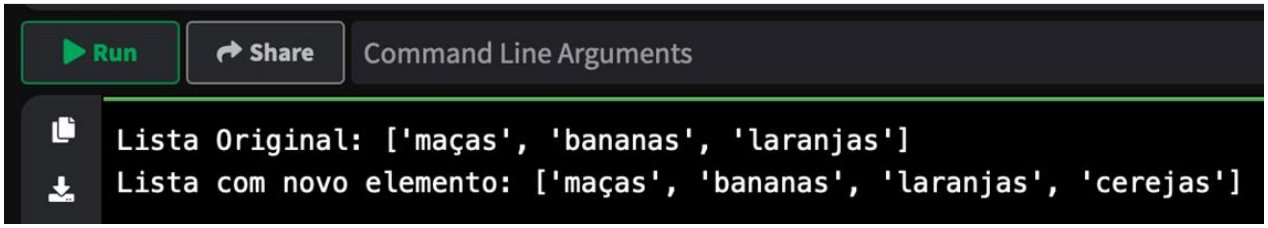
Exemplos:

```
languages = ['Python', 'Swift', 'C++']

# Aceder ao 1º elemento
print(languages[0])   # Python

# Aceder ao 3º elemento
print(languages[2])   # C++
```



Para adicionar elementos utilizamos o método ***append()***. Este por sua vez adiciona os novos elementos no final da lista:

```
frutas = ['maças', 'bananas', 'laranjas']
print('Lista Original:', frutas)

# adicionar novo elemento na lista
frutas.append('cerejas')

print('Lista com novo elemento:', frutas)
```



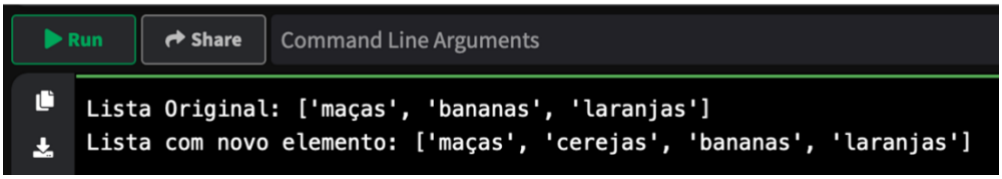
2.12.3 – Adicionar elementos (com insert)

Também podemos usar o método ***insert()*** quando desejam inserir um elemento num índice especifico, fazendo com que os outros elementos a partir desse índice andem para a frente.

Neste método temos de indicar o índice que desejam colocar o conteúdo e o valor:

insert (indice, valor)

```
1 frutas = ['maças', 'bananas', 'laranjas']
2 print('Lista Original:', frutas)
3
4 # adicionar novo elemento na lista
5 frutas.insert(1, 'cerejas')
6
7 print('Lista com novo elemento:', frutas)
```



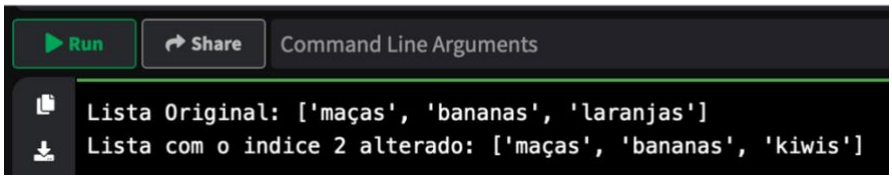
2.12.4 – Alterar elementos

Podemos alterar os itens de uma lista atribuindo novos valores. Como tal, devemos:

- ***invocar o nome da lista;***
- ***Indicar o índice a modificar;***
- ***Usar o operador = e indicar o novo valor;***

Exemplo: frutas[2] = kiwis

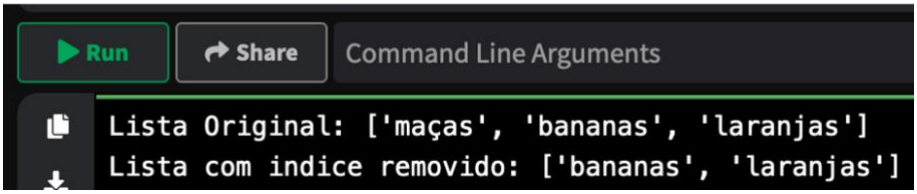
```
1 frutas = ['maças', 'bananas', 'laranjas']
2 print('Lista Original:', frutas)
3
4 # alterar indice 2 na lista
5 frutas[2] = "kiwis"
6
7 print('Lista com o indice 2 alterado:', frutas)
```



2.12.5 – Remover elementos

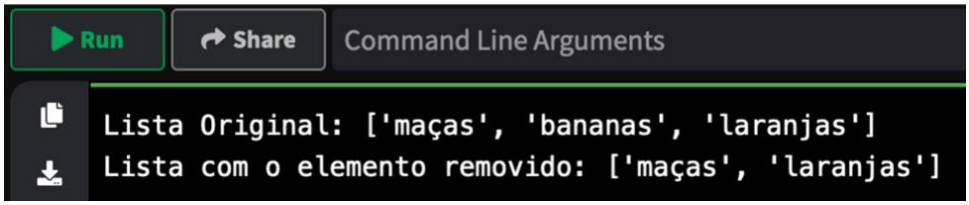
Podemos remover um índice da lista usando o método ***remove()*** e dentro dos parênteses, indicamos o valor a remover:

```
1 frutas = ['maças', 'bananas', 'laranjas']
2 print('Lista Original:', frutas)
3
4 # remover o indice 1 da lista
5 frutas.remove("maças")
6
7 print('Lista com indice removido:', frutas)
```



Também podemos remover elemento da lista usando o método ***pop()*** e dentro dos parênteses, indicamos o índice a remover:

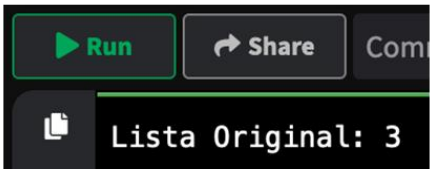
```
1 frutas = ['maças', 'bananas', 'laranjas']
2 print('Lista Original:', frutas)
3
4 # remover o indice 1 da lista
5 frutas.pop(1)
6
7 print('Lista com o elemento removido:', frutas)
```



2.12.6 – Tamanho de elementos numa lista

Podemos saber quantos elementos existem numa lista usando a função ***len()*** do qual retorna o número de elementos:

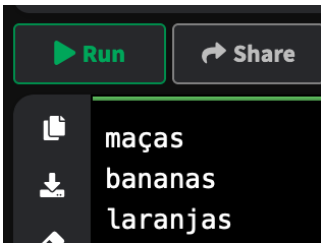
```
1  frutas = ['maças', 'bananas', 'laranjas']
2  print('Lista Original:', len(frutas))
```



2.12.7 – Percorrer listas (forma direta)

Para percorrer cada elemento de uma lista, podemos utilizar a estrutura de repetição for e com um print mostrar cada valor que está a percorrer:

```
1  frutas = ['maças', 'bananas', 'laranjas']
2
3  for valor in frutas:
4      print(valor)
```



2.12.8 – Percorrer listas (forma detalhada)

Mas, se precisar de saber qual o índice, também pode pedir ao python para ***descobrir o índice pelo elemento*** que está a percorrer naquele momento.

Para tal, vamos ter de utilizar a função ***index()***:

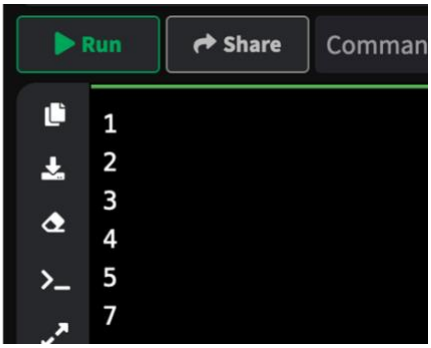
nome_lista.***index***(valor)

```
1  frutas = ['maças', 'bananas', 'laranjas']
2
3  for valor in frutas:
4      print(f"Indice {frutas.index(valor)} : {valor}")
```

2.12.9 – Ordenar listas (forma crescente)

Mas, se precisar de ***ordenar listas por ordem crescente***, podemos usar o método ***sort().***

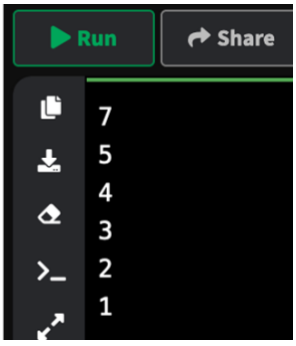
```
numeros = [3,7,1,5,2,4]

#método sort para ordenar os numeros inteiros
numeros.sort()

for valor in numeros:
    print(valor)
```



Opcionalmente, podemos ordenar as listas de forma personalizada. O método sort() pode levar dois argumentos (opcionais):

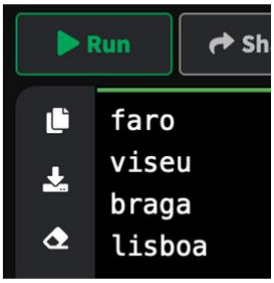
- ***reverse*** – por defeito falso (false). Se indicar o valor verdadeiro (true), este ordena a lista de forma decrescente;

```
numeros = [3,7,1,5,2,4]

#método sort para ordenar os numeros inteiros
#pedir para colocar por ordem decrescente
numeros.sort(reverse=True)

for valor in numeros:
    print(valor)
```

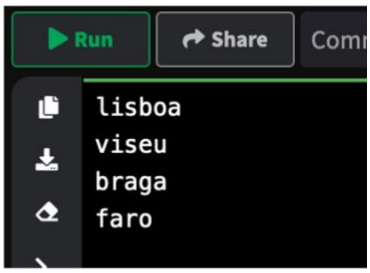


- ***key*** – pode ser invocada uma função para reorganização da informação;

```
cidades = ["viseu","braga","lisboa","faro"]

#método sort para ordenar as palavras
#pedir para colocar por ordem crescente
#pelo tamanho das letras (len)
cidades.sort(key=len)

for valor in cidades:
    print(valor)
```



```python
cidades = ["viseu","braga","lisboa","faro"]

#método sort para ordenar as palavras
#pedir para colocar por ordem decrescente
#pelo tamanho das letras (len)
cidades.sort(reverse=True, key=len)

for valor in cidades:
    print(valor)
```



2.13 – Subprogramas e funções

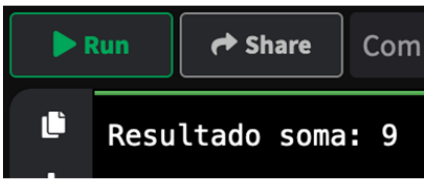
Um subprograma é um ***bloco de código*** que executa ***uma tarefa específica***.

O objetivo principal é:

- isolar o código que está repetido várias vezes;
- reutilizar as mesmas funcionalidades quantas vezes forem necessárias;

Suponha o seguinte exemplo para somar dois números inteiros

```
1  num1 = 4
2  num2 = 5
3
4  soma = num1 + num2
5  print(f"Resultado soma: {soma}")
```



Até agora nada de novo ☺

Mas, imaginem que vamos construir um programa para somar 2 números ***por 3 vezes.*** Vamos entrar na repetição de código para o mesmo objetivo:

```
1  num1 = 4
2  num2 = 5
3
4  soma = num1 + num2
5  print(f"Resultado soma: {soma}")
6
7  num1 = 2
8  num2 = 7
9
10 soma = num1 + num2
11 print(f"Resultado soma: {soma}")
12
13 num1 = 1
14 num2 = 3
15
16 soma = num1 + num2
17 print(f"Resultado soma: {soma}")
```

Será que não é possível eliminar esta repetição e isolar o código apenas num bloco genérico?

2.13.1 – Criar função (isolamento de código)

Mas, se precisar de saber qual o índice, também pode pedir ao python para ***descobrir o índice pelo elemento*** que está a percorrer naquele momento.

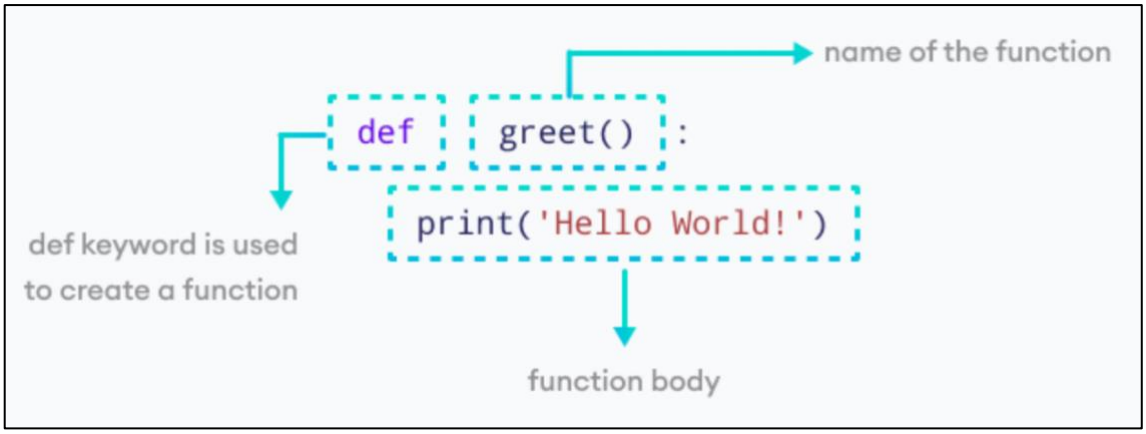
Para tal, vamos ter de utilizar a função ***index()***:

Como referido anteriormente, mm subprograma/função é um ***bloco de código*** que executa ***uma tarefa específica***.

O objetivo principal é:

- isolar o código que está repetido várias vezes;
- reutilizar as mesmas funcionalidades quantas vezes forem necessárias;

Para criar o corpo de uma função devemos construir a seguinte estrutura:



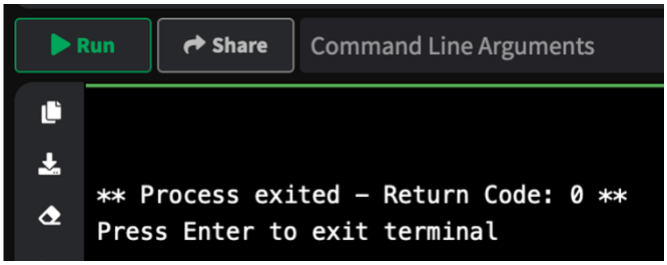
Neste exemplo, criamos uma ***função*** chamada ***greet()*** que imprime a mensagem "Olá mundo!"

Nota importante: ter cuidado com a indentação do código dentro de uma função, pois deve respeitar a mesma regra de avanço para o programa perceber o que pertence a função:

2.13.2 – Invocar função

Uma vez criada a função (vamos utilizar o exemplo anterior do greet()) e executarmos o código, não vamos obter nada.

```
1  #função para mostrar uma mensagem
2  def greet():
3      print("Ola mundo!")
```

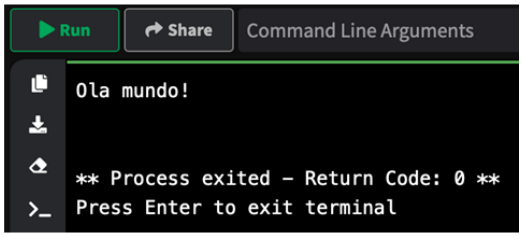


***Isto é super normal*, pois ainda não invocamos a função no código!!!!!**

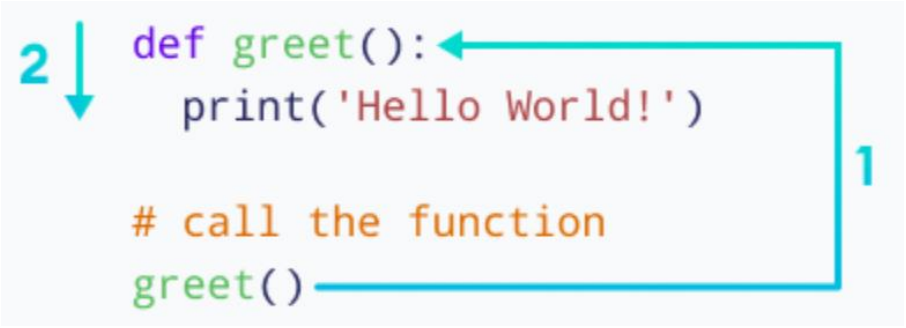
Criar uma função não significa que estamos a executar o código contido na função. Isso significa que o código está lá para usar ***se eventualmente for necessário***.

Para usar essa função, precisamos chamar a mesma:

```
1  #função para mostrar uma mensagem
2  def greet():
3      print("Ola mundo!")
4
5  #chamar a função greet
6  greet()
```

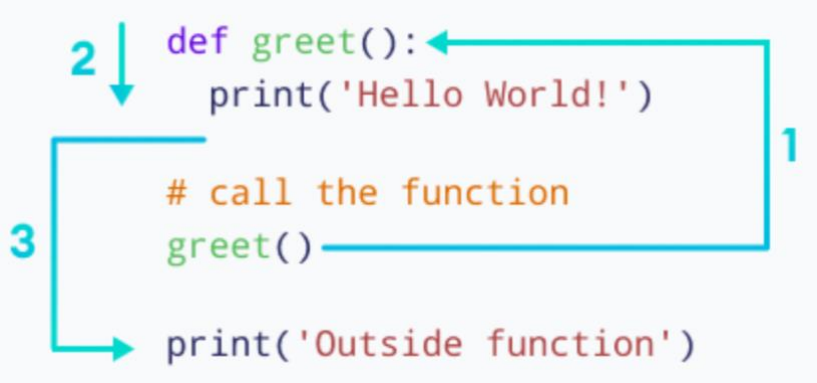


Vejamos o que acontece por detrás do código:



Se tiver mais código para além da invocação:

1. Quando a função greet() é chamada, o programa transfere o código para a função.
2. Todo o código dentro da função é executado.
3. O programa salta para a próxima instrução após a chamada de função.



2.13.3 – Argumentos das funções

Até agora vimos como criar uma função com a estrutura mais básica. Mas as funções têm mais recursos úteis para a programação.

Quando declaramos o nome da função apenas indicamos o nome e parênteses a vazio.

Dentro dos parênteses, podemos passar argumentos, que nada mais, são entradas de dados para dentro da função, quando esta é invocada.

Vejamos o seguinte exemplo:

Neste código, passamos o texto “Rita” como um argumento para a função mostra_nome.

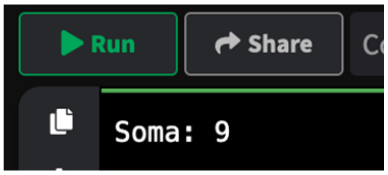
```
1 #função para mostrar o nome de uma pessoa
2 def mostra_nome(nome):
3     print("Ola " + nome)
4
5 # invocar a função e passar 1 argumento
6 mostra_nome("Rita")
```

Mas, se for necessário, ***podemos passar mais argumentos*** (se necessário), por forma a tornar a função reutilizável e dinâmica.

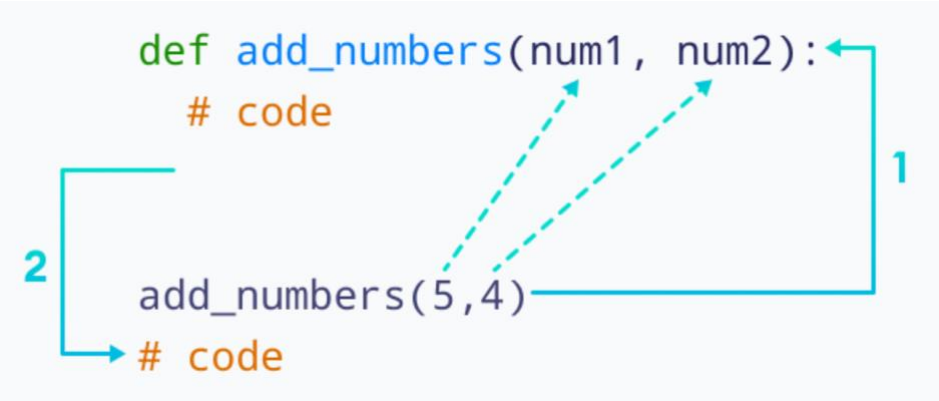
No próximo exemplo, vamos somar dois números:

- ***Na linha 7***, estamos a invocar a função e passar 2 valores (separados por virgulas);
- ***Na linha 3***, criamos a função recebe 2 parâmetros (separados por virgulas);

```
1  #função para mostrar o nome e idade de uma pessoa
2  def soma_numeros(num1, num2):
3      soma = num1 + num2
4      print(f"Soma: {soma}")
5
6  # invocar a função e passar 2 argumentos
7  soma_numeros(5, 4)
```



Parâmetros são as variáveis listadas dentro dos parênteses na definição da função. Neste caso, ***os valores são transmitidos aos parâmetros quando a função é executada (separados por virgulas, para o programa perceber a separação dos valores).***



Outro exemplo: ***Levar nome e idade de uma pessoa***

```
1  #função para mostrar o nome e idade de uma pessoa
2  def mostra_nome(nome, idade):
3      print(f"Ola {nome} e tenho {idade}")
4
5  # invocar a função e passar 2 argumentos
6  mostra_nome("Rita", 27)
```

2.13.4 – Retorno de informação das funções

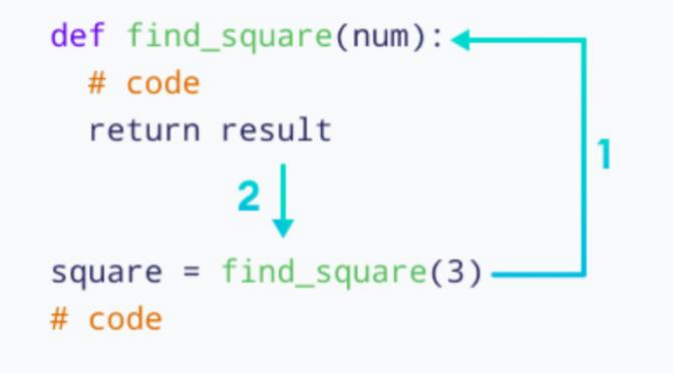
Até agora todo o código feito nas funções apenas fica contido no mesmo.

Agora, imagine que pretende devolver o que foi processado da função para o programa principal?

Para conseguir fazer isso, podemos utilizar a palavra ***return*** no ***final do código de cada função*** para ***devolver os valores desejados***.

Nota importante: depois de invocar a palavra return, não pode escrever mais código, ou seja, qualquer código após o return não é executado.

Funcionamento por detrás por código:



Exemplo: Queremos calcular a área de um terreno e no final desse cálculo, queremos ***retornar*** o resultado para o programa principal.

Como tal, vamos fazer o programa com os seguintes passos:

1. ***Criar a função calcula_area*** que recebe ***2 parâmetros de entrada*** (comprimento e largura);
2. ***Criar o código da função*** e no final ***retornar a área do terreno***;

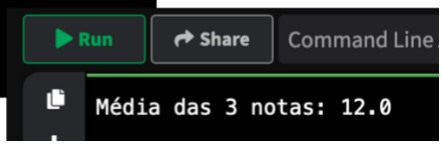
```
#Criar a função calcula_area que recebe 2 parametros
def calcula_area(comprimento, largura):
    #calcula a area do terreno
    area = comprimento * largura

    #no final do todo o código invocamos a palavra reservada
    #return para devolver o resultado de volta ao programa principal
    return area
```

3. ***Passar o resultado para uma variável*** depois de executar a função;

```python
#Criar a função calcula_area que recebe 2 parametros
def calcula_area(comprimento, largura):
    #calcula a area do terreno
    area = comprimento * largura

    #no final do todo o código invocamos a palavra reservada
    #return para devolver o resultado de volta ao programa principal
    return area

#Programa principal
res = calcula_area(5, 7)
print(f"A area do terreno é: {res}")
```

Exemplo do cálculo da média com 3 notas

```python
#Criar a calcula_media para devolver a media das notas
def calcula_media(nota1, nota2, nota3):
    #calcula a media das 3 notas
    media = (nota1 + nota2 + nota3) / 3

    #no final do todo o código invocamos a palavra reservada
    #return para devolver o resultado de volta ao programa principal
    return media

#Programa principal
res = calcula_media(13,7,16)
print(f"Média das 3 notas: {res}")
```



2.14 – Classes

Uma classe permite definir um objeto com base nos estados (os campos ou variáveis) e respetivas ações/métodos que definem o seu comportamento (funções).

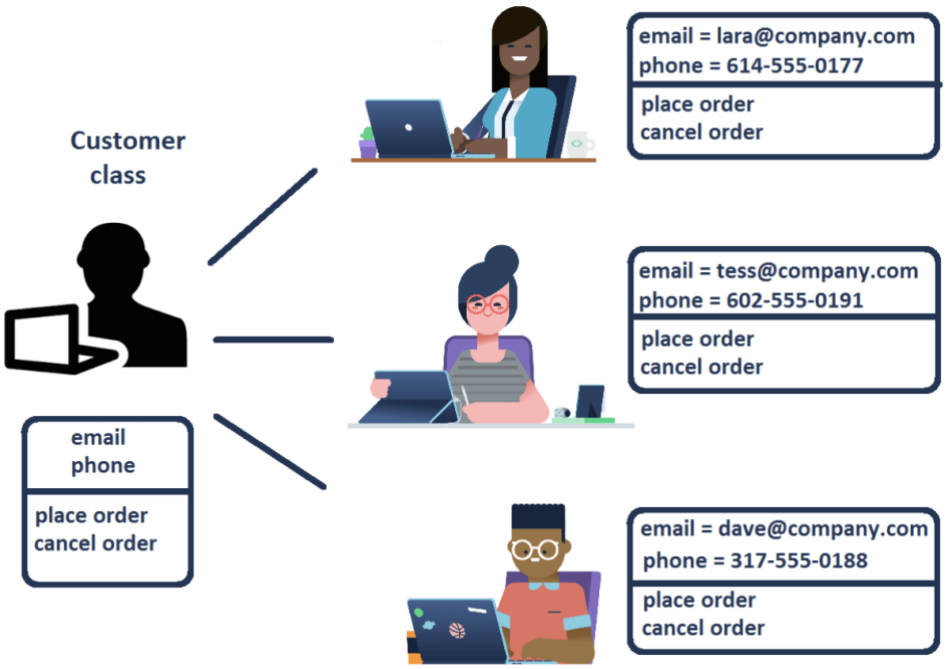
Ou seja, uma classe permite a criação de objetos com determinadas estruturas e comportamentos específicos.

Podemos pensar numa classe como um esboço (estrutura) de uma pessoa.

A classe contém todos os detalhes sobre as informações da pessoa (definidos pelo programador), como os estados e comportamentos.

Com base nessas descrições, construímos a estrutura de representa a estrutura da pessoa.

A partir da classe, podemos criar as informações de várias pessoas (com dados diferentes), com base na classe pessoa, do qual, compartilham a mesma estrutura entre ambas.



2.14.1 – Criação de classes

1. ***Uma classe tem sempre um nome***, onde a primeira letra deve sempre começar ***por maiúscula***;

```
# 1 - declaração do nome da classe
class Pessoa:
```

2. ***Fazer a implementação do construtor de inicialização*** (responsável por receber os dados para contruir o objeto). Este ***é o primeiro "método"*** que se ***constrói dentro da classe e*** que será executado quando ***um novo objeto é instanciado***.
   Dentro do construtor, devemos indicar ***os atributos*** da estrutura da classe.

   Os atributos são divididos em duas partes:

   1. ***Variáveis de instância***: definidos dentro de métodos e pertencem apenas ao objeto criado. No exemplo da classe Pessoa, id, nome e idade são ***atributos de instância*** porque são definidos dentro do método __init__ e pertencem a cada objeto com base na classe Pessoa.

```
def __init__(self, id, nome, idade):
```

   2. ***Variáveis da classe***: são as variáveis compartilhadas entre todas as instâncias da classe e podem ser acedidos diretamente pela classe ou por qualquer uma de suas instâncias. Colocamos a ***palavra reservada self*** atrás do nome para indicar que é uma variável de classe e não confundir com a variável que está declarada nos parenteses do método __init__):

```
self.ID = id
self.Nome = nome
self.Idade = idade
```

Metodo __init__ completo:

```
# 2 - Construtor de inicialização e variáveis de instância
def __init__(self, id, nome, idade):
    self.ID = id
    self.Nome = nome
    self.Idade = idade
```

4. ***Por último, criamos os métodos da classe*** que são responsáveis ***pelo comportamento das instâncias/objetos quando recebem mensagens provenientes de alguma parte do programa***;

Exemplo 1: Criar o método para mostrar os dados os dados do objeto

```
# 3 - Método para mostrar os dados das variáveis do objeto
def MostraDadosPessoa(self):
    print(f"ID: {self.ID} ,Nome: {self.Nome}, Idade: {self.Idade}")
```

Exemplo 2: Verificar se a idade é maior de 18 anos

```
def VerificaIdade(self):
    if(self.Idade >= 18):
        print("É maior ou igual que 18 anos.")
    else:
        print("É menor de idade.")
```

Código da classe completo (com a combinação dos 4 passos):

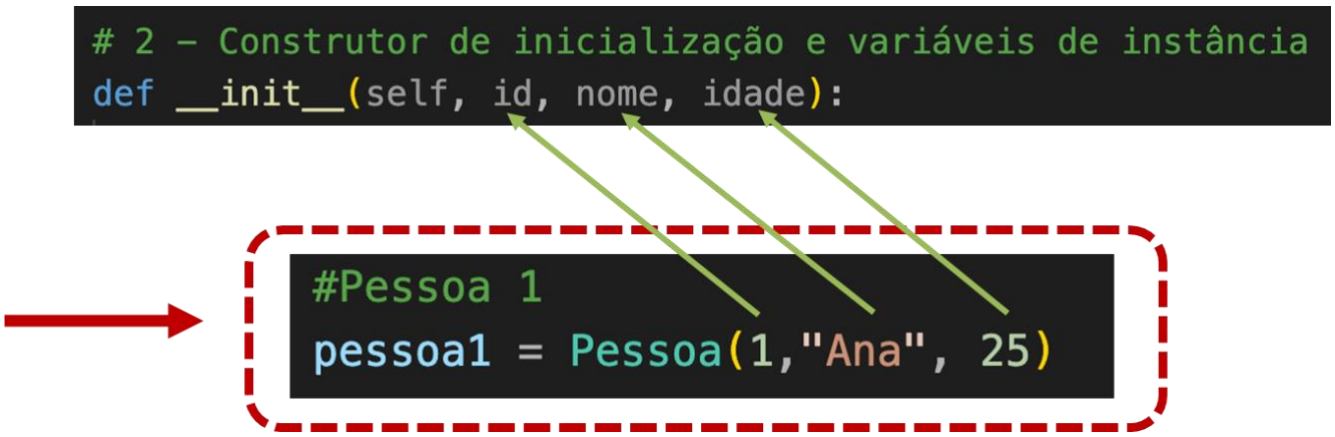
```
1   # 1 - Declaração do nome da classe
2   class Pessoa:
3
4       # 2 - Construtor de inicialização e variáveis de instância
5       def __init__(self, id, nome, idade):
6           self.ID = id
7           self.Nome = nome
8           self.Idade = idade
9
10      # 3 - Método para mostrar os dados das variáveis do objeto
11      def MostraDadosPessoa(self):
12          print(f"ID: {self.ID} ,Nome: {self.Nome}, Idade: {self.Idade}")
13
14      def VerificaIdade(self):
15          if(self.Idade >= 18):
16              print("É maior ou igual que 18 anos.")
17          else:
18              print("É menor de idade.")
```

2.14.2 – Utilização das classes

Depois de criada a nossa classe, vamos utilizar a mesma. Não esqueçam de que os ***objetos*** são ***instâncias de uma classe***.

Para poder utilizar uma classe, temos de seguir os seguintes passos:

Passo 1 - Criar uma variável e invocar a classe pretendida para criar um novo objeto (instância) da classe ***Pessoa*** e passar os valores pedidos no construtor de inicialização __init__***:***



Passo 2 - A classe recebe os novos dados do objeto e de seguida, e as variáveis da classe passam a informação para todos os métodos. Neste exemplo, vamos invocar o método para mostrar os dados. Para tal, vamos escrever a sequência:

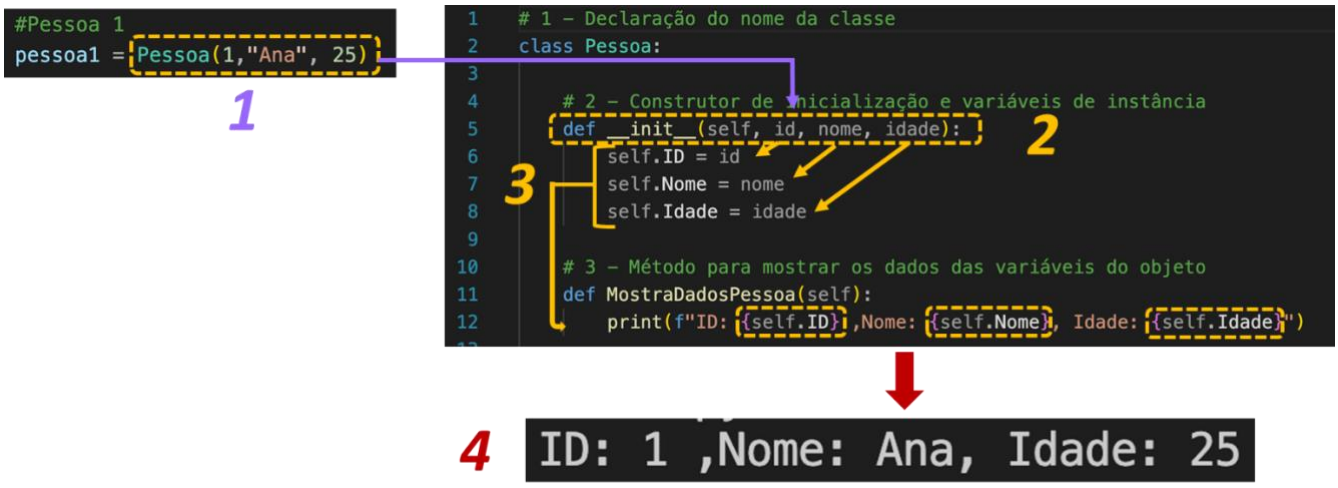
- o nome do objeto criado;
- Colocar . (ponto final) para aceder aos valores dentro do objeto;
- Chamar o método definido na classe com o nome ***MostraDadosPessoa();***

```
#Pessoa 1
pessoa1 = Pessoa(1,"Ana", 25)
pessoa1.MostraDadosPessoa()
```

Este por sua vez, vai chamar o método em causa, que neste caso, vai mostrar a mensagem com os dados das variáveis da classe que forma passados:

```
ID: 1 ,Nome: Ana, Idade: 25
```

Como funciona por detrás do código:

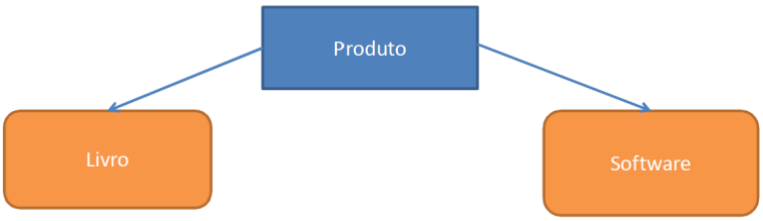


Passo 3 – Podem ***criar outros métodos na classe para outras finalidades***. Por exemplo, usar a variável de classe idade e criar o método para verificar se a idade é maior ou igual a 18 anos:

2.15 – Heranças

A herança é um dos pilares fundamentais da programação orientada a objetos (POO), do qual, permite com que uma subclasse (chamada de classe filha) derive atributos e métodos de uma superclasse (chamada de classe pai).

Esse recurso é essencial para a reutilização do código e simplifica a manutenção, facilitando a criação de programas eficientes.



No exemplo anterior, existe uma superclasse "Produto" que vai servir de base à construção de diferentes classes de produtos;

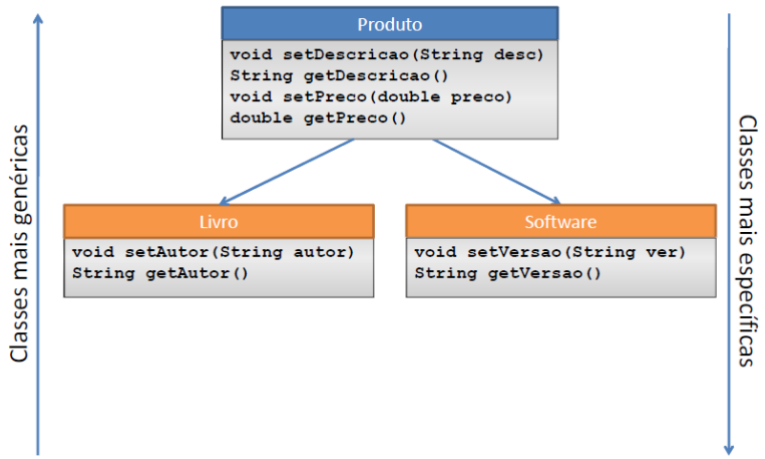
- As classes **Livro** e **Software** têm como ponto de partida a classe "**Produto**".
- As subclasses **Livro** e **Software** herdam todos os métodos públicos da classe "**Produto**";
- A subclasse **Livro** e **Software** definem **métodos específicos de cada classe**, que são únicos para **cada classe**.

Imaginem que está a criar uma aplicação com várias funções de utilizador, como por exemplo, alunos, professores e administradores.

Essas funções compartilham atributos comuns, como o nome e ID, mas também exigem funcionalidades que são exclusivas para cada classe.

Em vez de duplicar o código, a herança permite que defina o comportamento compartilhado em uma classe principal e o estenda em classes secundárias especializadas.

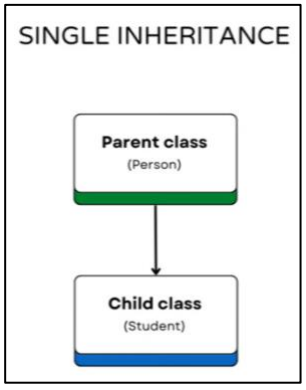
Exemplo: a classe Livro especifica um método para atribuir Autor e a classe Software especifica um método para definir a versão do software);



Ao contrário de algumas linguagens de programação, o Python permite ter dois tipos de heranças:

- *Heranças Simples*
- *Heranças Múltiplas*

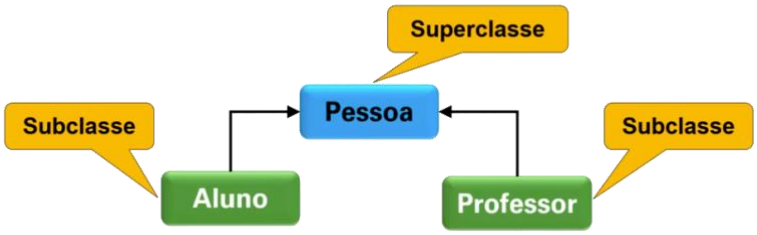
2.15.1 – Herança Simples



- Ocorre quando uma classe herda os atributos e métodos de ***somente uma outra classe***.
- Isto permite criar hierarquias de classes de maneira organizada e lógica.
- Em Python, a herança simples é facilmente implementada utilizando parênteses () para ***especificar a superclasse***.

Vamos considerar o seguinte exemplo: Temos uma ***superclasse*** com o nome Pessoa e as ***subclasses*** Aluno e Professor

Esta superclasse vai ter atributos e métodos que depois irá passar para subclasses que herdam desta classe.



Superclasse Pessoa

- Como tal, vamos construir a ***classe Pessoa e criar o construtor*** da classe. No construtor, vamos indicar as **variáveis de classe**: ID, nome e Idade

```python
#classe Pessoa que vai ter dados: ID, Nome e Idade
class Pessoa:
    #iniciar o construtor (com parametros de entrada)
    def __init__(self, id, nome, idade):
        self.ID = id
        self.Nome = nome
        self.Idade = idade
```

- Para além dessa informação, vamos criar um método da classe para retornar os dados das variáveis da classe com o nome Mostra_Dados:

```python
#metódo de instância (para retornar os dados)
def Mostra_Dados(self):
    return f"ID: {self.ID}, Nome: {self.Nome}, Idade: {self.Idade}"
```

Superclasse Pessoa (completo)

```python
#classe Pessoa que vai ter dados: ID, Nome e Idade
class Pessoa:
    #iniciar o construtor (com parametros de entrada)
    def __init__(self, id, nome, idade):
        self.ID = id
        self.Nome = nome
        self.Idade = idade

    #metódo de instância (para retornar os dados)
    def Mostra_Dados(self):
        return f"ID: {self.ID}, Nome: {self.Nome}, Idade: {self.Idade}"
```

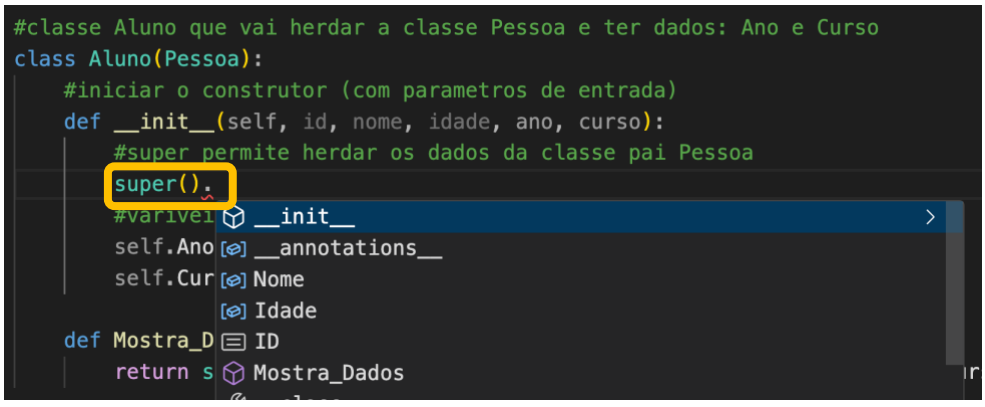
- Repare que construir a superclasse é a mesma coisa que construir uma classe normal (e não houve a preocupação de indicar alguma informação adicional);
- Quando criar as outras classes e se desejar fazer a herança, só aí é que vai ter de ter de indicar a herança;

Subclasse Aluno

- Vamos começar por implementar a subclasse que irá herdar dados da superclasse Pessoa. Como tal, vamos começar por construir o início da classe e para dizer que esta classe vai herdar outra classe, logo a seguir ao nome da classe, vamos colocar entre parênteses o nome da superclasse (neste caso a classe Pessoa);
- De seguida, vamos fazer o construtor da classe e, como tal, o construtor vai ter de indicar as variáveis que vão ser herdadas da superclasse e depois disso é que refere as variáveis especificas da nova classe:
  - **Variáveis herdadas da superclasse Produto:** Id, Nome e Idade
  - **Variáveis exclusivas da subclasse Aluno:** ano e curso

```
#classe Aluno que vai herdar a classe Pessoa e ter dados: Ano e Curso
class Aluno(Pessoa):
    #iniciar o construtor (com parametros de entrada)
    def __init__(self, id, nome, idade, ano, curso):
```

- Dentro do construtor, a primeira linha de código vai servir para chamar a herança da superclasse Pessoa e herdar três variáveis: ID, nome e idade.
- Para tal, vamos usar a palavra reservada **super** e logo a seguir colocamos **o ponto final** para ***aceder*** as propriedades e métodos da superclasse
- A função ***super()*** do Python é usada para ***dar acesso a métodos de uma classe herdada***.



- O super() é utilizado entre heranças de classes e vai disponibilizar métodos de uma superclasse (classe pai) para uma subclasse (classe filha).
- Atráves do super() definimos um novo comportamento para um determinado método construído na classe pai que vai ser herdado pela classe filha.

```
#iniciar o construtor (com parametros de entrada)
def __init__(self, id, nome, idade, ano, curso):
    #super permite herdar os dados da classe pai Pessoa
    super().__init__(id, nome, idade)
    #variveis de instancia exclusivas da classe Aluno
    self.Ano = ano
    self.Curso = curso
```

- No exemplo anterior, temos uma subclasse Aluno que herda os dados da superclasse Pessoa e utiliza o super() para chamar o método construtor da classe Pai no seu método construtor.
- Logo a seguir ao super(), declaramos as variáveis especificas da subclasse Aluno que são o ano e o curso.
- Após a criação do construtor (com herança da superclasse), vamos criar um método da classe para mostrar os dados, do qual, vamos invocar o método do super().MostraDados() para ir buscar a mensagem a superclasse e juntar com os novos dados da subclasse:

```
#Metodo de instancia com herança do Mostra_Dados da superclasse Produto
def Mostra_Dados(self):
    return super().Mostra_Dados() + f", Ano: {self.Ano}, Curso: {self.Curso}"
```

Subclasse Aluno (completa)

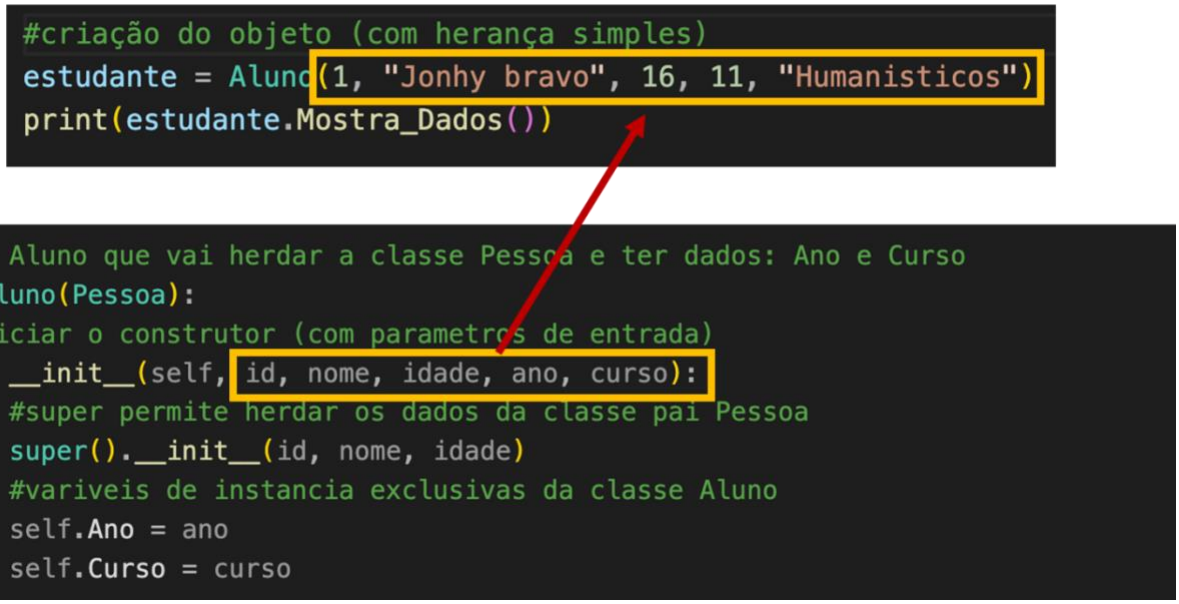
```
#classe Aluno que vai herdar a classe Pessoa e ter dados: Ano e Curso
class Aluno(Pessoa):
    #iniciar o construtor (com parametros de entrada)
    def __init__(self, id, nome, idade, ano, curso):
        #super permite herdar os dados da classe pai Pessoa
        super().__init__(id, nome, idade)
        #variveis de instancia exclusivas da classe Aluno
        self.Ano = ano
        self.Curso = curso

    #Metodo de instancia com herança do Mostra_Dados da superclasse Produto
    def Mostra_Dados(self):
        return super().Mostra_Dados() + f", Ano: {self.Ano}, Curso: {self.Curso}"
```

Testar a herança simples Pessoa → Aluno

Para instanciar e construir um novo objeto vamos usar a subclasse Aluno (uma vez que está vai herdar os dados da superclasse Pessoa)

```
#criação do objeto (com herança simples)
estudante = Aluno(1, "Jonhy bravo", 16, 11, "Humanisticos")
print(estudante.Mostra_Dados())
```

O número de parâmetros deve coincidir a entrada de dados no construtor da subclasse



Resultado final:

```
#criação do objeto (com herança simples)
estudante = Aluno(1, "Jonhy bravo", 16, 11, "Humanisticos")
print(estudante.Mostra_Dados())
```

```
ID: 1, Nome: Jonhy bravo, Idade: 16, Ano: 11, Curso: Humanisticos
```

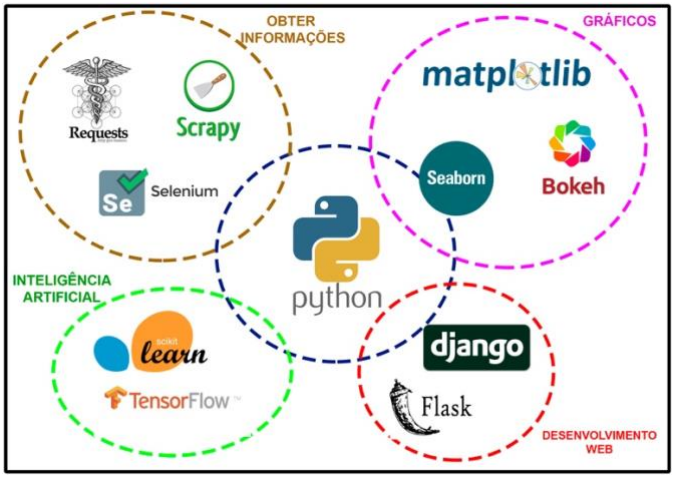
3 – Bibliotecas Python

3.1 – Bibliotecas? O que é isso?

Uma biblioteca em Python é um conjunto de módulos e funções pré-definidos que podem ser utilizados para facilitar o desenvolvimento dos programas.

Em outras palavras, é um conjunto de código que já foi escrito e pode ser reutilizado em diferentes projetos.

As bibliotecas em Python são criadas para resolver problemas específicos e fornecer funcionalidades extras que não estão disponíveis no núcleo da linguagem.



3.2 – Vantagens das Bibliotecas

Existem várias vantagens em utilizar bibliotecas em Python para o desenvolvimento de projetos.

Vamos ver algumas:

1. ***Aumento da produtividade:*** Ao utilizar bibliotecas, é possível aproveitar o trabalho de outros programadores, economizando tempo e esforço na implementação de funcionalidades;
2. ***Maior qualidade do código:*** As bibliotecas são desenvolvidas e testadas por uma comunidade de programadores experientes e passam por revisões e melhorias constantes;
3. ***Acesso a funcionalidades avançadas:*** oferecem uma gama de funcionalidades que são usadas para resolver problemas específicos;
4. ***Facilidade de integração:*** são projetadas para serem facilmente integradas em outros projetos, ou seja, pode combinar diferentes bibliotecas para obter várias funcionalidades;

Bibliografia

Metz, Sandi – Practical Object-Oriented Design: An Agile Primer Using Ruby 2th Edition. Addison-Wesley Professional, 2018. ISBN: 978-0134456478

Liberty, Jesse – Git for Programmers: Master Git for effective implementation of version control for your programming projects. Packt, 2021. ISBN: 978-1801075732

Henriques, Telmo – Gestão de Sistemas de Informação. FCA, 2019. ISBN: 978-972-722-903-1

Henriques, Telmo – Gestão de Sistemas de Informação - Frameworks, Modelos e Processos. FCA, 2019. ISBN: 978-972-722-899-7

Santos, Maribel & Ramos, Isabel – Business Intelligence - Da Informação ao Conhecimento. FCA, 2017. ISBN: 978-972-722-880-5

Costa, Ernesto – Programação em Python. FCA, 2015. ISBN: 978-972-722-816-4

Portela, Filipe & Pereira, Tiago C. - Introdução à Algoritmia e Programação com Python. FCA, 2023. ISBN: 978-972-722-931-4

Carvalho, Adelaide - Práticas de Python. FCA, 2021. ISBN: 978-972-722-918-5

Vasconcelos, José Braga de & Barão, Alexandre – Ciência de Dados nas organizações. FCA, 2017. ISBN: 978-972-722-885-0

Vasconcelos, José Braga de – Python – Algoritmia e programação Web. FCA, 2015. ISBN: 978-972-722-813-3