

Conceitos básicos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-Compartilhado 4.0 Internacional.

<http://github.com/malbarbo/na-programacao>

Até o momento nós estudamos alguns aspectos de

- Sistemas computacionais
- Algoritmos
- Linguagens de programação

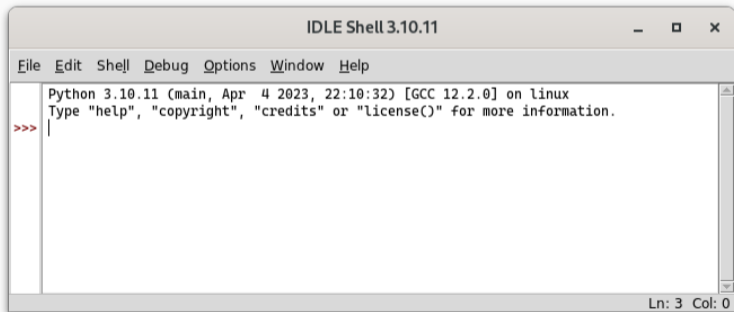
Agora vamos ver as construções básicas da linguagem Python, para em seguida começarmos a estudar o processo de projeto de programas.

Instalação

O Python é um software livre e pode ser baixado e instalado de <https://python.org>.

Além do interpretador, a instalação do Python vem com um ambiente de desenvolvimento e aprendizagem chamado IDLE.

Ao iniciar o IDLE a janela a seguir é exibida



```
Python 3.10.11 (main, Apr 4 2023, 22:10:32) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

The image shows a screenshot of the IDLE Shell 3.10.11 window. The window title is "IDLE Shell 3.10.11". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area displays the Python startup message: "Python 3.10.11 (main, Apr 4 2023, 22:10:32) [GCC 12.2.0] on linux" followed by "Type 'help', 'copyright', 'credits' or 'license()' for more information." Below this, the prompt ">>>" is shown with a vertical cursor on the next line. The status bar at the bottom right indicates "Ln: 3 Col: 0".

Utilizamos essa janela, chamada de janela de **interações** (ou REPL), para testar pequenos trechos de código.

O símbolo `>>>` é chamado de *prompt* e indica que o interpretador está pronto.

As interações acontecem da seguinte forma

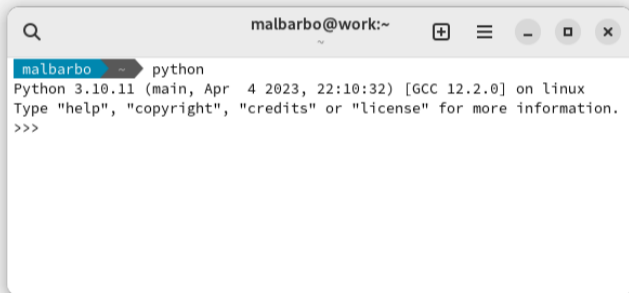
- Digitamos um trecho de código (*Read*)
- O código é avaliado (*Eval*)
- O resultado é mostrado na tela (*Print*)
- O processo se repete (*Loop*)

Exemplo de interação

```
>>> 3 * 4
```

```
12
```

O modo de interações também pode ser iniciado executado `python` no terminal de comandos.



```
malbarbo@work:~  
malbarbo ~$ python  
Python 3.10.11 (main, Apr 4 2023, 22:10:32) [GCC 12.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

Note que na janela de interações não criamos programas para serem utilizados por usuários, mas experimentamos aspectos do Python e testamos os nossos programas.

Veremos posteriormente como criar programas completos.

Agora vamos explorar o Python!

A primeira coisa que aprendemos de uma linguagem de programação são os tipos de valores (tipos de dados) e operações já disponíveis na linguagem.

Os primeiros computadores foram criados para fazerem cálculos matemáticos, então vamos começar com isso.

O Python tem diversos **tipos numéricos**, os dois principais são

Inteiros (**int**)

```
>>> 102
```

```
102
```

```
>>> -18
```

```
-18
```

Ponto flutuante (**float**), representação aproximada de números reais

```
>>> 1.3
```

```
1.3
```

```
>>> 0.345
```

```
0.345
```

```
>>> # Notação científica
```

```
>>> 1.23e2 # 1.23 * 102
```

```
123.0
```

Podemos usar as quatro operações aritméticas básicas com esses tipos numéricos e algumas outras operações, conforme veremos a seguir.

```
>>> # Soma e subtração
```

```
>>> 4 + 2
```

```
6
```

```
>>> 4 + 2.0 - 5
```

```
1.0
```

```
>>> # Multiplicação e divisão
```

```
>>> 3 * 5.0
```

```
15.0
```

```
>>> 7 / 2
```

```
3.5
```

```
>>> # Divisão sempre produz float
```

```
>>> 8 / 4
```

```
2.0
```

```
>>> # Piso da divisão
```

```
>>> 7 // 2
```

```
3
```

```
>>> 5 // 1.3
```

```
3.0
```

```
>>> # Resto da divisão
```

```
>>> 14 % 3
```

```
2
```

```
>>> # Esperávamos obter exatamente
```

```
>>> # 1.1, mas float é apenas
```

```
>>> # uma aproximação dos reais...
```

```
>>> 5 % 1.3
```

```
1.0999999999999999
```

O símbolo `#` (cerquilha), é utilizado para indicar um **comentário**. O comentário inicia em `#` e vai até o final da linha. Os comentários são ignorados pelo interpretador do Python, mas são utilizados para adicionar informações relevantes para os leitores do código.

Exponenciação

```
>>> # Exponenciação e radiciação
```

```
>>> 3 ** 4 # 3 elevado a 4
```

```
81
```

```
>>> 2 ** 80
```

```
1208925819614629174706176
```

```
>>> 16 ** 0.5 # raiz quadrada, o mesmo que 16 ** (1 / 2)
```

```
4.0
```

```
>>> # A exponenciação tem prioridade sobre a divisão
```

```
>>> 27 ** 1 / 3 # o mesmo que (27 ** 1) / 3
```

```
9.0
```

```
>>> # Usamos parênteses para mudar a prioridade
```

```
>>> 27 ** (1 / 3) # raiz cúbica
```

```
3.0
```

O Python utiliza a mesma precedência que estamos acostumados na matemática. Podemos usar o acrônimo PEMDAS para lembrar das prioridades

- Parênteses
- Exponenciação
- Multiplicação e Divisão
- Adição e Subtração

Operadores com a mesma precedência são avaliados da esquerda para a direita, exceto a exponenciação, que é da direita para a esquerda.

Qual é o resultado da avaliação de cada expressão a seguir?

```
>>> 15 // 7
```

```
>>> 15 % 7
```

```
>>> 12 // 27
```

```
>>> 12 % 27
```

```
>>> 3 * 4 - 5 / (8 // 3)
```

```
>>> 5 * 8 // 3 / 4 % 3
```

```
>>> 2 ** 2 ** 3 // 4 * 3
```

Exercício

Qual é o resultado da avaliação de cada expressão a seguir?

```
>>> 15 // 7
```

```
2
```

```
>>> 15 % 7
```

```
1
```

```
>>> 12 // 27
```

```
0
```

```
>>> 12 % 27
```

```
12
```

```
>>> 3 * 4 - 5 / (8 // 3)
```

```
9.5
```

```
>>> 5 * 8 // 3 / 4 % 3
```

```
0.25
```

```
>>> 2 ** 2 ** 3 // 4 * 3
```

```
192
```



```
>>> # Arredondamento
```

```
>>> round(3.4)
```

```
3
```

```
>>> round(3.5)
```

```
4
```

```
>>> round(3.5134, 2)
```

```
3.51
```

```
>>> # Conversão entre int e float
```

```
>>> int(7.6)
```

```
7
```

```
>>> int(-2.3)
```

```
-2
```

```
>>> float(4)
```

```
4.0
```

As operações que vimos até agora estão disponíveis automaticamente, outras operações estão disponíveis em módulos, que devem ser importados antes de poderem ser utilizados.

O Python tem uma extensa biblioteca padrão, com muitos módulos, este é um dos motivos pelos quais a linguagem é bastante utilizada. A documentação da biblioteca padrão do Python está disponível em <https://docs.python.org/3/library/index.html>.

Por hora, vamos ver apenas algumas funções do módulo `math`.

```
>>> import math
>>> # Piso
>>> # maior inteiro <= ao número
>>> math.floor(4.2)
4
>>> math.floor(4.0)
4
>>> math.floor(-2.3)
-3
```

```
>>> # Teto
>>> # menor inteiro >= ao número
>>> math.ceil(4.2)
5
>>> math.ceil(4.0)
4
>>> math.ceil(-2.3)
-2
```

Outro tipo de dado pré-definido em Python é a cadeia de caracteres (**str**), *string* em inglês.

Geralmente usamos strings para armazenar informações simbólicas, como por exemplo palavras e textos.

Uma string em Python é escrita entre apóstrofo (') ou aspas (")

```
>>> 'casa'
```

```
'casa'
```

```
>>> "gota d'agua"
```

```
"gota d'agua"
```

```
>>> "mesa"
```

```
'mesa'
```

Operações com strings

Assim como existem operações pré-definidas para números, também existem operações pré-definidas para strings.

```
>>> # Concatenação
>>> 'casa' + ' da ' + 'sogra'
'casa da sogra'
```

```
>>> # Repetição
>>> 'abc' * 3
'abcabcabc'
```

```
>>> 'algum' * 0
''
>>> 'algum' * -4
''
```

```
>>> # Quantidade de caracteres
>>> len('ciência da computação')
21
```

```
>>> # Conversão maiúscula
>>> 'José'.upper() # ou str.upper('José')
'JOSÉ'
```

```
>>> # Conversão minúscula
>>> 'José'.lower() # ou str.lower('José')
'josé'
```

```
>>> # Indexação de caractere
>>> # 0 primeiro caractere tem índice 0
>>> 'casa'[0] # ou str.__getitem__('casa', 0)
'c'

>>> 'casa'[1]
'a'

>>> # Acesso de índice fora do intervalo
>>> 'casa'[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

```
>>> # Substring do início até 3 - 1
>>> 'veja isso'[:3] # ou str.__getitem__('veja isso', slice(None, 3))
'vej'

>>> # Substring de 4 até o final
>>> 'veja isso'[4:] # ou str.__getitem__('veja isso', slice(4, None))
'isso'

>>> # Substring de 2 até 6 - 1
>>> 'veja isso'[2:6] # ou str.__getitem__('veja isso', slice(7, 9))
'ja i'
```

```
>>> # Conversão de int para str
```

```
>>> str(127)
```

```
'127'
```

```
>>> # Conversão de float para str
```

```
>>> str(4.1)
```

```
'4.1'
```

```
>>> # Concatenação de str e int
```

```
>>> 'Idade: ' + str(19)
```

```
'Idade: 19'
```

```
>>> # Conversão de str para int
```

```
>>> int('127')
```

```
127
```

```
>>> # Conversão de str para float
```

```
>>> float('25')
```

```
25.0
```

```
>>> float('12.67')
```

```
12.67
```


Inicialmente as expressões que vimos usavam apenas operadores matemáticos

```
30 * 2
```

Depois vimos que as expressões podem conter chamadas de funções

```
round(3.5)
```

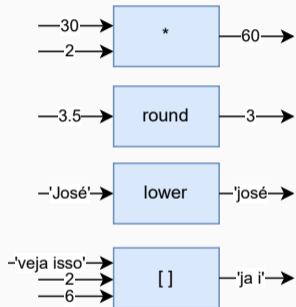
e chamadas de métodos

```
'José'.lower()
```

Por fim, vimos que strings podem ser indexadas

```
'veja isso'[2:6]
```

Embora a forma de utilizar operadores, funções, métodos e indexação seja diferente, o propósito dessas construções é o mesmo: computar valores de saída a partir de valores de entrada.



Se o propósito é o mesmo, por que não usar a mesma forma?

Por conveniência!

Por exemplo, se não tivéssemos a forma de operadores e apenas a forma de chamada de funções, então deveríamos escrever

```
from operator import add, mul
add(mul(30, 2), 3)
```

ao invés de `30 * 2 + 3`, o que seria inconveniente.

Além da conveniência de escrita, a forma de chamada métodos e indexação tem outras vantagens, que não vamos discutir nessa disciplina.

Qual é o resultado de cada expressão a seguir?

```
>>> len('casa') * 'x'
```

```
>>> str(10) + 2 * '*' + str(2.0)
```

```
>>> 'Jose da Silva'[:4].upper()
```

```
>>> 'Jose da Silva'[8:].lower()
```

```
>>> str(int('12') * 100)[1:3]
```

```
>>> int(('1' * 3 + 3 * '2').upper()[2:4])
```

Qual é o resultado de cada expressão a seguir?

```
>>> len('casa') * 'x'
'xxxx'
>>> str(10) + 2 * '*' + str(2.0)
'10**2.0'
>>> 'Jose da Silva'[:4].upper()
'JOSE'
>>> 'Jose da Silva'[8:].lower()
'silva'
>>> str(int('12') * 100)[1:3]
'20'
>>> int(('1' * 3 + 3 * '2').upper()[2:4])
12
```

Outro tipo de operação que podemos fazer com números e strings são as operações relacionais.

Que resposta você espera para a comparação `3 > 4`? E para `3 < 4`?

Em Python a resposta da primeira comparação é `False` (falso) e da segunda `True` (verdadeiro).

```
>>> 3 > 4
```

```
False
```

```
>>> 3 < 4
```

```
True
```

Na computação os valores verdadeiro e falso são chamados de booleanos. Em Python, o tipo dos valores booleanos é `bool`. As operações relacionais produzem como resposta um valor booleano.

```
>>> # Maior e maior ou igual
```

```
>>> 4 > 4
```

```
False
```

```
>>> 4 >= 4
```

```
True
```

```
>>> # Menor e menor ou igual
```

```
>>> 6.0 < 6.0
```

```
False
```

```
>>> 6.0 <= 1.0 + 5.0
```

```
True
```

```
>>> # Igual
```

```
>>> 5 == 6
```

```
False
```

```
>>> 9 == 5 + 2 ** 2
```

```
True
```

```
>>> # Diferente
```

```
>>> 3 * 2 != 4 + 2 ** 2
```

```
True
```

```
>>> 9 != 4 + 2 ** 2
```

```
False
```

Observe que as operações relacionais tem prioridade menor do que as operações aritméticas.

As operações relacionais podem ser utilizadas com outros tipos, incluindo strings e booleanos.

```
>>> # As strings são comparadas
>>> # lexicograficamente, o
>>> # que pode gerar algumas
>>> # supressas
>>> 'a' < 'b'
True
>>> 'á' < 'b'
False
```

```
>>> 'Abacaxi' < 'Abacate'
False
>>> 'André' < 'paulo'
True
>>> 'casa' == 'Casa'
False
>>> 'á' != 'a'
True
```



```
>>> # O valor False é considerado
```

```
>>> # menor que o valor True
```

```
>>> False < True
```

```
True
```

```
>>> True > False
```

```
True
```

```
>>> False == False
```

```
True
```

```
>>> False == True
```

```
False
```

```
>>> True == False
```

```
False
```

```
>>> True == True
```

```
True
```

Assim como existem operações com números e strings, também existem operações com booleanos.

As três operações mais comuns com booleanos são: **not** (negação), **or** (ou) e **and** (e).

```
>>> # 0 not é um operador unário.
```

```
>>> not True
```

```
False
```

```
>>> not False
```

```
True
```

```
>>> not not True
```

```
True
```

```
>>> # 0 not tem menor precedência
```

```
>>> # do que os operadores relacionais
```

```
>>> # e aritméticos.
```

```
>>> # 4 > 4.0 é False
```

```
>>> not 3 + 1 > 2 + 2.0
```

```
True
```

```
>>> # 14 == 14 é True
```

```
>>> not 2 + 3 * 4 == 14
```

```
False
```

Operadores booleanos

```
>>> # 0 and é um operador binário
>>> # que só produz True se os
>>> # dois operandos forem True.
```

```
>>> # Tabela verdade do and
>>> False and False
False
>>> False and True
False
>>> True and False
False
>>> True and True
True
```

```
>>> # 0 and tem menor precedência
>>> # do que os operadores relacionais
>>> # e aritméticos.
```

```
>>> # 15 > 8 é True
>>> # 4 == 3 é False
>>> 15 > 2 ** 3 and 4 == 1 + 2
False
>>> # 2 == 2 é True
>>> # 3 != 4 é True
>>> 2 == 1 + 1 and 3 != 4
True
```

Operadores booleanos

```
>>> # 0 or é um operador binário      >>> # 0 or tem menor precedência
>>> # que produz True se pelo menos >>> # do que os operadores relacionais
>>> # um dos operandos for True.      >>> # e aritméticos.

>>> # Tabela verdade do or           >>> # 15 > 8 é True
>>> False or False                  >>> # 4 == 3 é False
False                                >>> 15 > 2 ** 3 or 4 == 1 + 2
>>> False or True                    True
True                                  >>> # 2 == 3 é False
>>> True or False                    >>> # 3 + 1 != 4 é False
True                                   >>> 2 == 2 + 1 or 3 + 1 != 4
>>> True or True                      False
True
```

Quem tem maior prioridade, o **and** ou o **or**? O **and**. Vamos criar uma expressão que mostre que isso é verdade.

```
>>> True or False and False
True
```

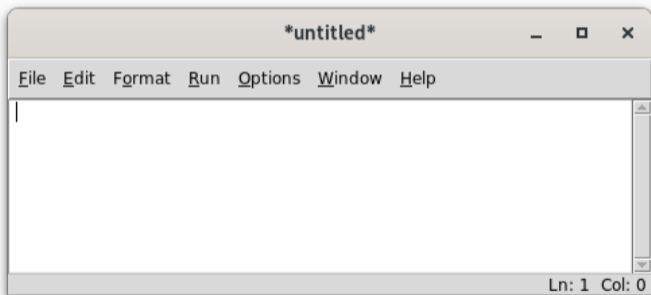
```
>>> # É equivalente a expressão anterior
>>> True or (False and False)
True
```

```
>>> # Se o or tivesse prioridade...
>>> (True or False) and False
False
```

Além de podermos usar as operações e funções pré-definidas no Python, também podemos definir as nossas próprias funções.

Apesar de ser possível definir uma nova função na janela de interações, nós vamos fazer isso na janela de edição de código. Isso nos permite salvar o código para uso/edição posterior.

Para abrir a janela de edição de código selecionamos o menu File → New File (ctrl + n).



Vamos criar uma função que calcula o dobro de um dado número.

O que precisamos para definir essa função?

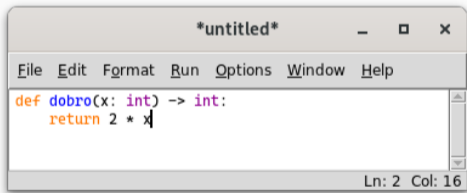
- O nome da função (**dobro**)
- O nome e tipo da entrada (**x: int**)
- O tipo da saída (**int**)
- As instruções que calculam o valor da saída a partir da entrada (**2 * x**)

Com essas informações, definimos a função usando a forma

```
def nome(entrada1: tipo, entrada2: tipo, ...) -> tipo:  
    return exp
```

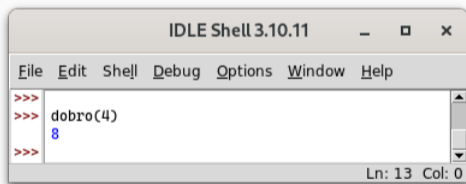

Definindo novas funções

Escrevemos o código da função na janela de edição de código e salvamos o arquivo (File → Save - ctrl + s).



```
File Edit Format Run Options Window Help
def dobro(x: int) -> int:
    return 2 * x
Ln: 2 Col: 16
```

Para testarmos a função executamos o arquivo (Run → Run Module - F5) e chamamos a função na janela de interações.



```
IDLE Shell 3.10.11
File Edit Shell Debug Options Window Help
>>> dobro(4)
8
>>>
Ln: 13 Col: 0
```

O que acontece se escrevermos uma chamada função **dobro** após a sua definição e executarmos o arquivo (Run Module)?

```
def dobro(x: int) -> int:  
    return 2 * x
```

```
dobro(4)
```

A função **dobro** será executada para o valor **4** mais nenhum resultado será exibido na tela.

Por que na execução do exemplo na janela de interações o resultado é exibido e aqui não?

Por que no modo de interação a exibição é feita automaticamente (o **P** – *print* – de REPL) para facilitar a interação com o Python. No arquivo de código, precisamos indicar explicitamente que queremos que o resultado seja exibido.

A forma mais comum de exibir um valor em Python é utilizando a função `print`.

```
def dobro(x: int) -> int:  
    return 2 * x
```

```
print(dobro(4))
```

Ao executar o código, o valor **8** será exibido na tela.

Note que o `print` posiciona o cursor no início da próxima linha, dessa forma, a próxima informação começará a ser exibida no início da próxima linha.

A função `print` pode ser utilizada com mais de um argumento e os argumentos podem ser de tipos diferentes

```
def dobro(x: int) -> int:  
    return 2 * x  
  
print('O dobro de 4 é:', dobro(4))
```

Saída

O dobro de 4 é: 8

Note que não colocamos espaço após `'é:'` na chamada do `print`, mas um espaço aparece na saída. Isto porque o `print` adiciona um espaço automaticamente entre cada argumento antes de exibir na tela.

Esse comportamento do `print` pode ser indesejável em algumas situações, como por exemplo, se quiséssemos colocar um ponto final após o valor calculado do dobro. Nesse caso, podemos instruir o `print` a não fazer a separação automática (especificando `sep= ''`) e fazermos a separação manualmente

```
def dobro(x: int) -> int:  
    return 2 * x
```

```
print('O dobro de 4 é: ', dobro(4), '.', sep='')
```

Agora que aprendemos algumas das construções básicas do Python, podemos avançar e ver como fazer um programa completo.

Para isso, vamos precisar de mais duas coisas

- Instrução de entrada
- Variáveis

Uma **variável** é um nome para uma região da memória (célula) que é utilizada para armazenar valores.

Cada variável tem um tipo, que determina o conjunto de valores que podem ser armazenados na memória associada com ela.

Uma variável pode ser primeiro declarada e depois inicializa ou pode ser declarada e inicializada de uma vez só.

A forma geral para declaração de variável é (o **valor inicial** é opcional)

```
nome: tipo [= valor inicial]
```

```
>>> a: int = 10
>>> b: int = 2 * a
>>> b
20
```

As variáveis **a** e **b** foram declaradas com tipo **int** e inicializadas na declaração. A variável **a** foi inicializada com o valor **10** e a variável **b** com o valor **2 * a**.

O símbolo **=** representa **atribuição**. Para executar uma atribuição o Python primeiro avalia a expressão do lado direito para obter um valor, e depois associa a memória que armazena esse valor com o nome da variável.

O Python executa as instruções de forma sequencial, uma linha após a outra, por isso, a ordem das instruções é importante.

Qual o resultado da execução das instruções abaixo?

```
>>> x: int = 2 * y
>>> y: int = 10
>>> x
?
```

Um erro de execução! Quando o Python avalia a expressão `2 * y` a variável `y` ainda não foi definida, então não é possível calcular o valor da expressão.

Qual o resultado de **b** no seguinte trecho de código?

```
>>> a: int = 10
>>> b: int = 2 * a
>>> a = 30
>>> b
?
```

20. O Python executar uma linha por vez, na primeira linha a variável **a** é criada referenciando uma célula de memória com o valor **10**. Depois a expressão **2 * a** é avaliada com resultado **20** e a variável **b** é criada referenciando a célula de memória que armazena esse valor. Depois a variável **a** é alterada, passando a referenciar a célula de memória com o valor **30**. Por fim, o valor armazenado na célula de memória associada com **b**, que é **20**, é exibido.

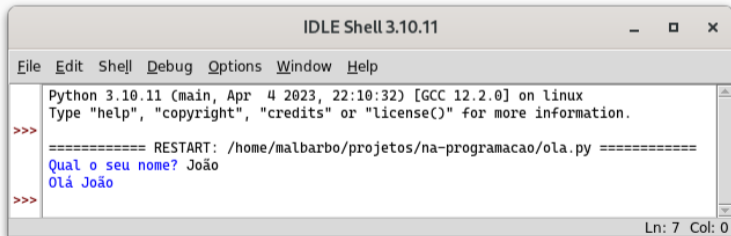
A forma mais comum de ler um valor digitado pelo usuário em Python é utilizando a função `input`.

A função `input` exibe uma mensagem e aguarda o usuário digitar a entrada e pressionar enter. Em seguida o valor digitado pelo usuário é retornado como uma string.

Se escrevermos o seguinte código em um arquivo `ola.py`,

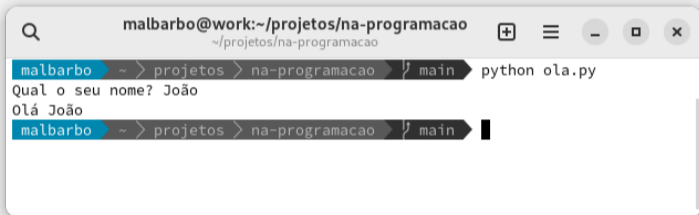
```
nome: str = input('Qual o seu nome?: ')  
print('Olá', nome)
```

executarmos com F5, e fizermos a entrada do nome `João`, obtemos o seguinte resultado



```
Python 3.10.11 (main, Apr 4 2023, 22:10:32) [GCC 12.2.0] on linux  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: /home/malbarbo/projetos/na-programacao/ola.py =====  
Qual o seu nome? João  
Olá João  
>>>
```

Também podemos executar o programa no terminal de comando



```
malbarbo@work:~/projetos/na-programacao
~/projetos/na-programacao
malbarbo ~ > projetos > na-programacao > main > python ola.py
Qual o seu nome? João
Olá João
malbarbo ~ > projetos > na-programacao > main > █
```

Note que é preciso estar no diretório onde o arquivo está salvo (no caso `projetos/na-programacao/`) ou utilizar o caminho completo para o arquivo.

Instrução de entrada vs modo interativo

Apesar de ser possível utilizar as funções de entrada e saída na janela de interações, isso não é muito comum, afinal, podemos executar diretamente as funções com os argumentos que queremos e ver o resultado automaticamente, não é necessário perguntar nada para o usuário!

Por exemplo, se estamos testando a função `len`, ao invés de fazermos a entrada

```
>>> palavra: str = input('Digite uma palavra: ')
```

```
Digite uma palavra: casa
```

```
>>> len(palavra)
```

```
4
```

podemos fazer diretamente

```
>>> len('casa')
```

```
4
```

Agora já podemos criar um programa completo com entrada, processamento e saída!

Vamos fazer um programa que utiliza a função **dobro** para fazer o processamento.

```
def dobro(x: int) -> int:
    return 2 * x

# Entrada
numero: int = int(input('Digite um número inteiro: '))

# Processamento
d: int = dobro(numero)

# Saída
print('O dobro de', numero, 'é', d)
```

Note que a função `input` produz como resultado uma string, como queremos um inteiro, usamos a função `int` para converter a entrada do usuário para um inteiro.

Programa para calcular o dobro de um número



```
malbarbo@work:~/projetos/na-programacao
~/projetos/na-programacao
malbarbo ~ > projetos > na-programacao > main > python dobro.py
Digite um número inteiro: 14
O dobro de 14 é 28
malbarbo ~ > projetos > na-programacao > main > █
```

Apesar do programa funcionar corretamente, podemos melhorar a organização agrupando as instruções de entrada, processamento e saída em uma função, que convencionalmente chamaremos de `main`.

Além disso, colocamos a função `main` no início do arquivo.

```
def main():  
    # Entrada  
    numero: int = int(input('Digite um número inteiro: '))  
  
    # Processamento  
    d: int = dobro(numero)  
  
    # Saída  
    print('O dobro de', numero, 'é', d)  
  
def dobro(x: int) -> int:  
    return 2 * x  
  
main()
```

Note que precisamos chamar explicitamente a função `main` no final do arquivo.

O que aconteceria se removêssemos a chamada da função `main` e executássemos o arquivo?

Nada, sem a chamada da função `main`, nenhuma instrução é executada.

Nós vimos anteriormente que o Python executa as instruções de forma sequencial, uma linha após a outra. Mas isso nem sempre é verdade, vamos ver como o Python executa o programa dobro.

Execução passo a passo

```
1  def main():
2      # Entrada
3      numero: int = int(input('Número: '))
4
5      # Processamento
6      d: int = dobro(numero)
7
8      # Saída
9      print('O dobro de', numero, 'é', d)
10
11 def dobro(x: int) -> int:
12     return 2 * x
13
14 main()
```

Em qual ordem as linhas do programa são executadas pelo Python?

14, e depois?

3 (primeira linha de código da função `main`), 6, e depois?

12 (primeira linha de código da função `dobro`), e depois?

6 (retorna com o resultado para onde `dobro` foi chamada), 9, e depois?

14 (retorna para onde `main` foi chamada)

Vamos parar um pouco e pensar sobre erros.

Você já encontrou algum tipo de erro enquanto fazia os programas? Sim!

Que tipo de erro?

- O programa nem inicia a execução
- O programa executa mas é interrompido por um erro
- O programa executa até o final mas gera uma resposta errada

Classificamos esses erros em estáticos e dinâmicos.

Os **erros estáticos** são aqueles detectados antes da execução do programa.

Os **erros dinâmicos** são detectados durante a execução do programa.

Por padrão, o único tipo de erro estático detectado pelo Python é o erro sintático.

Um **erro sintático** ocorre quando o programa não segue as regras sintáticas da linguagem e o interpretador não consegue “entender” a estrutura do programa, por isso o o interpretador nem inicia a execução do programa.


```
x: int = (2 + 4
```

Qual é o erro nesse código?

Faltou fechar o parênteses.

```
x: int = (2 + 4
          ^
```

```
SyntaxError: '(' was never closed
```

Erros sintáticos

```
nota maxima: int = 10
def: float = 20.3
```

Quais os erros nesse código?

Identificador com espaço no nome

```
nota maxima: int = 10
  ^^^^^^
```

SyntaxError: invalid syntax

Use da palavra chave **def** como identificador

```
def: float = 20.3
  ^
```

SyntaxError: invalid syntax

```
def soma(a: int b: int) -> int
  return a + b
```

Quais são os erros nesse código?

Falta a vírgula antes de b

```
def soma(a: int b: int) -> int
      ^^^^^
```

SyntaxError: invalid syntax. Perhaps you forgot a comma?

Falta os dois pontos após o tipo de retorno da função

```
def soma(a: int, b: int) -> int
                        ^
```

SyntaxError: expected ':'

```
def soma(a: int, b: int) -> int:  
return a + b
```

Qual é o erro nesse código?

Falta a indentação (recoo) do **return**.

```
    return a + b  
    ^
```

IndentationError: expected an indented block after function definition

```
def main():  
    nome: str = input('Qual é o seu nome?: ')  
    print('Olá', nome)
```

Qual é o erro nesse código?

A indentação está inconsistente. Devemos sempre utilizar 4 espaço para fazer a indentação.

```
    print('Olá', nome)  
        ^
```

IndentationError: unindent does not match any outer indentation level

```
a: int = 10 + "3"
```

Qual é o erro nesse código?

Uso de operandos de tipos inválidos para o operador +.

```
10 + "a"
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Esse é um erro sintático? Não! É um erro semântico.

Um **erro semântico** ocorre quando o interpretador não “consegue” atribuir um significado para uma construção mesmo ela sendo válida sintaticamente.

O Python identifica esse erro de forma estática ou dinâmica? De forma dinâmica, lembre-se, os únicos erros identificados de forma estática pelo Python são os erros sintáticos.

```
def dobro(x: int) -> int:  
    return 2 * x  
  
print(dobro(10))  
print(dobro(10.0))  
print(dobro('10'))
```

Quais são os erros nesse código?

Nenhum! Whyyyyy?

O Python ignora todas as anotações de tipo. Se um programa está sintaticamente correto, o Python faz a sua execução e só para quando o programa termina ou um erro semântico é encontrado.

Esse exemplo executa até o termino ou é interrompido por um erro? Executa até o final! Howwww?

Quando **dobro** é chamada com o valor **10** a expressão $2 * x$ produz **20** e esse valor é exibido na tela.

Em seguida **dobro** é chamada com o valor **10.0** e a expressão $2 * x$ produz **20.0** e esse valor é exibido na tela.

Por fim, **dobro** é chamada com o valor **'10'** e a expressão $2 * x$ produz **1010** e esse valor é exibido na tela.

Embora nesse caso específico seja interessante poder usar a função **dobro** para diversos tipos de dados, mesmo que não projetamos a função com esta intenção, em outros casos essa flexibilidade pode gerar erros de execução, ou pior, resultados inesperados.

Essa característica do Python é chamada de tipagem dinâmica, isso é, os tipos são associados com os valores, e não com as variáveis. Em Python, qualquer valor pode ser atribuído a qualquer variável.

Outras linguagens de programação, como C/C++, utilizam tipagem estática, onde os tipos são associados com as variáveis. Nessas linguagens, um valor só pode ser atribuído para uma variável se o tipo do valor é compatível com o tipo da variável.

Existem muitas considerações que podemos fazer sobre as vantagens e as desvantagens de cada modelo, mas nós vamos nos ater à um aspecto: o pedagógico.

Considerando a pedagogia que estamos utilizando na disciplina, é importante que os tipos sejam verificados estaticamente.

Mas como podemos fazer isso se o Python não funciona dessa forma?

Utilizando uma ferramenta chamada **mypy**.

O **mypy** é um analisador estático, que além de fazer uma análise estática dos tipos, também identifica uma série de erros de forma estática, que só seriam identificados pelo Python durante a execução do programa.

```
def dobro(x: int) -> int:
    return 2 * x

print(dobro(10))
print(dobro(10.0))
print(dobro('10'))
```

Ao executar o `mypy` para o arquivo `dobro.py` com o comando

```
mypy dobro.py
```

obtemos o seguinte resultado

```
x.py:5: error: Argument 1 to "dobro" has incompatible type "float"; expected "int"
x.py:6: error: Argument 1 to "dobro" has incompatible type "str"; expected "int"
Found 2 errors in 1 file (checked 1 source file)
```

Se um programa foi verificado pelo **mypy**, isto é, não tem erros de sintaxe ou semântica, significa que ele não tem erros? Não! Ainda podemos ter erros durante a execução do programa.

Um erro de execução pode fazer o programa

- Ser interrompido e exibir uma mensagem de erro (crashar)
- Entrar em um laço infinito e nunca terminar (travar)
- Continuar a execução e produzir a resposta errada

Como garantir que um programa não terá erros durante a execução? Veremos isso ao longo da disciplina.

Agora que conhecemos os conceitos básicos de programação e do Python, podemos avançar para o processo de projeto de programas.

Mas antes, pratique fazendo a lista de exercícios disponível na página da disciplina!

Até mais e bons estudos.