Conceitos básicos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

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

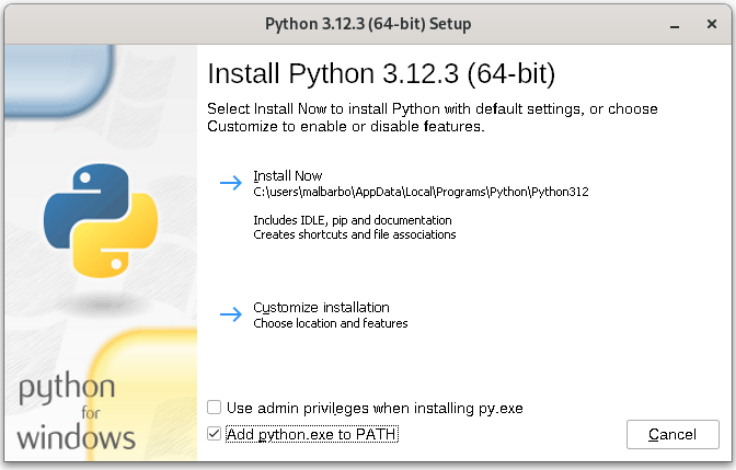
O Python é um software livre e pode ser baixado e instalado de https://python.org.

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

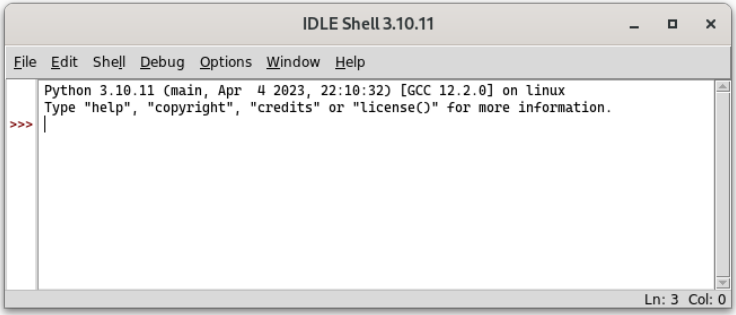
Em um sistema Linux, é provável que o Python já venha instalador por padrão. Nesse caso é preciso instalar apenas o IDLE. Em um sistema baseado no Debian, use o comando

```
$ sudo apt install idle
```

Durante a instalação no Windows é importante marcar a opção “Add python.exe to PATH”.



Ao iniciar o IDLE a janela a seguir é exibida



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.

Janela de interações (REPL)

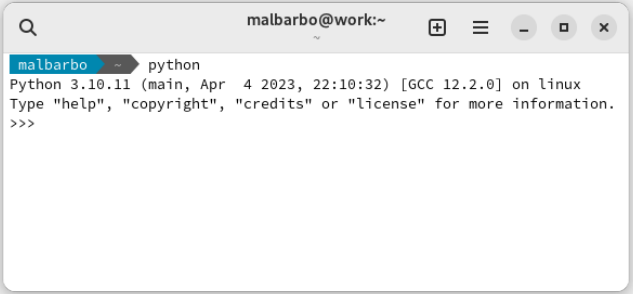
As interações acontecem da seguinte forma

- 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.



Observação

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 * 10^2
123.0
```

Podemos usar as quatro operações aritméticas básicas com esses tipos numéricos e algumas outras operações.

Operadores básicas

```
>>> # 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

>>> # Módulo
>>> 14 % 3
2
>>> -14 % 3
1

>>> # float é uma aproximação dos reais
>>> 5 % 1.3
1.0999999999999999
```

Comentários

O símbolo # (cerquilha), é utilizado para indicar um **comentário**. O comentário inicia na # 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
>>> # raiz quadrada, o mesmo que 16 ** (1 / 2)
>>> 16 ** 0.5
4.0

>>> # A exponenciação tem prioridade sobre a divisão
>>> # O mesmo que (27 ** 1) / 3
>>> 27 ** 1 / 3
9.0
>>> # Usamos parênteses para mudar a prioridade
>>> 27 ** (1 / 3) # raiz cubica
3.0
```

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

- **P**arênteses
- **E**xponenciação
- **M**ultiplicação e **D**ivisão
- **A**dição e **S**ubtração

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

Exercício

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(-1.6)
-2
>>> round(3.5134, 2)
3.51
```

```
>>> # Conversão entre int e float
>>> int(7.6)
7
>>> int(-2.3)
-2
>>> float(4)
4.0
```

Módulos

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`.

Piso e teto

```
>>> # Importação do módulo
>>> 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 númeo
>>> 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
>>> # O 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(2, 6))
'ja i'
```

Conversão entre strings e números

```
>>> # 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
```

Formas de expressões

Inicialmente as expressões que vimos usavam apenas operadores matemáticos

```
30 * 2
```

Depois vimos que as expressões podem conter chamadas de funções

```
round(3.5)
```

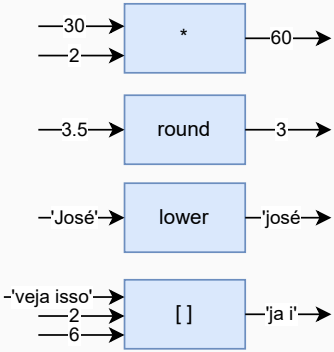
e chamadas de métodos

```
'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, para escrever a expressão `30 * 2 + 3` teríamos que escrever

```
int.__add__(int.__mul__(30, 2), 3)
```

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

Exercício

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])
```

Exercício

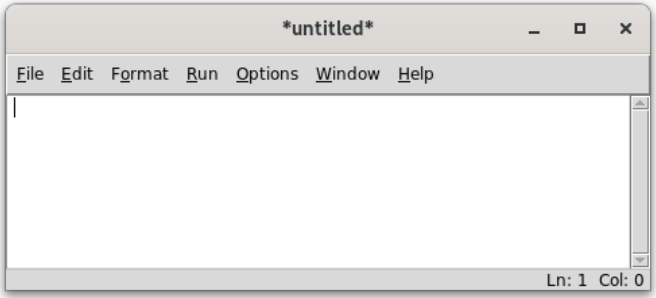
Qual é o resultado de cada expressão a seguir?

```
>>> 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
```

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 permite salvar o código para uso/edição posterior.

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



Funções na programação são semelhantes as funções na matemática, discutiremos as diferenças ao longo da disciplina. Por ora, vamos ver uma função matemática e tentar escrever "a mesma" função em Python.

Considere a função $f : \mathbb{Z} \to \mathbb{Z}$, que associa cada número inteiro ao dobro do seu valor, isto é, $f(x) = 2x$.

Quais são as partes que podemos identificar nessa definição?

- O nome da função (f)
- O nome do argumento de entrada (x) e seu domínio ($\mathbb{Z}$) (conjunto dos valores da entrada)
- O contradomínio ($\mathbb{Z}$) (conjunto dos valores da saída)
- A expressão que define a função ($2x$)

Vamos definir essa mesma função em Python.

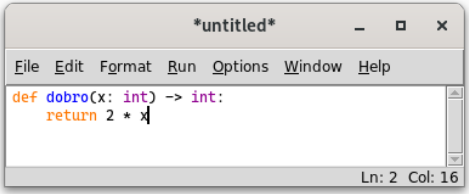
O que precisamos para definir a função?

- O nome da função (`dobro`)
- O nome e tipo da entrada (`x: int`)
- O tipo da saída (`int`)
- O corpo da função, isto é, as instruções que calculam o valor da saída a partir da entrada (`2 * x`)

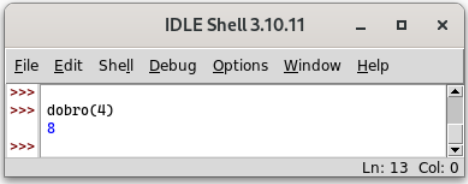
Para definir a função, usamos uma forma (sintaxe) específica:

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

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



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



Definição de funções

A forma geral para definição de funções:

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

def e **return** são **palavras chaves** (reservadas) e têm um significado pré-definido: **def** indica a definição de uma função; e **return** indica qual é a saída da função.

Os quatros espaços em branco antes do **return** é chamado de **indentação** (ou recuo). Em algumas linguagens a indentação é opcional, mas em Python é obrigatória.

Os símbolos `(`, `)`, `:`, `,` e `->`, entre outros, são os **delimitadores**.

Uma função pode ter zero ou mais parâmetros, mas conceitualmente só tem um valor de retorno.

O **identificador** (nome) da função, dos parâmetros e dos tipos deve começar com uma letra ou _ e pode ser seguido de letras, números e _, e não pode ser uma palavra reservada.

Espaços não podem ser usados em nomes.

As letras diacríticas (acentos, cedilha, etc) podem ser usados nos identificados, mas não é uma boa prática, por isso não vamos utilizar.

Exercício

Escreva uma função chamada `polegadas_em_mm` que converte uma medida `x` em polegadas para milímetros com até duas casas de precisão. Uma polegada equivale a 2,54 cm. Veja se a função funciona corretamente para os seguintes exemplos:

```
>>> polegadas_em_mm(10)
254.0
>>> polegadas_em_mm(1/2)
12.7
>>> polegadas_em_mm(5/8)
15.88
```

```python
def polegadas_em_mm(x: float) -> float:
    return x * 2.54
```

A função está correta? Não!.

```python
def polegadas_em_mm(x: float) -> float:
    return x * 25.4
```

E esta? Não, o último exemplo falha.

```python
def polegadas_em_mm(x: float) -> float:
    return round(x * 25.4, 2)
```

Operadores relacionais

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

Os **operadores relacionais** determinam se uma relação entre dois valores é verdadeira ou falsa.

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
```

Operadores relacionais

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
```

Quem tem maior prioridade, os operadores relacionais ou aritméticas? Os aritméticos.

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

As strings são comparadas lexicograficamente, o que pode gerar algumas surpresas.

```
>>> 'a' < 'b'
True
>>> 'á' < 'b'
False
>>> 'Paulo' < 'andré'
True
```

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

O valor `False` é considerado menor que o valor `True`, isso porque o `False` quando convertido para `int` é `0` e o `True` é `1`.

```
>>> int(False)
0
>>> int(True)
1
>>> False < True
True
>>> True > False
True
```

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

Operadores lógicos

Assim como existem operações com números e strings, também existem operações com booleanos, que são chamadas de **operações lógicas**.

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

O **not** é um operador unário, que produz o valor contrário do seu argumento.

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

```
>>> # 4 > 4.0 é False
>>> not 3 + 1 > 2 + 2.0
True
>>> # 14 == 14 é True
>>> not 2 + 3 * 4 == 14
False
```

Qual é precedência do **not** em relação aos operadores relacionais e aritméticos? É menor.

O **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
```

Qual é a precedência do **and** em relação aos operadores relacionais e aritméticos? É menor.

```
>>> # 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
```

O **or** é um operador binário que produz `True` se pelo menos um dos operandos for `True`.

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

Qual é a precedência do **or** em relação aos operadores relacionais e aritméticos? É menor.

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

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
```

Avaliação em curto circuito

Considere a expressão `x != 0 and 20 // x == 4`

Qual é o resultado da expressões quando `x` é `5`? `True`.

E quando `x` é `0`? `False`.

Por que? A avaliação não deveria falhar já que `20` está sendo dividido por `0`?

O Python, assim como a maioria das linguagens, faz uma avaliação mínima (também chamada de avaliação em **curto circuito**) de expressões booleanas, isto é, ele calcula apenas o mínimo para conseguir dar a resposta.

No caso, quando `x` é `0`, a expressão `x != 0` produz `False`, então, o resultado do **`and`** só pode ser `False`, independe do resultado da expressão `20 // x == 4`, por isso o Python não avalia essa segunda expressão.

Para expressões com **`or`** a ideia de avaliação mínima também é utilizada. Em um **`or`** com duas expressões, se a primeira for `True`, então o resultado do **`or`** só pode ser `True`, não sendo necessário avaliar a segunda expressão.

Dê um exemplo de uma expressão com **`or`** que falharia caso o Python não utilizasse avaliação mínima.

Exercício

Escreva uma função chamada `comaca_a` que recebe como parâmetro uma string `s` e determina se `s` começa com `'a'`. Veja se a função funciona corretamente para os seguintes exemplos:

```
>>> comeca_a('casa')
False
>>> comeca_a('abacate')
True
>>> comeca_a('Ana')
False
>>> comeca_a('')
False
```

```
def comeca_a(s: str) -> bool:
    return s[0] == 'a'
```

Esta função está correta? Não, o último exemplo gera uma falha de execução.

```
def comeca_a(s: str) -> bool:
    return s != '' and s[0] == 'a'
```

```
def comeca_a(s: str) -> bool:
    return s[:1] == 'a'
```

Exercício

Escreva uma função chamada `novo_seculo` que recebe como parâmetro uma string `data` no formato `dd/mm/aaaa` e indica se a data representa o primeiro dia de um novo século.

```
>>> novo_seculo('01/01/1900')
True
>>> novo_seculo('01/01/2000')
True
>>> novo_seculo('03/01/2100')
False
>>> novo_seculo('01/02/2000')
False
>>> novo_seculo('01/01/1230')
False
```

```
def novo_seculo(data: str) -> bool:
    return data[:2] == '01' and \
           data[3:5] == '01' and \
           data[8:] == '00'
```

Esta função está correta? Sim!

O código da função está claro? Talvez nem tanto... Por que? O propósito de algumas expressões, como `data[3:5]`, não está evidente.

Como podemos melhorar? Podemos usar variáveis locais!

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.

Já usamos variáveis para armazenar os valores dos argumentos das funções, mas podemos usar variáveis para armazenar valores que não são argumentos.

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

As variáveis que são usadas como parâmetros para as funções são declaradas na assinatura da função e são inicializadas a cada chamada da função com os argumentos especificados na chamada.

A forma geral para declaração de variável é

```
nome: tipo = valor
```

onde `tipo` e `valor` são opcionais, mas pelos menos um deve ser especificado.

```
>>> 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 essas instruções de **forma sequencial**, uma linha após a outra, por isso, a ordem das 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 referenciado 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 vim, o valor armazenado na célula de memória associada com b, que é 20, é exibido.

Variáveis locais

Uma **variável local** é declarada no escopo (“dentro”) de uma função. Elas são criadas quando a linha que estão declaradas são executadas e deixam de existir quando a função devolve a resposta.

As variáveis locais são usadas para armazenar valores intermediários durante a execução da função.

Vamos usar variáveis locais para deixar o código da função `novo_seculo` mais legível

Revisão novo_seculo

```python
def novo_seculo(data: str) -> bool:
    return data[:2] == '01' and \
           data[3:5] == '01' and \
           data[8:] == '00'
```

```python
def novo_seculo(data: str) -> bool:
    dia: str = data[:2]
    mes: str = data[3:5]
    decada: str = data[8:]
    return dia == '01' and \
           mes == '01' and \
           decada == '00'
```

Qual código deixar a intenção mais clara? O que usa variáveis locais auxiliares.

Podemos simplificar? Sim, podemos omitir o tipo das variáveis.

```python
def novo_seculo(data: str) -> bool:
    dia = data[:2]
    mes = data[3:5]
    decada = data[8:]
    return dia == '01' and mes == '01' and decada == '00'
```

Nós vimos anteriormente que o Python executa as instruções de forma sequencial, uma linha após a outra. No entanto, quando uma função é chamada a execução é desviada para o início da função, e quando a função finaliza o execução volta para onde estava antes da chamada da função.

Execução passo a passo

```
1  def quadrado(a: float) -> float:
2      return a * a
3
4  def raiz(a: float) -> float:
5      return a ** 0.5
6
7  def hipotenusa(a: float, b: float) -> float:
8      a2 = quadrado(a)
9      b2 = quadrado(b)
10     return raiz(a2 + b2)
11
12 hipotenusa(3.0, 4.0)
```

Em qual ordem as linhas do programa são executadas pelo Python? 12, 8, 2, 8, 9, 2, 9,10, 5, 10, 12. (Feito em sala)

O que é exibido na tela? Nada!

Diferenças entre a janela de interações e a de edição de código

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 começara a ser exibida no início da próxima linha.

Veremos mais detalhes em outro momento.

Erros

Vamos parar um pouco e pensar sobre erros.

Já encontramos alguns tipos de erros enquanto fazíamos os nosso exemplos:

- O programa não 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 (recuo) 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.

```python
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.

```python
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)
```

Erros de execução

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 (falhar)
- 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.