

COMPUTAÇÃO 1 – PYTHON

AULA 2 TEORICA

SILVIA BENZA

SILVIABENZA@GMAIL.COM

FUNÇÕES

LEMBRAM DA AULA PRATICA PASSADA?

Exercício: Calcule a área da coroa circular (anel) formada por dois círculos de raios r_1 e r_2 ($r_1 > r_2$ e $\pi = 3.14$).

LEMBRAM DA AULA PRATICA PASSADA?

Exercício: Calcule a área da coroa circular (anel) formada por dois círculos de raios r_1 e r_2 ($r_1 > r_2$ e $\pi = 3.14$).

#Função Coroa Circular

```
def coroa(r1,r2):  
    return (3.14*r1**2) - (3.14*r2**2)
```

**ALGUÉM TESTOU O
PYTHON TUTOR??**

CALCULO DA COROA NO PYTHON TUTOR

```
1 #Função Coroa Circular
2
→ 3 def coroa(r1,r2):
4     return (3.14*r1**2) - (3.14*r2**2)
5
6
7
8 # Calculando a Coroa Circular formada
9 # pelos círculos de raios 3 e 2
10
11 coroa(3,2)
```

[Edit code](#)



[**<< First**](#)

[**< Back**](#)

Step 1 of 5

[**Forward >**](#)

[**Last >>**](#)

→ line that has just executed

→ next line to execute

Frames

Objects

CALCULO DA COROA NO PYTHON TUTOR

```
1 #Função Coroa Circular
2
3 def coroa(r1,r2):
4     return (3.14*r1**2) - (3.14*r2**2)
5
6
7
8 # Calculando a Coroa Circular formada
9 # pelos círculos de raios 3 e 2
10
11 coroa(3,2)
```

[Edit code](#)



[**<< First**](#)

[**< Back**](#)

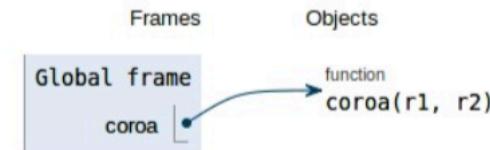
Step 2 of 5

[**Forward >**](#)

[**Last >>**](#)

→ line that has just executed

→ next line to execute



CALCULO DA COROA NO PYTHON TUTOR

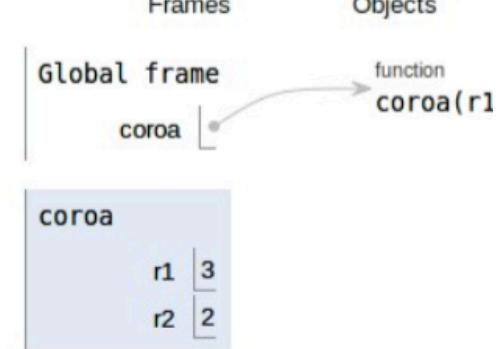
```
1 #Função Coroa Circular
2
3 def coroa(r1,r2):
4     return (3.14*r1**2) - (3.14*r2**2)
5
6
7
8 # Calculando a Coroa Circular formada
9 # pelos círculos de raios 3 e 2
10
11 coroa(3,2)
```

[Edit code](#)



[<< First](#) [< Back](#) Step 3 of 5 [Forward >](#) [Last >>](#)

→ line that has just executed
→ next line to execute



CALCULO DA COROA NO PYTHON TUTOR

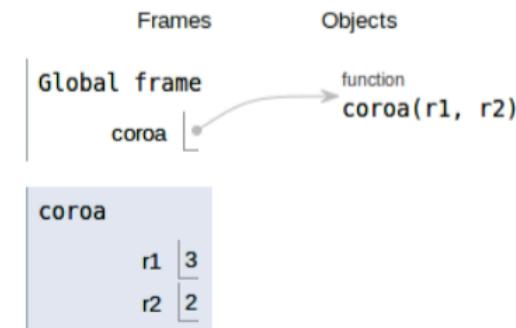
```
1 #Função Coroa Circular
2
3 def coroa(r1,r2):
4     return (3.14*r1**2) - (3.14*r2**2)
5
6
7
8 # Calculando a Coroa Circular formada
9 # pelos círculos de raios 3 e 2
10
11 coroa(3,2)
```

[Edit code](#)



[<< First](#) [< Back](#) Step 4 of 5 [Forward >](#) [Last >>](#)

- line that has just executed
- next line to execute



CALCULO DA COROA NO PYTHON TUTOR

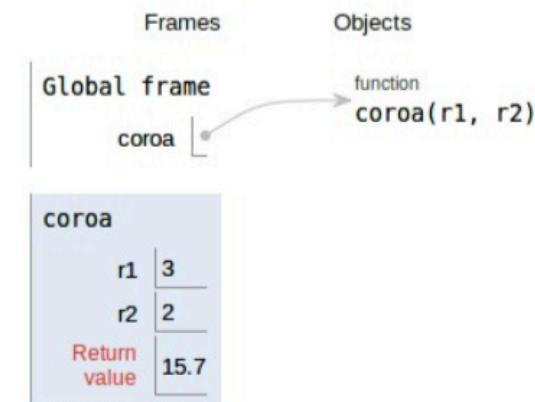
```
1 #Função Coroa Circular
2
3 def coroa(r1,r2):
4     return (3.14*r1**2) - (3.14*r2**2)
5
6
7
8 # Calculando a Coroa Circular formada
9 # pelos círculos de raios 3 e 2
10
11 coroa(3,2)
```

[Edit code](#)

<< First < Back Step 5 of 5 Forward > Last >>

→ line that has just executed

→ next line to execute



AINDA NA AULA PRATICA PASSADA

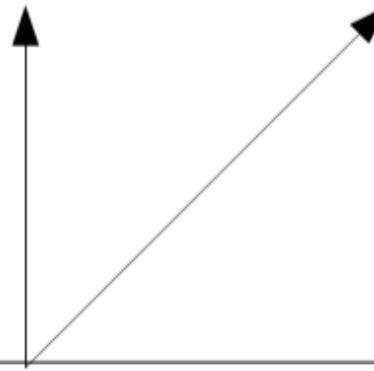
Exercício: Calcule a área de um círculo de raio r_1 .

AINDA NA AULA PRATICA PASSADA

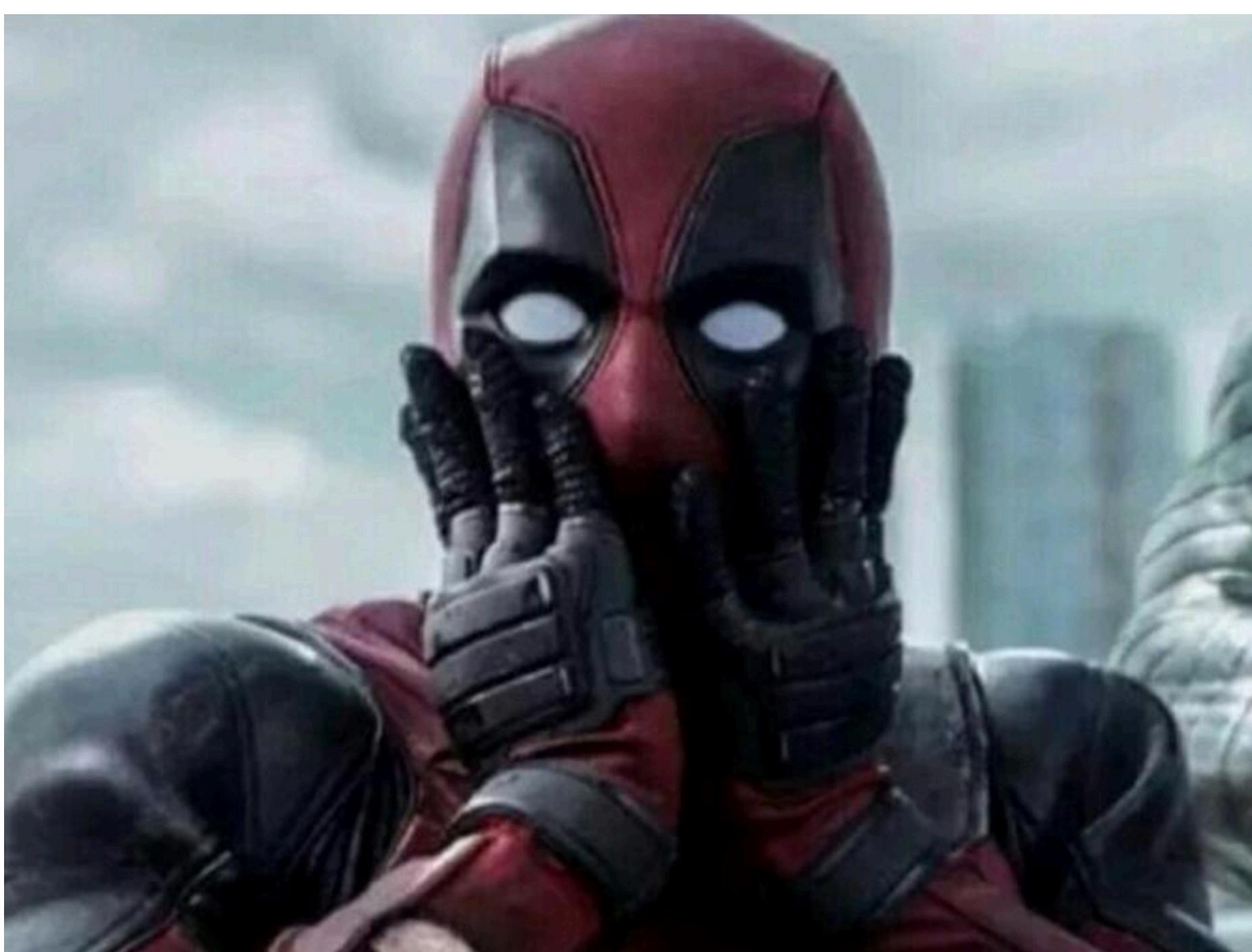
Exercício: Calcule a área de um círculo de raio r1.

```
def areac(r1):  
    return 3.14*r1**2
```

```
def coroa(r1,r2):  
    return (3.14*r1**2) - (3.14*r2**2)
```



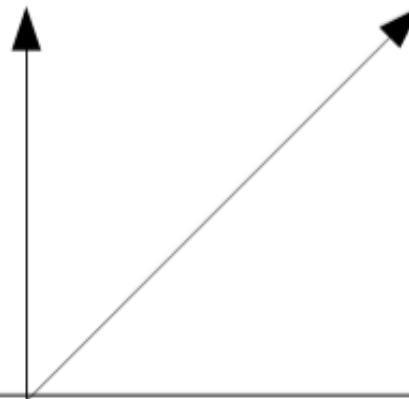
```
def areac(r1):  
    return 3.14*r1**2
```



POSSO CHAMAR UMA FUNÇÃO DENTRO DE OUTRA!

```
def coroa(r1,r2):  
    return areac(r1) - areac(r2)
```

```
def areac(r1):  
    return 3.14*r1**2
```



UE, COMO?

```
def coroa(r1,r2):  
    return areac(r1) - areac(r2)
```

```
def areac(r1):  
    return 3.14*r1**2
```

coroa(3,2) ↴

)

```
def coroa(r1,r2):  
    return areac(r1) - areac(r2)
```

```
def areac(r1):  
    return 3.14*r1**2
```

coroa(3,2)

areac(3) – areac(2)

```
def coroa(r1,r2):  
    return areac(r1) - areac(r2)
```

```
def areac(r1):  
    return 3.14*r1**2
```

coroa(3,2)

areac(3) – areac(2)

areac(3)
3.14*3**2=
=28.26
return 28.26

```
def coroa(r1,r2):  
    return areac(r1) - areac(r2)
```

```
def areac(r1):  
    return 3.14*r1**2
```

coroa(3,2)

areac(3) – areac(2)

areac(3)
3.14*3**2=
=28.26
return 28.26

areac(2)
3.14*2**2=
=12.56
return 12.56

```
def coroa(r1,r2):  
    return areac(r1) - areac(r2)
```

```
def areac(r1):  
    return 3.14*r1**2
```

```
coroa(3,2)  
areac(3) – areac(2) 15.70
```

```
areac(3)  
3.14*3**2=  
=28.26  
return 28.26
```

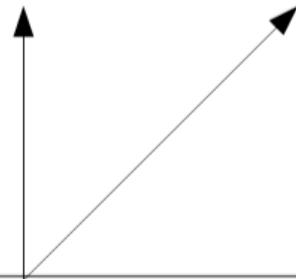
```
areac(2)  
3.14*2**2=  
=12.56  
return 12.56
```

E CONTINUAR CHAMANDO OUTRA!!!

```
def coroa(r1,r2):  
    return areac(r1) - areac(r2)
```

```
def areac(r1):  
    return 3.14*r1**2
```

```
def quadrado(x):  
    return x**2
```



PI É BASTANTE USADO, VAMOS CHAMAR TAMBÉM!

```
def coroa(r1,r2):  
    return areac(r1) - areac(r2)
```

```
def areac(r1):  
    return 3.14*quadrado(r1)
```

```
def pi():  
    return 3.14
```

```
def quadrado(x):  
    return x**2
```

A função pi definida é uma função constante

OK, PODEMOS PARAR POR AQUI

```
def coroa(r1,r2):  
    return areac(r1) - areac(r2)
```

```
def areac(r1):  
    return pi()*quadrado(r1)
```

```
def pi():  
    return 3.14
```

```
def quadrado(x):  
    return x**2
```

A função pi definida é uma função constante

UM POUCO DE TEORIA ABSTRAÇÃO

É uma técnica de programação que nos permite pensar num problema em diversos níveis

A idéia é que quando estamos pensando num problema macroscopicamente, não estamos preocupado com minúcias

Dividir para conquistar:

- Um problema é dividido em diversos sub-problemas
- As soluções dos sub-problemas são combinadas numa solução do problema maior



PROGRAMAÇÃO ESTRUTURADA

É uma disciplina de programação que incorpora o princípio de “Dividir para Conquistar”

Programas são divididos em sub-programas

- Cada sub-programa é invocado por meio de um identificador e uma lista de entradas
 - Permite especificar como um problema pode ser resolvido *em geral*
 - O mesmo sub-programa pode ser invocado para resolver diversos problemas de mesma natureza mas com valores específicos diferentes
- Os resultados computados por um sub-programa pode ser combinado com os de outros sub-programas

DEFININDO FUNÇÕES

Em Python, sub-programas têm o nome de **funções**

Formato geral:

```
def nome (arg, arg ,..., arg):
```

```
    comando
```

```
    comando
```

```
    return expressão
```

Onde:

- *nome* é o nome da função
- *args* são especificações de argumentos da função
 - Uma função pode ter 0, 1 ou mais argumentos
- *comandos* contêm as instruções a ser executadas quando a função é invocada
- **return** indica o valor a ser devolvido pela função,
 - Onde a *expressão* é opcional e designa o valor a ser retornado

ARGUMENTOS DE FUNÇÕES

Argumentos (ou parâmetros) são como variáveis que recebem seus valores iniciais do chamador

Essas variáveis, assim como outras definidas dentro da função são ditas *locais*, isto é, só existem no lugar onde foram definidas

- Ao retornar ao ponto de chamada, as variáveis locais são descartadas

Se uma função define n argumentos, valores para todos eles devem ser passados pelo chamado

- Exceção: argumentos com valores default

ARGUMENTOS DE FUNÇÕES

```
>>> def f(x):  
    return x*x
```

```
>>> print f(10)
```

```
100
```

```
>>> print x
```

```
....
```

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

```
>>> print f()
```

```
....
```

```
TypeError: f() takes exactly 1 argument (0 given)
```

ARGUMENTOS DEFAULT

É possível dar valores *default* a argumentos

- Se o chamador não especificar valores para esses argumentos, os defaults são usados

Formato:

`def nome (arg1=default1, ..., argN=defaultN)`

Se apenas alguns argumentos têm default, esses devem ser os últimos

- Se não fosse assim, haveria ambigüidade na passagem de argumentos

ARGUMENTOS DEFAULT

```
>>> def f(nome,saudacao="Oi",pontuacao="!!"):  
        return saudacao+","+nome+pontuacao  
>>> print f("Joao")  
Oi,Joao!!  
>>> print f("Joao","Parabens")  
Parabens,Joao!!  
>>> print f("Joao","Ah","...")  
Ah,Joao...
```

PASSANDO ARGUMENTOS COM NOMES

É possível passar os argumentos sem empregar a ordem de definição desde que se nomeie cada valor passado com o nome do argumento correspondente

Ex.:

```
>>> def f(nome,saudacao="Oi",pontuacao="!!"):  
    return saudacao+","+nome+pontuacao  
  
>>> print f(saudacao="Valeu",nome="Joao")  
Valeu,Joao!!
```

VARIÁVEIS E ATRIBUIÇÃO

AGORA PODE USAR?

THANK GOD!



FINALLY

VARIÁVEIS

São nomes dados a áreas de memória

- Nomes podem ser compostos de algarismos, letras ou _
- O primeiro caractere não pode ser um algarismo
- Palavras reservadas (if, while, etc) são proibidas

Servem para:

- Guardar valores intermediários
- Construir estruturas de dados

Uma variável é modificada usando o comando de atribuição:

Var = expressão

É possível também atribuir a várias variáveis simultaneamente:

var1,var2,...,varN = expr1,expr2,...,exprN

VARIÁVEIS

Variáveis são criadas dinamicamente e destruídas quando não mais necessárias, por exemplo, quando saem fora de escopo (veremos isso mais tarde)

O *tipo* de uma variável muda conforme o valor atribuído, i.e., int, float, string, etc.

Ex.:

```
>>> a ="1"  
>>> b = 1  
>>> a+b
```

Traceback (most recent call last):

 File "<stdin>", line 1, in ?

 TypeError: cannot concatenate 'str' and 'int' objects

VARIÁVEIS E ATRIBUIÇÃO

Variáveis são usadas para guardar dados intermedidários nas funções

```
# olafulano versao 1
def olafulano(fulano):
    return "Olá " + fulano
```

```
# olafulano versao 2
def olafulano(fulano):
    nome = fulano
    return "Olá " + nome
```



*nome é uma variável
do programa !*

MEMÓRIA

nome → “Carlos”

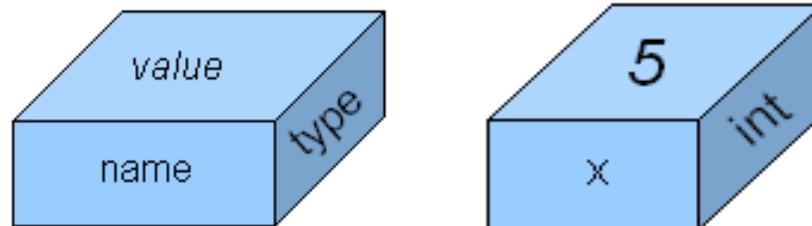
VARIÁVEIS E ATRIBUIÇÃO

Variável

- É uma maneira simbólica de fazer referência a dados armazenados na memória do computador.

Toda variável engloba os seguintes aspectos, semelhantes aos parâmetros de uma função:

- Nome (identificador): é a representação simbólica da variável, que será utilizada pelo programador para fazer referência aos dados que ela armazena.
- Valor: o que de fato está armazenado.
- Tipo: o tipo de dado que está armazenado.



NOMES DE VARIÁVEIS

Letras, números e *underline* (não começar por números)

- minhaVariavel = 1
- minha_variavel = 1
- minhaVariavel2 = 2
- minha_variavel_2 = 2

Dica: em programas muito grandes e complexos, escolha (se possível) nomes que *descrevam* o significado da variável. Exceto em programas muito simples ou exemplos didáticos, evite nomes genéricos como “x”, “y”, “a”, etc.

ATRIBUIÇÃO

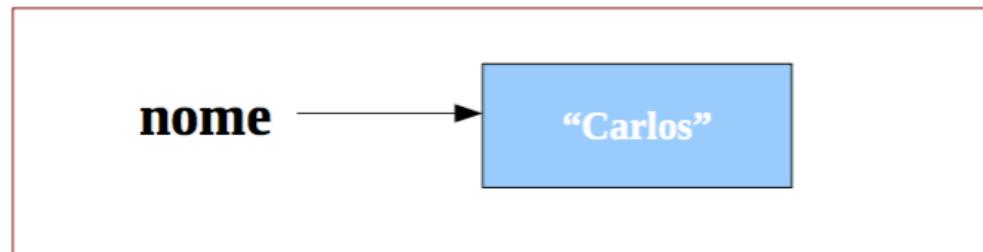
O símbolo `=` é usado para atribuir um valor a uma variável.

```
...
nome = "Carlos"
return "Olá" + nome
```

`var = valor`

`var1, var2,..., varN = valor1, valor2, ..., valorN`

MEMÓRIA



ATRIBUINDO VALORES A VARIÁVEIS

No interpretador python:

```
>>>a=1 # atribuo o valor 1 a variável a
>>> a    # dá o valor armazenado em a
1
>>>a=2*a   # armazeno na variável a o valor que está em a multiplicado por 2
>>>a        # dá o valor armazenado em a
2
```

ATTRIBUIÇÃO MÚLTIPLA

```
>>> a,b,c = 1,2,3
```

```
>>> a
```

1

```
>>> b
```

2

```
>>> c
```

3

COMO CRIAR E USAR UMA VARIÁVEL?

Uma variável é criada com um comando de atribuição:
variável = valor

Um *Alias* é um identificador que se refere a uma variável existente.

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

```
>>> y = x
```

“y” é um *alias* para a variável “x”. Portanto, possui o mesmo valor e aponta para o mesmo endereço de “x”.

ALIAS (PSEUDÔNIMO)

O que acontece se atribuirmos um novo valor a “x”?

```
>>> x = 5
```

```
>>> y
```

```
4
```

“y” permaneceu inalterada!!

O que aconteceu foi algo bastante sutil (e bizarro): “x” é do tipo *int*, que é um tipo *imutável* (falaremos sobre isso mais tarde).

- Ao escrevermos “x = 5”, em vez de modificar a variável “x” já existente, simplesmente criamos outra variável com o nome “x” e atribuímos a ela o valor 5. A variável “x” antiga é jogada fora.
- Como “y” era um *alias* para a variável “x” antiga, seu valor permaneceu inalterado.

QUAL A DIFERENÇA?

```
def testea( ):  
    a = 10  
    a,b=3*a,a  
    return a,b
```

```
def testea2( ):  
    a=10  
    a=3*a  
    b = a  
    return a,b
```

TIPO

Python é uma linguagem **dinamicamente tipada** ou **fracamente tipada**.

O tipo é atribuído de acordo com o valor atribuído à variável.
Não é necessário *declarar previamente* o tipo.

```
>>> x = 4  
  
>>> type(x)  
<type 'int'>
```

O tipo de uma variável pode mudar depois de alguma operação ou nova atribuição.

```
>>> x = complex(x)  
  
>>> type(x)  
<type 'complex'>
```

ESCOPO

Onde a variável existe e onde ela deixa de existir.

As variáveis definidas dentro de uma função são ditas **variáveis locais**, porque não podem ser acessadas fora da função.

```
def produtoSomaDiferenca(a, b):
```

```
    x = a + b
```

```
    y = a - b
```

```
    return x*y
```

- As variáveis “x” e “y” são locais, pois só *existem* dentro da função. Depois que a função é executada, elas são destruídas.
- Dizemos que a função é o escopo de “x” e “y”.
- Tentar chamá-las fora da função ocasionaria um erro.

TIPOS DE DADOS: NUMÉRICOS

TIPOS DE DADOS?

São categorias de valores que são processados de forma semelhante

Por exemplo, números inteiros são processados de forma diferente dos números de ponto flutuante (decimais) e dos números complexos

Tipos primitivos: são aqueles já embutidos no núcleo da linguagem

- Simples: números (int, long, float, complex) e cadeias de caracteres (strings)
- Compostos: listas, dicionários, tuplas e conjuntos

Tipos definidos pelo usuário: são correspondentes a classes (orientação objeto)

TIPOS DE DADOS?

São categorias de valores que são processados de forma semelhante

Por exemplo, números inteiros são processados de forma diferente dos números de ponto flutuante (decimais) e dos números complexos

Tipos primitivos: são aqueles já embutidos no núcleo da linguagem

- **Simples:** números (`int`, `long`, `float`, `complex`) e cadeias de caracteres (`strings`)
- **Compostos:** listas, dicionários, tuplas e conjuntos

Tipos definidos pelo usuário: são correspondentes a classes (orientação objeto)

PYTHON COMO CALCULADORA

```
>>> 10
10
>>> # Um comentário é precedido do caractér "#"
... # Comentários são ignorados pelo interpretador
... 10+5
15
>>> 10-15 # Comentários podem aparecer também após código
-5
>>> 10*3
30
>>> 10/3
3
>>> 10/-3 # Divisão inteira retorna o piso
-4
>>> 10%3 # Resto de divisão inteira simbolizado por %
1
```

NÚMEROS

Há vários tipos numéricos que se pode usar em python

- **Int**: números inteiros de *precisão fixa*
 - 1 , 2 ,15,-19
- **Long**: números inteiros de *precisão arbitrária*
 - 1L , 10000L , -9999999L
- **Floats**: números racionais de *precisão variável*
 - 1.0 , 10.5 , -19000.00005 , 15e-5
- **Complex**: números complexos
 - 1+1j , 20j , 1000+100J

NÚMEROS INTEIROS

Os ints têm precisão fixa ocupando tipicamente uma palavra de memória

- Em PC's são tipicamente representados com 32 bits (de -2³¹-1 a 2³²)

Os números inteiros de precisão arbitrária (longs) são armazenados em tantas palavras quanto necessário

- Constantes do tipo long têm o sufixo L ou l
- Longs são manipulados bem mais lentamente que ints
- Quando necessário, cálculos usando ints são convertidos para longs

NÚMEROS INTEIROS

```
>>> a=2**30 # Potenciação
```

```
>>> a
```

```
1073741824
```

```
>>> b=a*1000
```

```
>>> b
```

```
1073741824000L
```

```
>>> b/1000
```

```
1073741824L
```

NÚMEROS INTEIROS

Constantes podem ser escritas com notação idêntica à usada em C

- Hexadecimal: preceder dígitos de 0x
- Octal: preceder dígitos de 0
- Ex.:

```
>>> 022
```

```
18
```

```
>>> 0x10
```

```
16
```

```
>>> 0x1f
```

```
31
```

NÚMEROS DE PONTO FLUTUANTE

São implementados como os double's da linguagem C – tipicamente usam 2 palavras

Constantes têm que possuir um ponto decimal ou serem escritas em notação científica com a letra “e” (ou “E”) precedendo a potência de 10

Ex:

```
>>> 10 # inteiro  
10  
>>> 10.0 # ponto flutuante  
10.0  
>>> 99e3  
99000.0  
>>> 99e-3  
0.09900000000000005
```

NÚMEROS COMPLEXOS

Representados com dois números de ponto flutuante: um para a parte real e outro para a parte imaginária

Constantes são escritas como uma soma sendo que a parte imaginária tem o sufixo j ou J

Ex.:

```
>>> 1+2j  
(1+2j)  
>>> 1+2j*3  
(1+6j)  
>>> (1+2j)*3  
(3+6j)  
>>> (1+2j)*3j  
(-6+3j)
```

MÓDULOS

Módulos Python

Funções que realizam tarefas comuns tais como cálculos matemáticos, manipulações de strings, manipulação de caracteres , programação Web, programação gráfica, etc.

Bibliotecas: coleção de módulos

MÓDULO *MATH*

Módulo que permite que o programador realize certos cálculos matemáticos.

Para usar uma função que está definida em um módulo, *primeiro* o programa deve importar o módulo usando o comando import

```
>>> import math
```

Após ter importado o módulo, o programa pode chamar as funções daquele módulo da seguinte forma:

NomeDoModulo.NomeDaFunção(arg0,...,argn)

Exemplo

```
>>> math.sqrt(81)  
9.0
```

MÓDULO *MATH*

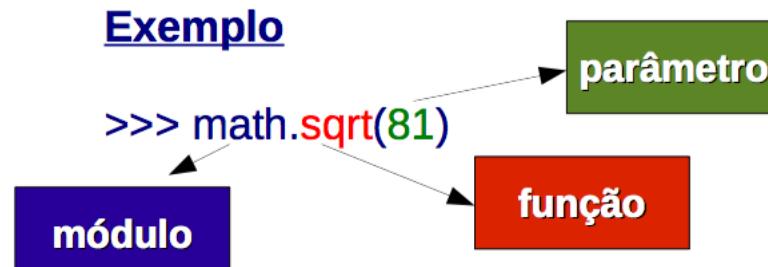
Módulo que permite que o programador realize certos cálculos matemáticos.

Para usar uma função que está definida em um módulo, *primeiro* o programa deve importar o módulo usando o comando `import`

```
>>> import math
```

Após ter importado o módulo, o programa pode chamar as funções daquele módulo da seguinte forma:

```
NomeDoModulo.NomeDaFunção(arg0,...,argn)
```



MÓDULO *MATH*

Podemos importar parte dos módulos:

`from math import * # importa todos os elementos do
módulo math`

`from math import nome-função # importa apenas a
função nome-função`

Exemplos

```
>>> from math import *
```

```
>>> from math import sin
```

MÓDULO *MATH*

```
>>> import math  
>>> sin(30)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'sin' is not defined  
>>> math.sin(30)  
-0.9880316240928618
```

MÓDULO *MATH*

```
>>> import math
>>> sin(30)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> math.sin(30)
-0.9880316240928618
>>> import math
>>> sin(radians(30))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> math.sin(radians(30))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'radians' is not defined
>>> math.sin(math.radians(30))
0.4999999999999994
```

MÓDULO *MATH*

```
>>> from math import sin  
>>> sin(30)  
-0.9880316240928618  
>>> sin(radians(30))  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'radians' is not defined
```

```
>>> from math import *  
>>> sin(radians(30))  
0.4999999999999994
```

MÓDULO

Para ter acesso aos módulos do python:

```
>>> help()  
help> modules
```

Para saber sobre um módulo específico, basta digitar o nome:

```
help> math
```

Help on built-in module math:

NAME
math

FILE
(built-in)

DESCRIPTION

This module is always available. It provides access to the mathematical functions defined by the C standard.

FUNCTIONS
acos(...)
acos(x)

Return the arc cosine (measured in radians) of x.

```
>>> import math  
>>> help(math.cos)
```

Help on built-in function cos in module math:

cos(...)
cos(x)

Return the cosine of x (measured in radians).

Pressiona-se “q” para retornar ao interpretador.

COMPUTAÇÃO 1 - PYTHON

AULA 2 TEÓRICA

SLIDES BASEADOS NOS TRABALHOS:
AULA2 TEÓRICA DO DCC UFRJ
AULA DE FUNÇÃO E DE TIPOS DE DADOS DO
CLAUDIO ESPERANÇA DO PESC