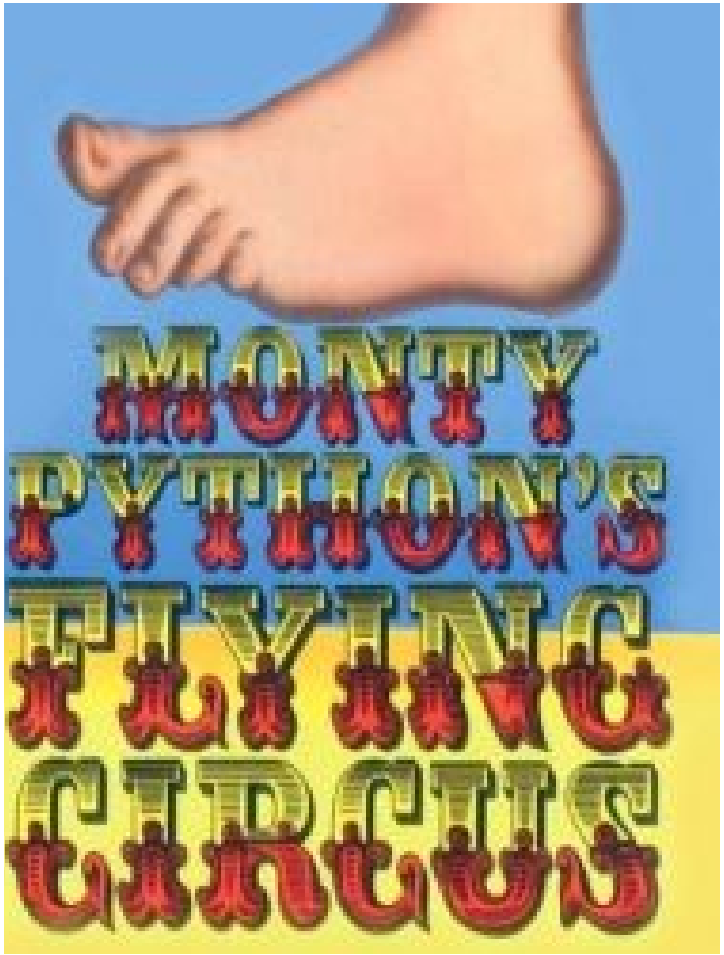
Python

Guilherme Baldo Carlos
Nickolas Oliveira Soares
Nicolas Garcia Cavalcante

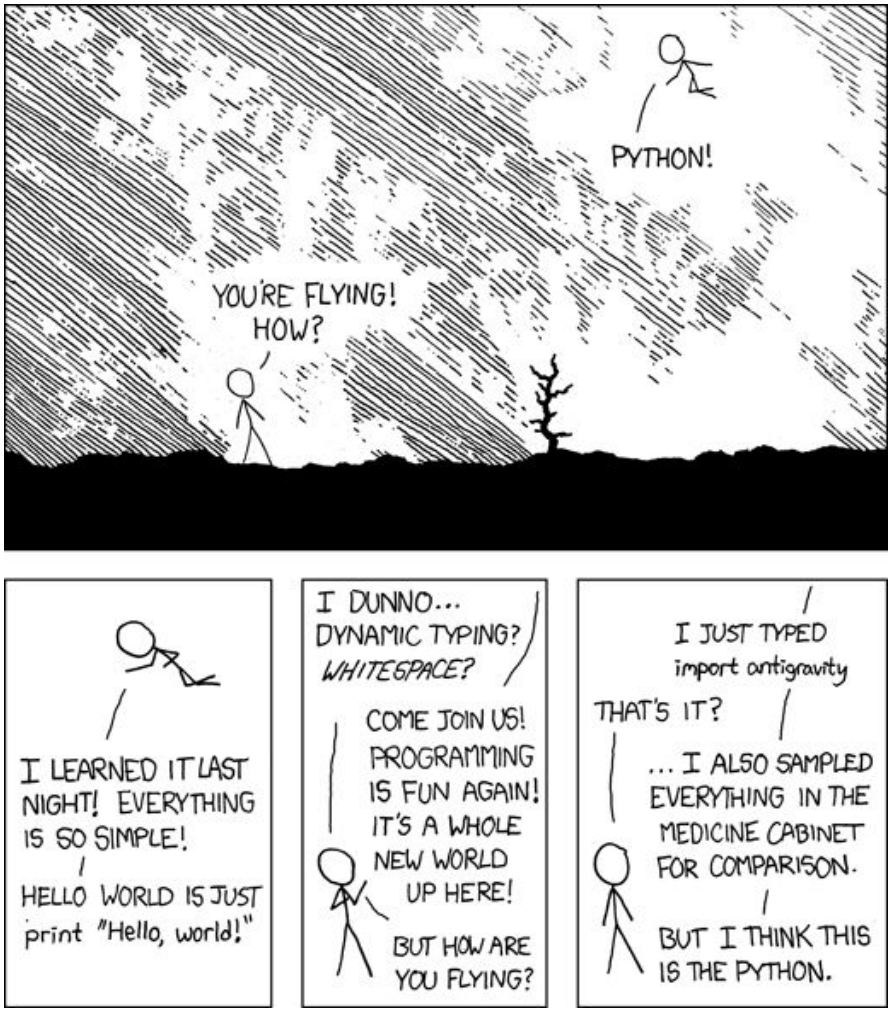
Histórico

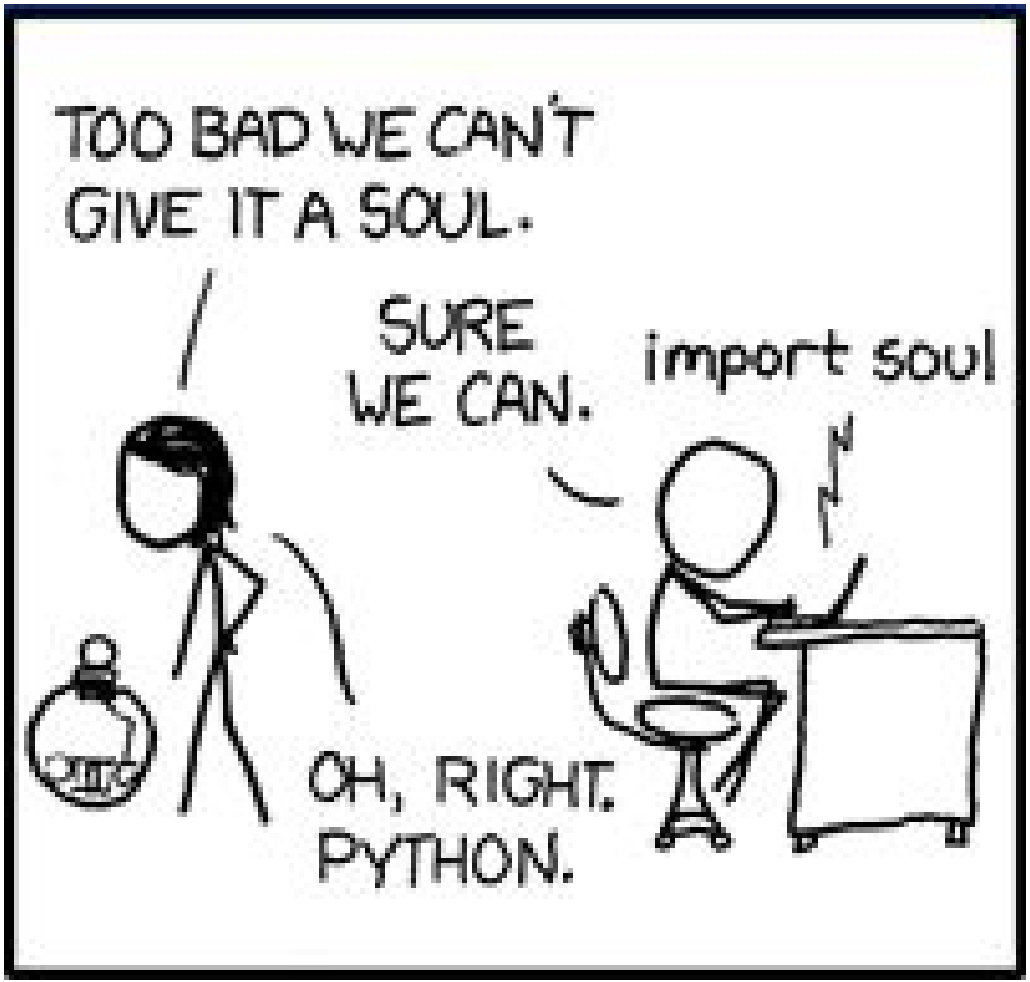
- Python foi criada no início dos anos 90 por Guido van Rossum no CWI (Stichting Mathematisch Centrum) na Holanda. Ela foi criada para ser sucessora da linguagem ABC.
- Em 2001 foi criada a Python Software Foundation (PSF), organização sem fins lucrativos que mantêm os direitos de propriedade intelectual sobre o Python.
- Python é Open-source



Visão geral

- Interpretada
- Sintaxe legível e de fácil aprendizagem
- Interativa
- Portável
- Funcional, estruturada e orientada a objetos.
- Possui tipagem dinâmica
- Pode ser facilmente integrado com outras linguagens como C++ e Java.


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.


TOO BAD WE CAN'T GIVE IT A SOUL.
SURE WE CAN.
import soul
OH, RIGHT. PYTHON.

Compilação e interpretação

- O código fonte é traduzido para bytecode (formato binário com instruções para o interpretador). O bytecode é multiplataforma e garante a portabilidade.
- Por padrão o interpretador compila o código e armazena o bytecode em disco, para evitar a necessidade de recompilar na próxima execução, diminuindo tempo de carga.
- O interpretador executa o bytecode.

Sintaxe

- Python foi desenvolvido para ser uma linguagem de boa legibilidade.
- Utiliza palavras ao invés de símbolos para algumas operações (and, or …).
- A linguagem utiliza indentação para separação de blocos de código. Assim, a indentação é obrigatória em python.
- # marca o início de um comentário de uma linha e aspas triplas são utilizados para comentários de múltiplas linhas.

```
# comentário
if True or False:
        print(“Hello World 1”)
else:
        print(“Hello World 2”)
print(“Bye World”)
```

Saída:

```
Hello World 1
Bye World
```

Identificadores

- Python é case-sensitive.
- Os caracteres válidos para identificadores são as letras maiúsculas e minúsculas (A a Z, a a z), o underscore (_) ou, com exceção do primeiro caractere, dígitos (0 a 9).
- Identificadores não tem limite de tamanho.

Palavras reservadas ou palavras-chave

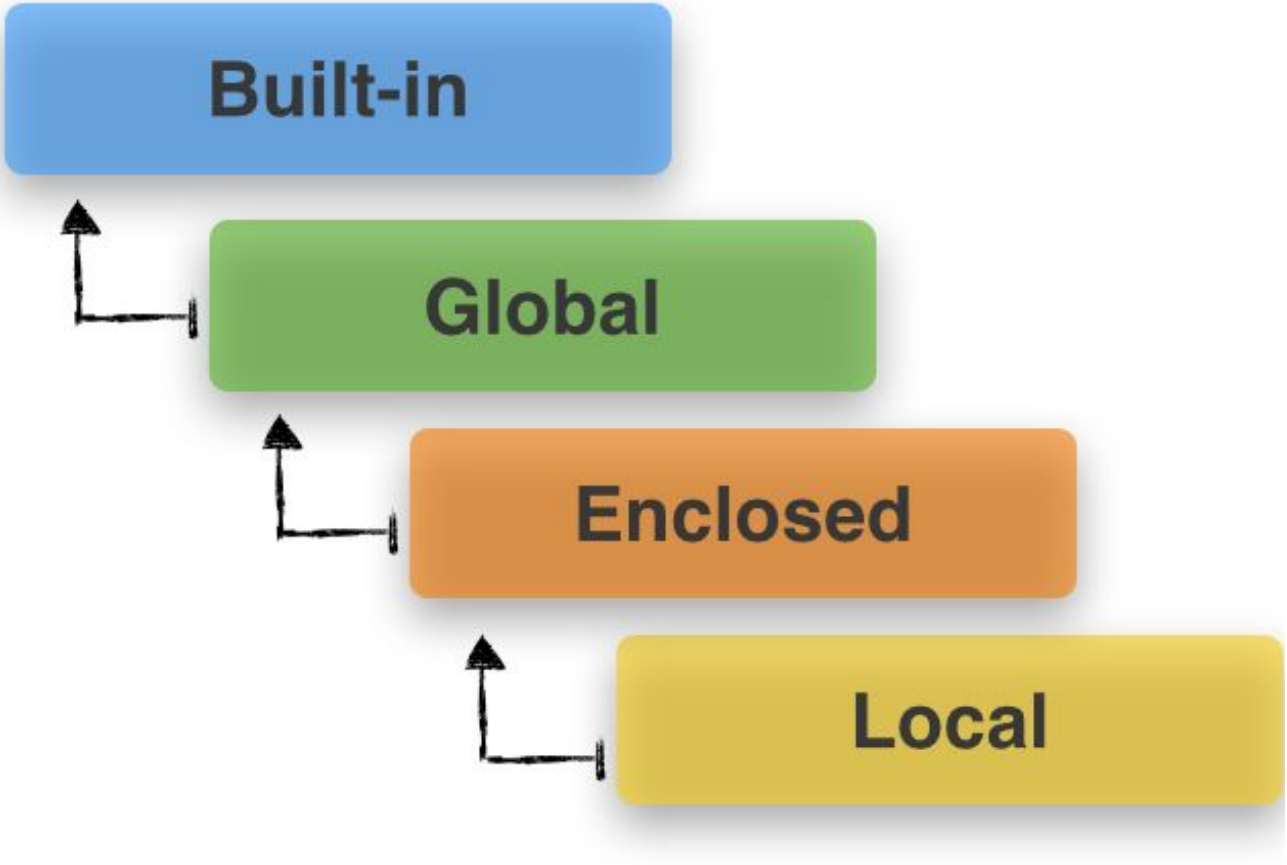
False	class	finally	is	return	None	continue	for	
lambda	try	True	def	from	nonlocal	while	and	
del	global	not	with	as	elif	if	or	
yield	assert	else	import	pass	break	except	in	
raise								

Amarrações

- Em Python tudo é um objeto.
- Nomes fazem referência a objetos e as amarrações entre estes nomes e objetos são feitas de forma dinâmica (com exceção de palavras reservadas).
- A medida em que nomes são utilizados na execução do código amarrações novas são feitas.

Escopo

- Escopo estático.
- Quando um nome é utilizado em um bloco de programa, a amarração utilizada é a definida no escopo mais próximo.
- Quando um nome não é encontrado em nenhum ambiente de amarração a exceção NameError é lançada. Se o nome foi encontrado mas ainda não aconteceu nenhum amarração a ele a exceção UnboundLocalError é lançada.


Built-in
Global
Enclosed
Local

```
a = “valor fora”

def f():

    a = “valor dentro”

    print(a)

f()

print(a)
```

Saída

valor dentro
valor fora

```
a = “valor fora”

def f():
    global a
    a = “valor dentro”
    print(a)

f()

print(a)
```

Saída

```
valor dentro
valor dentro
```

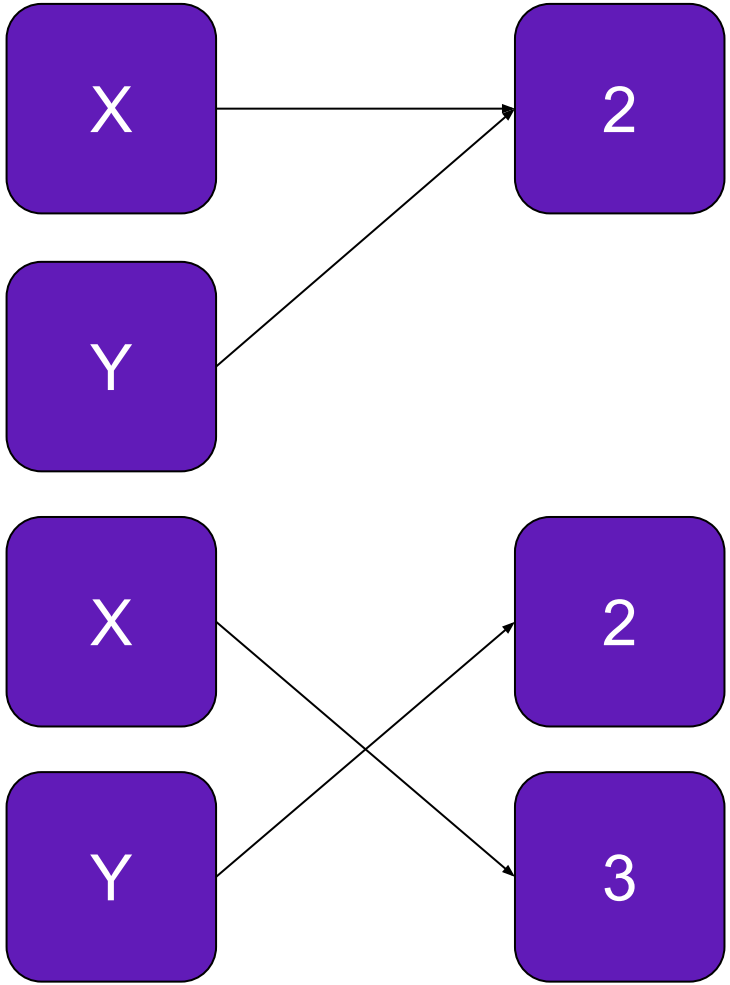
Tipos

- Utiliza tipagem dinâmica.
- É type-safe.
- Todos tipos são objetos.
- Mutáveis e imutáveis.

Tipos padrões

- Numéricos (são imutáveis)
  - Integers - Representam os números inteiros, seu tamanho depende da memória disponível. Complemento de 2 para negativos. x = 2
  - Booleans - Representam valores True e False, se comportam como os valores 1 e 0. x = True
  - Real - Representam os números reais, funcionam como números de precisão dupla no nível de máquina. x = 2.2
  - Complex - Representam números complexos, sendo implementados como um par de reais. x = complex(1,2)

```
x = 2

y = x

x = 3

print(x)

print(y)
```



Saída

3
2

Tipos padrões

- Strings - Sequência de valores codificados em Unicode que representam um texto. (imutável)

x = “Hello World” x = ‘Hello World’

x = ‘‘‘ Essa é uma string muito grande para caber em apenas uma mísera linha’’’

x = “a”

```
x = “Python”

print(x[0])

print(x[1:3])

print(x[3:])

print(x[-1])

x = “I like ” + x
```

Saída

```
P
yt
hon
n
```

Tipos padrões

- Tuples - Os itens em uma tupla podem ser de tipos arbitrários. (imutável)

x = (“dog”, True, 2.2) x = “a”, “b”, “c” x = () x = (50,)

Tipos padrões

- Lists - Similar a Tuples. (mutável)

x = [“dog”, True, 2.2] x = [“a”, “b”, “c”] x = [] x = [50]

```
x = [“dog”, True, 2.2]

x[1] = “cat”

print(x)

x[2] = [1, 2, 3]

print(x)
```

Saída

[‘dog’, ‘cat’, 2.2]
[‘dog’, ‘cat’, [1, 2, 3]]

Tipos padrões

- Dictionary - É um mapeamento. Representa um conjunto finito de objetos arbitrários que são indexados por keys. (mutável).

x = {“ham”: “yes”, “egg”: “yes”, “spam”: “no”}

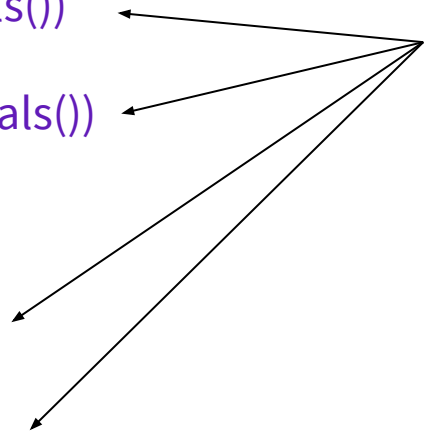
x = {1: “Hello”, 2: “World”, (1,2,3): “Uma tupla”, 2.2: “Até um real”}

```
x = {1: “Hello”, 2: “World”, (1,2,3): “Uma tupla”, 2.2: “Até um real”}

print(x[1])

print(x[(1,2,3)])

print(x[2.2])

print(x.keys())

print(x.values())
```

Saída

```
Hello
Uma tupla
Até um real
[1, 2, 2.2, (1, 2, 3)]
[“Hello”, “World”, “Até um real”, “Uma tupla”]
```

```
a = “valor fora”

def f():

        a = “valor dentro”

        print(locals())

        print(globals())

f()

print(locals())

print(globals())
```



Dicionários

Tipos padrões

- Set - Sequência unívoca (sem repetições) não ordenada. (mutável)
- Frozenset - Sequência unívoca (sem repetições) não ordenada. (imutável)
- Os dois tipos implementam operações de conjuntos, tais como: união, interseção e diferença.

```
s1 = set([1, 2, 3])

s2 = set([1, 2, 4])

print(s1.union(s2))

print(s1.difference(s2))

print(s1.intersection(s2))
```

Saída

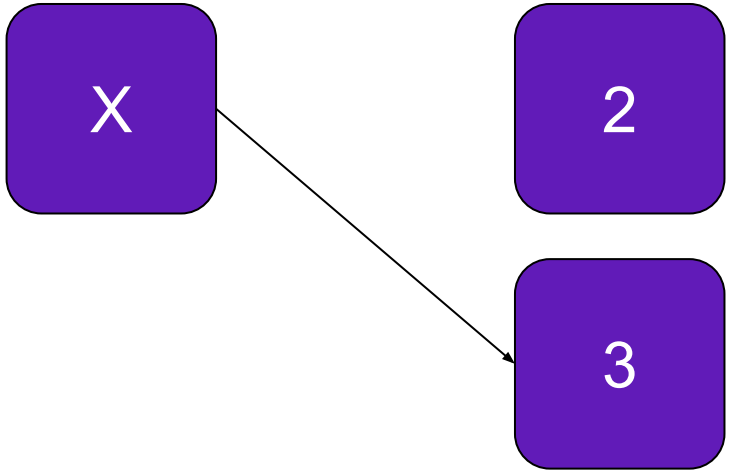
```
set([1, 2, 3, 4])
set([3])
set([1, 2])
```

Gerência de memória

- A gerência de memória em Python envolve uma região privada do monte, onde todos objetos e estruturas de dados estão alocados. Essa gerência é realizada internamente pelo gerente de memória do Python.
- Objetos não são explicitamente destruídos.
- Quando não existe mais nenhuma referência para determinado objeto ele é coletado pelo coletor de lixo.

x = 2

x = 3



I/O padrão

- Mostrando mensagem na tela

```
print(“Oi”)
```

- Lendo entrada do teclado

```
input(“Insira sua entrada”)
```

I/O Arquivos

```
file = open(“arquivo.txt”, “wb”)

file.write(“Hello World”)

file.close()

file = open(“arquivo.txt”, “r+”)

str = file.read()

print(str)

file.close()
```

Saída

Hello World

Serialização

- Python utiliza um módulo chamado pickle para realizar a serialização. (Existem outras ferramentas como Marshal e Shelve).

```
import pickle

favorite_color = {“lion”: “yellow”, “kitty”: “red”}

pickle.dump(favorite_color, open(“save.p”, “wb”))
```

```
import pickle

favorite_color = pickle.load(open(“save.p”, “rb”))
```

Operadores

- Aritméticos

+ - * / % ** //

- Relacionais

== != <> > < >= <=

- Atribuição

= += -= *= %= **= //=

Operadores

- Operadores bitwise

& | ^ ~ << >>

- Operadores lógicos

and or not

- Operadores de associação

in not in

- Operadores de identidade

is is not

Closure

```
def makeInc(x):
    def inc(y):
        return y + x
    return inc
incrementa5 = makeInc(5)
print(incrementa5(10))
print(incrementa5(1.2))
```

Saída

```
15
6.2
```

Operator	Description
`lambda`	Lambda expression
`if – else`	Conditional expression
`or`	Boolean OR
`and`	Boolean AND
`not x`	Boolean NOT
`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparisons, including membership tests and identity tests
`|`	Bitwise OR
`^`	Bitwise XOR
`&`	Bitwise AND
`<<`, `>>`	Shifts
`+`, `-`	Addition and subtraction
`*`, `@`, `/`, `//`, `%`	Multiplication, matrix multiplication division, remainder [5]
`+x`, `-x`, `~x`	Positive, negative, bitwise NOT
`**`	Exponentiation [6]
`await x`	Await expression
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	Subscription, slicing, call, attribute reference
`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	Binding or tuple display, list display, dictionary display, set display

precedência

Comandos

- Atribuições

`x = a + 2*y` (simples)

`a = b = 1` (múltipla)

`a += 3` (composta)

Não existe
o operador para
incremento ou
decremento unário
em python.

Comandos

- Condicionais

```
if x > 0:
    a = 0
elif x == 0
    a = 1
else
    a = 2
```

```
if x > y:
    n = 1
    n += 3
    if n < 5:
        n = 10
        m = n*2
```

Não existe switch em python.

Comandos

- Iterativos

```
count = 0
while (count < 9):
    print(str(count) + " é menor que 9)
    count += 1
```

Saída

```
0 é menor que 9
1 é menor que 9
2 é menor que 9
3 é menor que 9
4 é menor que 9
5 é menor que 9
6 é menor que 9
7 é menor que 9
8 é menor que 9
```

```
count  = 0

while (count < 5):

    print(str(count) + “ é menor que 5)

    count  += 1

else:

    print(str(count) + “não é menor que 5)
```

Saída

```
0 é menor que 5
1 é menor que 5
2 é menor que 5
3 é menor que 5
4 é menor que 5
5 não é menor que 5
```

```
for letter in “Python”:
    print(letter)

fruits = [“banana”, “apple”, “mango”]
for fruit in fruits:
    print(fruit)

for num in range(2,7):
    print(num)
else:
    print(“Good bye”)
```

Saída

```
P
y
t
h
o
n
banana
apple
mango
2
3
4
5
6
Good bye
```

```
for i in range(5):
    print(i)

for i in range(3, 6):
    print(i)

for i in range(4, 10, 2):
    print(i)

for i in range(0, -10, -2):
    print(i)
```

Saída

```
0
1
2
3
4
3
4
5
4
6
8
0
-2
-4
-6
-8
```

- break - Termina a execução do loop, e o código continua executando a primeira instrução após o loop. (não executa o código em else)
- continue - Força o código pular para próxima iteração do loop.
- pass - É utilizado quando um comando é requerido sintaticamente mas você não quer executar nada.

Python não possui goto.

```
for letter in “Python”:
    if(letter == “h”):
        pass
    print(letter)
```

Modularização

- Funções

```
def functionName(parameters):
    function_body
        return [expression]
```

```
def printMe(x):
    print(x)
    return

printMe(10)
printMe(“Hello”)
printMe([1, 2, 3])
```

Parâmetros

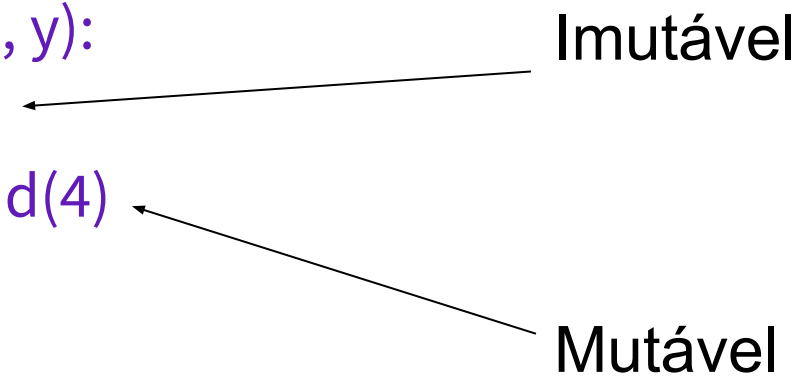
- Correspondência real-formal: posicional e por palavra-chave.
- Suporta valores default.
- Suporta passagem de parâmetros de número variável.
- Direção de passagem: Unidirecional de entrada variável.
- Mecanismo de passagem: Referência.
- Momento da passagem: Normal.

```
def juroSimples(C = 0, i = 1, t = 12):
    return (C*i*t)/100


M1 = juroSimples()
M2 = juroSimples(5000, 2, 24)
M3 = juroSimples(5000, 2)
M4 = juroSimples(5000, t = 24)
M5 = juroSimples(i = 3, C = 4000, t = 18)
```

```
def paralelo(*R):
    req = 0
    for r in R:
        req += (1 / r)
    req = (1 / req)
    return req


k = 1000
rt = paralelo(10*k, 56*k, 3.9*k)
```

Parâmetros variáveis.
Tratado como tupla.

```
def func(x, y):
    x = x + 1
    y.append(4)

x = 1
y = [1, 2, 3]
func(x, y)
print 'x:', x
print 'y:', y
```



Saída
1
[1, 2, 3, 4]

Funções anônimas

- A palavra chave lambda é utilizada para criar funções anônimas.

Sintaxe: lambda [arg1 [,arg2, …, argn]]:expression

```
sum = lambda arg1, arg2: arg1 + arg2


print(sum(10,20))

print(sum(2.2, 9))

print(sum(1,2))
```

Saída

```
30
11.2
3
```

Classes

```
class Pessoa:

    __ atributoPrivado = None

    atributoPublico = None

    __atributoStatic = list()

    def __init__(self, parametros ) :

        self.__atributoPrivado =
```

```
class Aluno(Pessoa):

    def __init__(self, parametrosPessoa, parametrosAluno):

        super(self, Pessoa).__init__(parametrosPessoa)

        self.__parametrosAluno = parametrosAluno
```

Classes

- atributo privado
  - apenas convenção
  - cria a mascara _<nome da classe>__<variável ”privada”>

```
class Classe:

    def __init__(self, a):

        self.__a = a
```

```
>>>objeto = Classe(“nome”)
>>>objeto._Classe__a
>>>nome
```

Módulos

- Qualquer arquivo de código fonte em Python é um módulo
- Podem ser importados em outros módulos, utilizando a palavra chave import
- Python dá suporte:
  - importação completa do módulo (import modulo)
  - importação relativa (from arq import x, y)
  - apelidos para módulos (from arq import x as z)

Módulos: importação

```
// funcao.py

def f(x):
    return x

def g(x):
    return x+1
```

```
>>> import funcao
>>> funcao.f(1), funcao.g(1)
>>> (1,2)
>>> from funcao import f
>>> f(1), funcao.g(1)
>>> (1,2)
>>> from funcao import *
```

Polimorfismo

Ad-hoc

- **Coerção :**
  Feita implicitamente pelo interpretador Python.

- **Sobrecarga:**
  Python não suporta sobrecarga de subprogramas.
  É permitida sobrecarga de operadores em classes definidas pelo programador

Sobrecarga de operadores

- A sobrecarga de operadores é feita a partir da definição de métodos especiais da classe.

  Exemplo:
  __add__ -> sobrecarga da adição
  __sub__ -> sobrecarga da subtração
  __mul__ -> sobrecarga da multiplicação

- Cabe ao programador implementar tais métodos.

Sobrecarga de operadores

```
>>> x = Tempo(2,‘h’)
>>> y = Tempo(60,‘m’)
>>> x+y
180
```

```
class Tempo():
    __unidades =  {‘h’: 60, ‘m’: 1, ‘s’: 1/60}
    def __init__(self,t,u):
        self.tempo = t
        self.unidade = u
    def converteMin(self):
        return self.tempo * Tempo.__unidades[self.unidade]
    def __add__(self,other):
        return self.converteMin() + other.converteMin()
```

Polimorfismo

Universal

- **Paramétrico**

Embutido na linguagem. Uma exceção é lançada caso a implementação do subprograma tente acessar atributos ou métodos que não pertencem ao objeto passado como parâmetro.

Polimorfismo

Universal

- **Inclusão**

  Característico da linguagem por ter orientação a objetos.

  Python oferece herança simples e múltipla de classes.

Polimorfismo

Herança simples e múltipla

Sintaxe:

```
class ClasseBase1:
# corpo da classe base 1
```

```
class ClasseDerivada1(ClasseBase1):
# corpo da classe derivada 1 [exemplo de herança simples]
```

```
class ClasseBase2:
# corpo da classe base 2
```

```
class ClasseDerivada2(ClasseBase1, ClasseBase2):
# corpo da classe derivada 2 [exemplo de herança múltipla]
```

Polimorfismo

Herança simples

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

class Aluno(Pessoa):
    def __init__(self,n,e=‘’):
        self.escola = e
        Pessoa.__init__(self,n)
```

```
class Professor(Pessoa):
    def __init__(self,n,e,f):
        self.escola = e
        self.formacao = f
        Pessoa.__init__(self,n)
```

Exceções

- Exceções indicam que algum tipo de condição excepcional ocorreu durante a execução do programa, logo, exceções estão associadas a condições de erro em tempo de execução.

Exceções

- Elas podem ser geradas/lançadas (quando é sinalizado que uma condição excepcional ocorreu) e capturadas (quando é feita a manipulação (tratamento) da situação excepcional, onde as ações necessárias para a recuperação de situação de erro são definidas.

Exceções

- Na prática, acontece que a maioria das exceções não é tratada e acabam resultando em mensagens de erro. Exemplos:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

```
>>> f = open("arquivoquenaoexiste","r")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'arquivoquenaoexiste'
```

Tratamento de exceções

```
while True:
    try:
        x = int(raw_input("Nobre senhor e senhora, digite um \
    numero: "))
        break
    except ValueError:
        print("Meu senhor... minha senhora... Isso não é um \
    número válido. Vai lá, entre outra vez!")
```

Tratamento de exceções

Caso ocorra uma exceção durante a execução da cláusula try, os comandos que ainda faltam ser executados nela são ignorados. Se o tipo de exceção ocorrida tiver sido previsto com alguma cláusula except, então essa cláusula será executada. Ao fim da cláusula também termina a execução do try como um todo.

Tratamento de exceções

```
try:
    x = int(raw_input('Please enter a number: '))
finally:
    print('Thank you for your input')
```

Tratamento de exceções

Exceções podem ser lançadas com a palavra reservada 'raise'.

```
class MeuErro(Exception):
  def __init__(self, valor):
    self.valor = valor
  def __str__(self):
    return repr(self.valor)

try:
  raise MeuErro(2*2)
except MeuErro as e:
  print 'Minha exceção ocorreu, valor:', e.valor
```

Saída:

```
Minha exceção ocorreu, valor: 4
```

Concorrência

- Python possui suporte padrão a programação concorrente.
- O suporte é implementado no módulo threading. Neste módulo estão implementados também ferramentas para manipular threads com segurança, como semáforos e monitores.

```
from threading import Thread

class simpleThread(Thread):
        def run(self):
                print(str(self.name) + ": Oi")
                print(str(self.name) + ": Tchau")

thread1 = simpleThread(name = "A")
thread2 = simpleThread(name = "B")
thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

Herdar de Thread e definir um método run.

Não necessariamente nesta ordem.

Saída

A: Oi
A: Tchau
B: Oi
B: Tchau

Semáforo

```
from threading import Semaphore

semaforo = Semaphore()

semaforo.acquire() # equivalente a operação P()

# código a ser protegido

semaforo.release() # equivalente a operação V()
```

Monitor

```
from threading import Lock

lock = Lock()


with lock:

    # código a ser protegido
```

Avaliação da linguagem

Critérios gerais	C	Java	Python
Aplicabilidade	sim	parcial	sim
Confiabilidade	não	sim	sim
Aprendizado	não	não	sim
Eficiência	sim	parcial	parcial
evolutibilidade	não	sim	sim
reusabilidade	sim	sim	sim
integração	sim	parcial	sim
custo	depende da aplicação	depende da aplicação	depende da aplicação

Critérios específicos	C	Java	Python
Escopo	sim	sim	sim
Expressões e comandos	sim	sim	sim
Tipos primitivos	sim	sim	sim
Gerenciamento de memória	programador	sistema	sistema/Programador
Persistência de dados	biblioteca de funções	biblioteca de classes,serialização	biblioteca de classes,serialização

Critérios específicos	C	Java	Python
Passagem de parâmetros	lista variável e por valor	lista variável,por valor e por copia de referência	lista variável,por valor e por copia de referência
Encapsulamento e proteção	parcial	sim	sim
Sistemas de tipos	não	sim	sim
Verificação de tipos	estática	estática/dinamica	dinâmica
Polimorfismo	Coerção e sobrecarga	todos	todos
Exceções	não	sim	sim
Concorrência	não	sim	sim

Referências

https://www.python.org/

https://docs.python.org/3/

http://mindbending.org/pt/a-historia-do-python

https://www.tutorialspoint.com/python/

https://jeffknupp.com

https://wiki.python.org/

http://sebastianraschka.com