

Introdução à linguagem Python

Thiago Martins

Python

Programa (aulas 1 a 5):

1. Características gerais da linguagem, tipos fundamentais, tipos agregados, introdução a variáveis, mutabilidade.
2. Programas, controle de fluxo. Introdução a funções. Escopo e visibilidade de variáveis.
3. Exercícios.
4. Objetos e Classes. Métodos e atributos. Construtores. Herança. Encapsulamento (ou falta de?). Erros e Exceções.
5. Exercícios.

Python

Programa (aulas 6 a 13):

6. Aspectos funcionais da linguagem. Funções como objetos. Clausuras. Geradores. Funções de alta ordem. Exercícios.
7. Módulos e Pacotes. Exercícios.
8. Biblioteca Python: IO de arquivos, urllib.
9. Biblioteca Python: JSON, CSV, Expressões Regulares. Exercícios.
10. Computação científica em Python: Numpy. Exercícios.
11. Computação científica em python Matplotlib. Exercícios.
12. Computação científica em python: Pandas.
13. Aspectos variados: mypy(?) sql(?) I/O assíncrono(?), web(?)

Python

Referências

Página da disciplina (em construção...)

How to Think Like a Computer Scientist: Interactive Edition por A. B. Downey.

<https://runestone.academy/runestone/books/published/thinkcspy/index.html>

Documentação da versão 3 de python

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

Google colab

<https://colab.research.google.com/>

Python

Linguagem de alto nível

- Estruturas de controle de fluxo, programação estruturada, memória gerenciada, etc...

Linguagem “genérica”

- Usada em comunicação, criação de conteúdo, jogos, computação científica, etc...

Linguagem Interpretada

- Código-fonte é processado no momento de sua execução

Python

Linguagem de programação *imperativa*

... mas com suporte a construtos funcionais.

Orientada a objetos

... mas sem “radicalismos”

Simples

... mas poderosa!

Histórico

- Criada por Guido van Rossum no *Centrum Wiskunde&Informatica* da Holanda
- Primeira versão pública em 1991
- Versão 1 em 1994
 - Desenvolvimento migra para Corporation for National Research Initiatives, EUA

Python

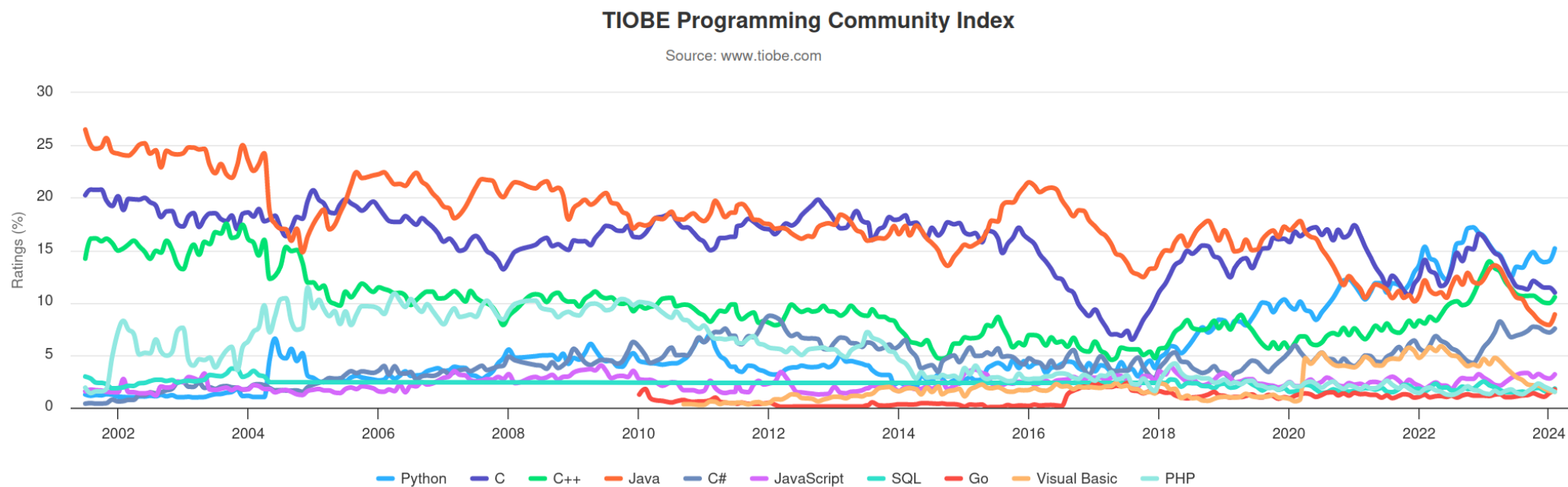
Histórico

- 2000: Licença GPL
- Versão 2 em 2000
- Versão 3 em 2008

Transição 2 para 3 *ainda* dolorosa!

Python

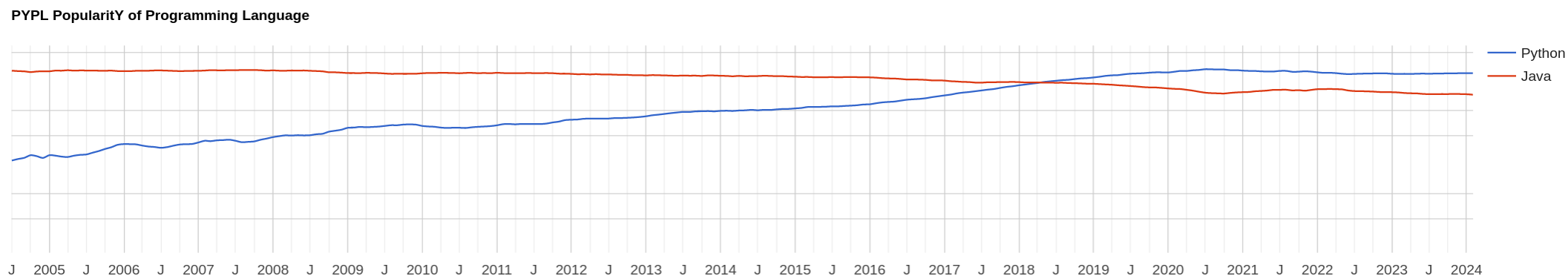
Uso



Python

Uso

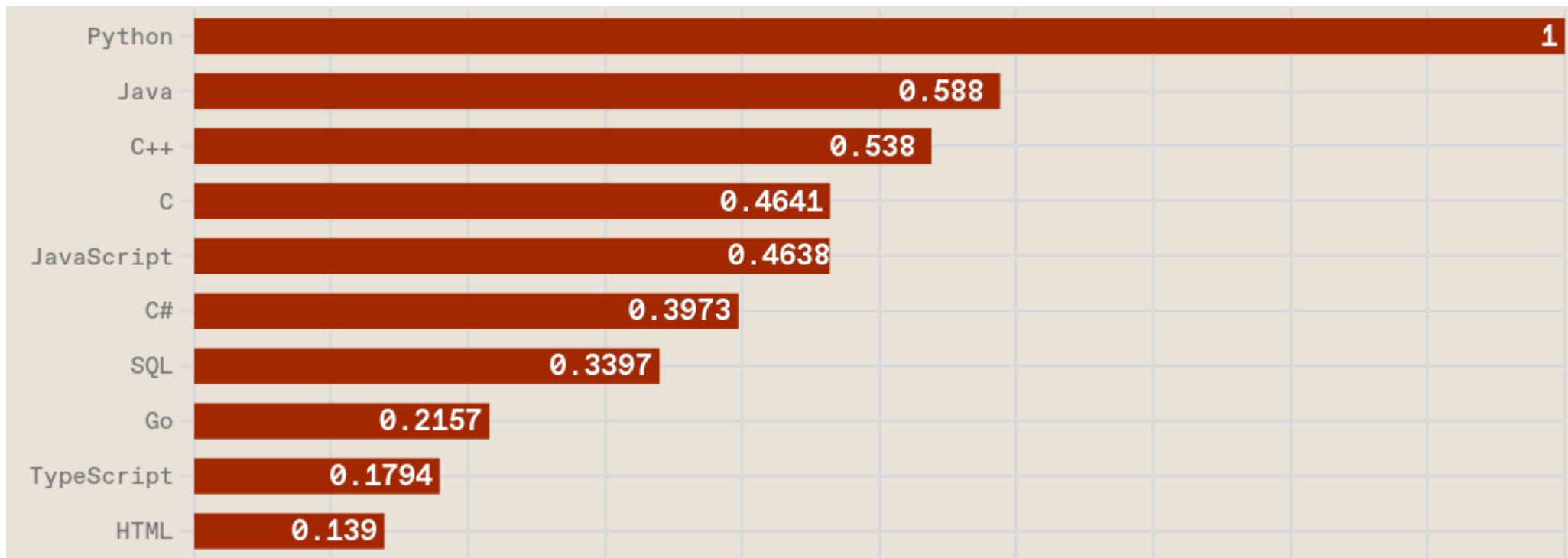
pypl em 2024, Python vs. Java



Python

Uso

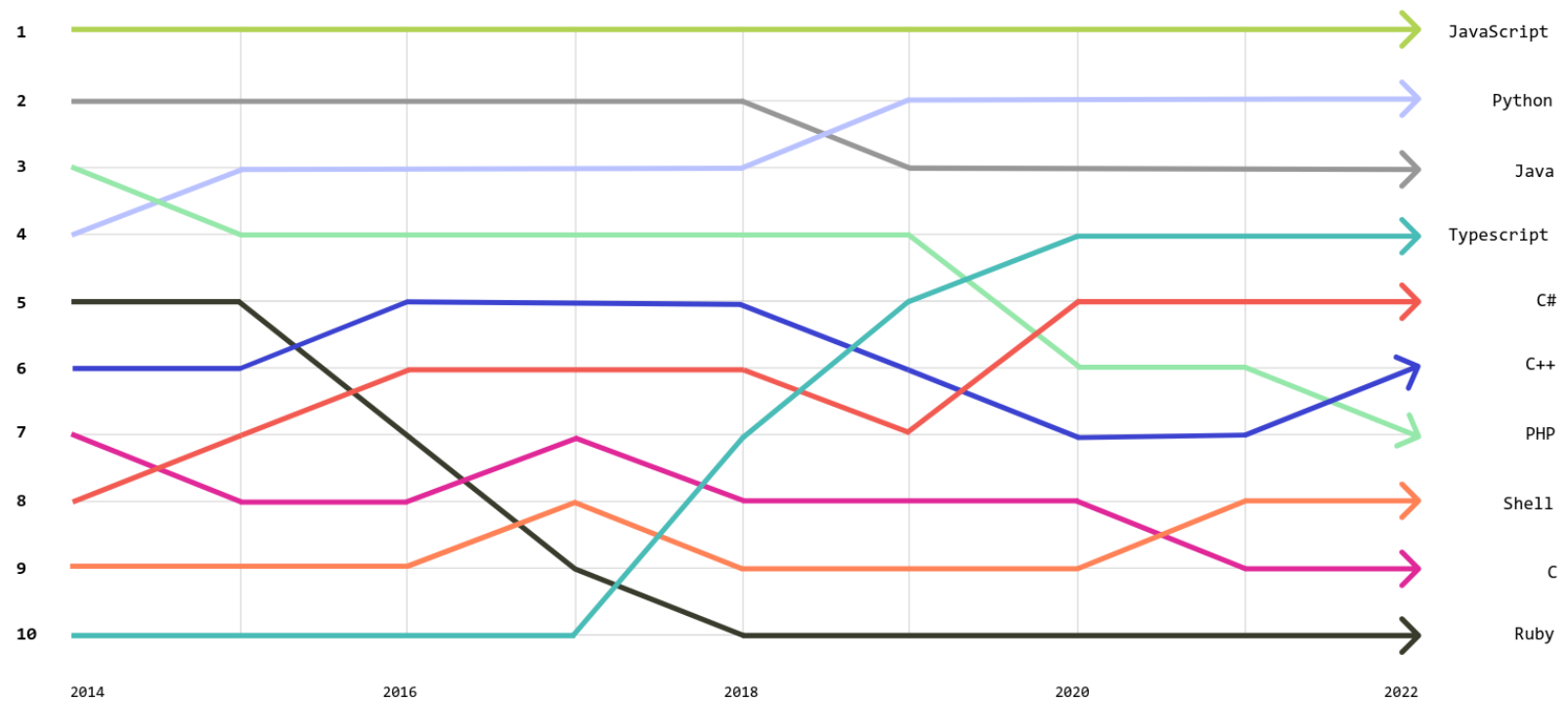
Escala da IEEE Spectrum em 2019



Python

Uso

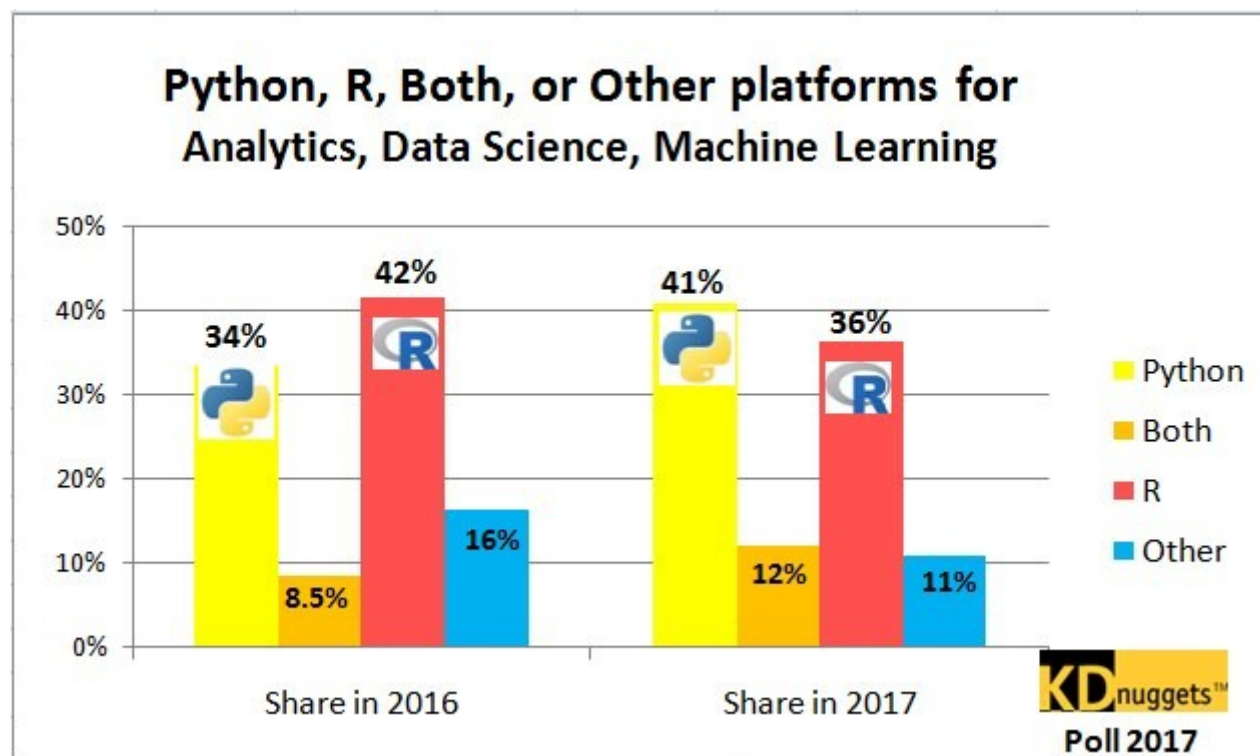
Github (2022)



Python

Uso

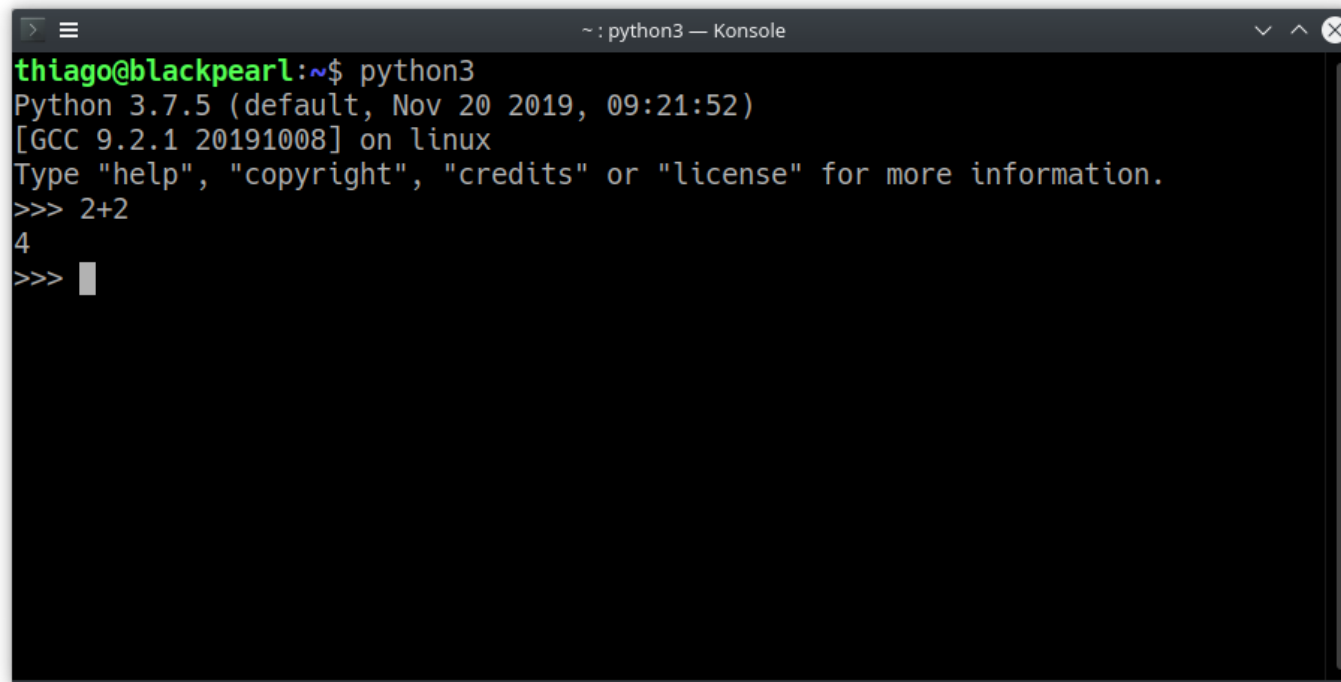
Kdnuggets:
Data science



Python

Ferramentas:

interpretador CPython



```
~ : python3 — Konsole
thiago@blackpearl:~$ python3
Python 3.7.5 (default, Nov 20 2019, 09:21:52)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> 
```

Python

Ferramentas:

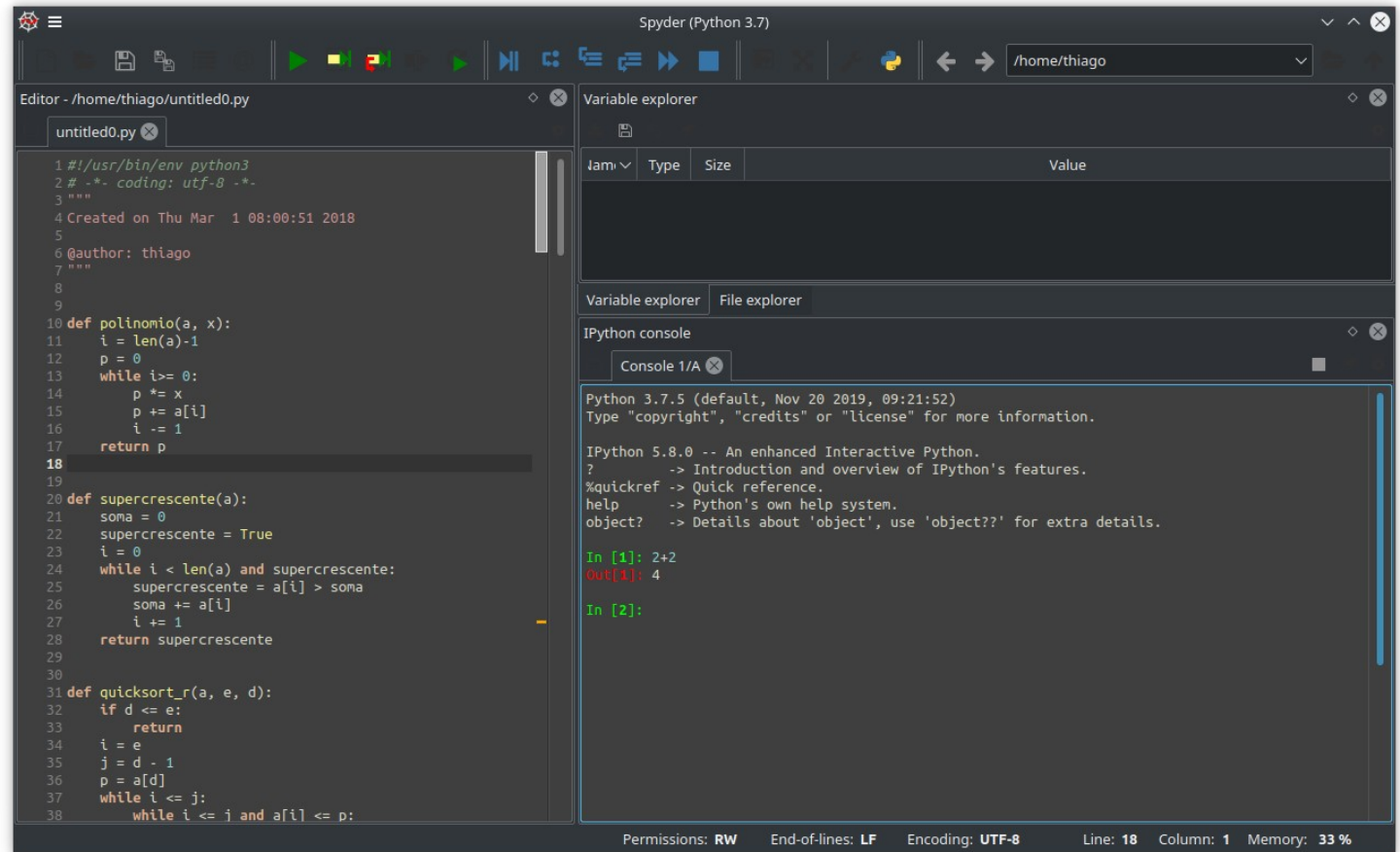
IDE's

Spyder

Netbeans

Eclipse

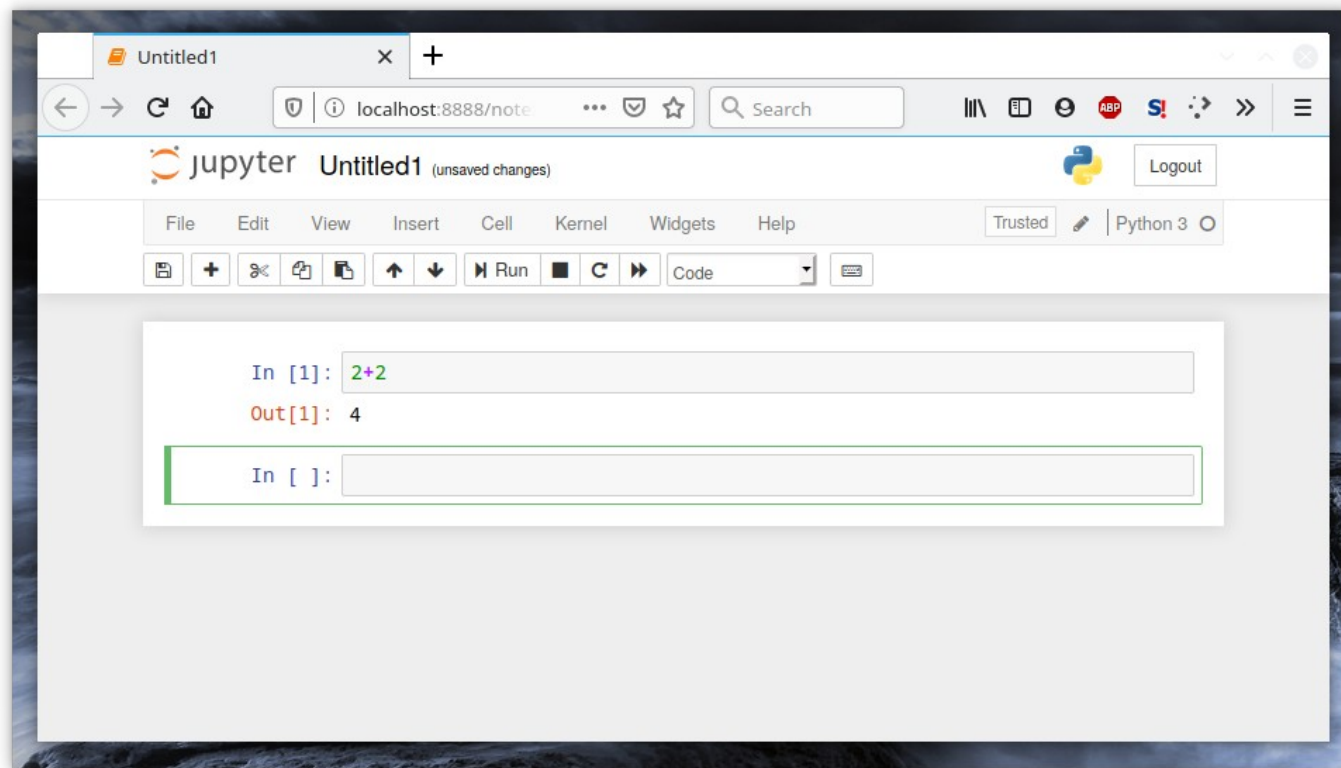
VSCode...



Python

Ferramentas:

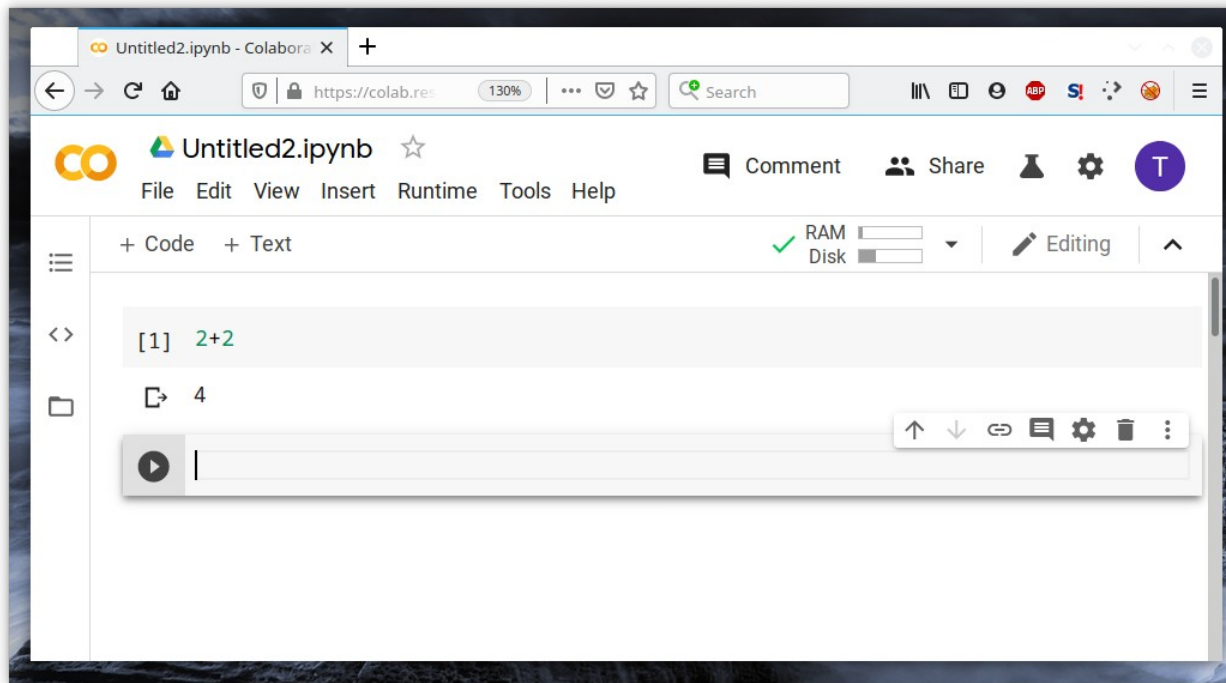
Jupyter notebook



Python

Ferramentas:

Google CoLab



Amazon SageMaker
Microsoft Azure Notebooks
...

Conceitos fundamentais

Instruções:

Diretrizes *executadas* pelo interpretador Python

e. g.: `print(2+2)`

Expressões:

Trechos de linguagem que possuem um *valor*

e. g.: `2+2`

Variáveis:

Referências para objetos na memória.

e. g.: `a = 2`

Tipos de dados

Simples:

Numéricos: int, float, complex

Booleanos: bool

Agregados Sequenciais:

string, lista, tupla

Agregados não-Sequenciais:

Dicionários

Conjuntos

Objetos

Funcionais:

Funções

Geradores

Booleanos

Constantes: True, False

Operadores:

ou	x or y
e	x and y
não	not x

Obedecem *curto-circuito*: (mais adiante...)

Dados numéricos

Construtores: `int()`, `float()`, `complex()`

`int(1) → 1`

`int(0.5) → 0`

`int("1") → 1`

`int("1.4") → Erro`

Dados numéricos

Construtores: `int()`, `float()`, `complex()`

`float(1) → 1`

`float(0.5) → 0.5`

`float("1") → 1`

`float("1.4") → 1.4`

`float("j") → Erro`

`complex("1+j") → 1+i`

Dados numéricos

Operadores

Aritméticos

Divisões inteiras

arredondadas

para baixo

soma $x + y$

subtração $x - y$

multiplicação $x * y$

divisão (real) x / y

divisão (inteira) $x // y$

resto da divisão $x \% y$

Exponenciação $x ** y$

Dados numéricos

Operadores

bit a bit

AND	$x \& y$
OR	$x y$
NOT	$\sim x$
XOR	$x \wedge y$
Shift p/ esquerda	$x \ll y$
Shift p/ direita	$x \gg y$

Dados numéricos

Operadores

Comparação

Estritamente menor que	$x < y$
Menor ou igual a	$x \leq y$
Igualdade	$x == y$
Maior ou igual a	$x \geq y$
Estritamente maior que	$x > y$
Diferente de	$x \neq y$

Strings

Sintaxes para literais:

`'exemplo de cadeia'`

`"outro exemplo"`

`'aspas duplas "dentro" de simples'`

`"ou simples 'dentro' de duplas"`

Caracteres especiais:

`'\'' '\\"' '\n' '\r' '\b'`

Múltiplas linhas:

`"""a b c d
e f g h"""`

Strings

Concatenação

`'exemplo ' + 'de ' + 'cadeia' => 'exemplo de cadeia'`

Constantes adjacentes são concatenadas

`'exemplo ' 'de ' 'cadeia'`

Acesso indexado (índice baseado em zero)

`'abc'[0] → 'a' 'abc'[2] → 'c'`

`'abc'[-1] → 'c' 'abc'[-2] → 'b'`

Slices (fatias)

`'abcdef'[1:2] → 'bc' 'abcdef'[:3] → 'abc' 'abcdef'[3:] → 'def'`

Comprimento

`len('abc') → 3`

Strings

Pertencimento

`'bc' in 'abcde' → True`

`'ef' in 'abcde' → False`

Atenção! Complexidade **quadrática**

Comparação

`'abc' == 'abc' → True`

`'abc' < 'abd' → True`

`'abc' < 'baa' → True`

`'ab' < 'abc' → True`

`'' < 'abc' → True`

Listas

Tipo *sequencial* agregado

[1, 2, 3] [0.5, 0.6, 0.7]

Tipo *heterogêneo*(!)

[1, 1.5, 2] [1, 'dois'] [1, [1, 2]]

Endereçável com índice *baseado em zero*

[1, 1.5, 2][0] → 1 [1, 1.5, 2][1] → 1.5

Esta sintaxe de endereçamento por [] vale para diversos tipos.

Listas

Comprimento

`len([1, 2, 3])` → 3

Pertencimento:

`1 in [1, 2, 3]` → True `4 in [1, 2, 3]` → False

Concatenação

`[1, 2, 3]+[4, 5, 6]` → `[1, 2, 3, 4, 5, 6]`

Tuplas

Tipo *sequencial* agregado

(1, 2, 3) (0.5, 0.6, 0.7)

Tipo *heterogêneo*(!)

(1, 1.5, 2) (1, 'dois')

Endereçável com índice *baseado em zero*

(1, 1.5, 2)[0] → 1 (1, 1.5, 2)[1] → 1.5

Tuplas

Comprimento

`len((1, 2, 3)) → 3`

Pertencimento:

`1 in (1, 2, 3) → True` `4 in (1, 2, 3) → False`

Concatenação

`(1, 2, 3)+(4, 5, 6) → (1, 2, 3, 4, 5, 6)`

Qual a diferença para Listas???

Variáveis e objetos

Toda variável é uma *referência* a um objeto do ambiente

$a = 1$ -Variável a aponta agora para um inteiro na memória

$a + 1 \rightarrow 2$

$a = \text{"dois"}$ -Variável a aponta agora para uma string

$b = a$ -Variável b aponta agora para *o mesmo* objeto apontado por a .

Nota-se que o significado de "=" é distinto do normalmente usado na matemática.

$a = a + 1$????

Típico de linguagens imperativas

Variáveis e objetos

Toda variável é uma *referência* a um objeto do ambiente

$a = 1$ -Variável a aponta agora para um inteiro na memória

$a + 1 \rightarrow 2$

$a = \text{"dois"}$ -Variável a aponta agora para uma string

$b = a$ -Variável b aponta agora para *o mesmo* objeto apontado por a .

Nota-se que o significado de "=" é distinto do normalmente usado na matemática.

$a = a + 1$????

Típico de linguagens imperativas

Variáveis e objetos

Atribuição *composta*

<code>a += 1</code>	o mesmo que*	<code>a = a + 1</code>
<code>a -= 1</code>	o mesmo que*	<code>a = a - 1</code>
<code>a *= 2</code>	o mesmo que*	<code>a = a + 2</code>
<code>/=</code>	<code>%=</code>	<code>//=</code>
	<code>**=</code>	...

Atribuição simultanea

`a, b = 1, 2`

-a aponta para 1, b para 2

`a, b = b, a`

-Troca as referências

*quase sempre (vide objetos mutáveis adiante)

Variáveis e objetos

Variáveis em Python têm um tipo: a *referência*

- Mas esta pode apontar para *qualquer* tipo de objeto...

Cada elemento de uma lista ou tupla é na verdade uma referência

Operador type: recupera o tipo de um objeto

```
type("abc") → Class str
```

```
a = 1
```

```
type(a) → Class int
```

```
a = [1, 2]
```

```
type(a) → Class list
```

Variáveis e objetos

Objetos e identidade

```
a = 1000
```

```
b = a
```

Operador `is`: testa se duas referências apontam para o mesmo objeto

```
a is b → True
```

Isso é *diferente* da comparação

```
a = 10000
```

```
b = a - 1
```

```
b += 1
```

Resultados de

```
a == b
```

```
a is b
```

Objetos e mutabilidade

Se cada objeto tem uma identidade única...

```
a = 10000
```

```
b = a
```

```
a = a + 1
```

```
b is a → False
```

É correto dizer que o inteiro em a foi incrementado?

Objetos e mutabilidade

Inteiros em python são *imutáveis!!!*

```
a = a + 1
```

a expressão (a + 1) do lado direito da atribuição *cria* um novo objeto!

A maioria dos tipos básicos em Python são imutáveis

```
a = "123"
```

```
a[2] = "4" ERRO!
```

```
a = (1, 2, 3)
```

```
a[2] = 4 ERRO!
```

Strings, floats, complexos, tuplas são imutáveis.

Objetos e mutabilidade

Mas *listas* não o são! É ESTA a diferença para tuplas!

```
a = [1, 2, 3]
```

```
a[2] = 4
```

```
a[2] → 4
```

Elementos podem ser acrescentados ou removidos de listas!

```
a = [1, 2, 3]
```

```
del a[1]
```

```
a → [1, 3]
```


Listas como pilhas

Adicionando elementos ao final da lista:

```
a = []  
a.append(1)  
a.append(2)  
a.append(3)
```

E removendo...

```
a.pop() → 3  
a.pop() → 2  
a.pop() → 1
```

Objetos e mutabilidade

Do mesmo modo, cada elemento de uma tupla é uma referência.

Embora as referências sejam imutáveis, os objetos para os quais elas apontam podem não sê-lo!

```
a = [1, 2, 3]
```

```
b = (a, 4)
```

```
b → ([1, 2, 3], 4)
```

```
del a[0]
```

```
b → ([2, 3], 4)
```

Objetos e mutabilidade

ATENÇÃO!!!!!!

Variáveis são sempre *referências* para objetos!

Em se tratando de objetos mutáveis, isso pode levar a comportamentos não-intuitivos! (aliasing)

```
a = [1, 2, 3]
```

```
b = a
```

```
b.append(4)
```

```
a ➔ [1, 2, 3, 4]          !!!!!!!
```

Pois a e b apontam para o *mesmo* objeto...

Objetos e mutabilidade

Vários operadores produzem um *novo* objeto

```
a = [1, 2, 3]
```

```
b = a[:]
```

```
b is a → False
```

```
a = [1, 2, 3]
```

```
b = a
```

```
a = a + [4, 5, 6]
```

```
b is a → False
```

Objetos e mutabilidade

Vários operadores produzem um *novo* objeto

... mas atribuição composta não!

```
a = [1, 2, 3]
b = a
a += [4, 5, 6]
b is a → True
```



```
a = [1, 2, 3]
b = a
a = a + [4, 5, 6]
b is a → False
```

Sets (Conjuntos)

Tipo agregado *não endereçável*

`{1, 2, 3}` `{0.5, 0.6, 0.7}`

Tipo *heterogêneo* (mas...)

`{1, 1.5, 2}` `{1, 'dois'}` `{1, (1, 2)}`

Só admite tipos com *hash imutável*

`{1, [1, 2]}` **ERRO!**

`{1, {1, 2}}` **ERRO!**

Sets são *mutáveis* (a versão imutável é frozenset)

Sets (Conjuntos)

Tamanho

`len({1, 2, 3}) → 3`

Pertencimento

`1 in {1, 2, 3}) → True`

`4 in {1, 2, 3}) → False`

Inserir novos elementos

`a={1, 2, 3}`

`a.add(4)`

`a → {1, 2, 3, 4}`

Sets (Conjuntos)

Remove elementos (se presentes)

```
a={1, 2, 3}
```

```
a.discard(3)
```

```
a → {1, 2}
```

```
a.discard(4)
```

```
A → {1, 2}
```


Sets (Conjuntos)

Copia conjuntos

```
a = a.copy()
```

```
a.discard(3)
```

```
a → {1, 2}
```

```
a.discard(4)
```

```
A → {1, 2}
```

Dados numéricos

Operadores

contém $x \text{ in } y$

igualdade $x == y$

subconjunto $x \leq y$

subconjunto estrito $x < y$

Intersecção $x \& y$

União $x | y$

diferença $x - y$

diferença simétrica $x ** y$

Sets (Conjuntos)

Copia conjuntos

```
b = a.copy()
```

```
b is a → False
```

Incorpora um conjunto

```
a = {1, 2, 3}
```

```
a.update({0, 3, 4})
```

```
A → {0, 1, 2, 3}
```

Remove elementos pertencentes a um conjunto

```
a = {1, 2, 3}
```

```
a.difference_update({0, 3, 4})
```

```
A → {1, 2}
```

Dicionários

Tipo agregado *endereçável* por valores arbitrários (mas...)

`{1:2, 2:3, 3:4}[1] → 2`

`{"um":1, "dois":2, "três":3}["dois"] → 2`

Sintaxe { chave : variável,... }

Só admite chaves com *hash imutável*

`{[1, 2]:3}` **ERRO!**

Mas as variáveis podem ser alteradas!

`a={1:2, 2:3, 3:4}`

`a[1]=0`

`a → {1:0, 2:3, 3:4}`

Dicionários

Tipo agregado *endereçável* por valores *arbitrários* (mas...)

`{1:2, 2:3, 3:4}[1] → 2`

`{"um":1, "dois":2, "três":3}["dois"] → 2`

Sintaxe { chave : variável,... }

Só admite chaves com *hash imutável*

`{[1, 2]:3}` **ERRO!**

Mas as variáveis podem ser alteradas!

`a={1:2, 2:3, 3:4}`

`a[1]=0`

`a → {1:0, 2:3, 3:4}`

Dicionários

Se uma entrada não existe, a atribuição *cria* uma nova

```
a={1:2, 2:3, 3:4}
```

```
a[4] = 5
```

```
a → {1:2, 2:3, 3:4, 4:5}
```

Remoção de uma entrada:

```
del a[1]
```

```
a → {2:3, 3:4, 4:5}
```

Tamanho:

```
len(a) → 3
```

Verifica a existência de uma chave:

```
1 in a → False
```

```
4 in a → True
```

Programação em Python

Um programa é uma *sequência de instruções*

```
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

Programação em Python

Execução:

Inserir o conteúdo em um arquivo .py:

```
thiago@betelgeuse:~$ cat > programa.py
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
thiago@betelgeuse:~$
```


Programação em Python

Execução:

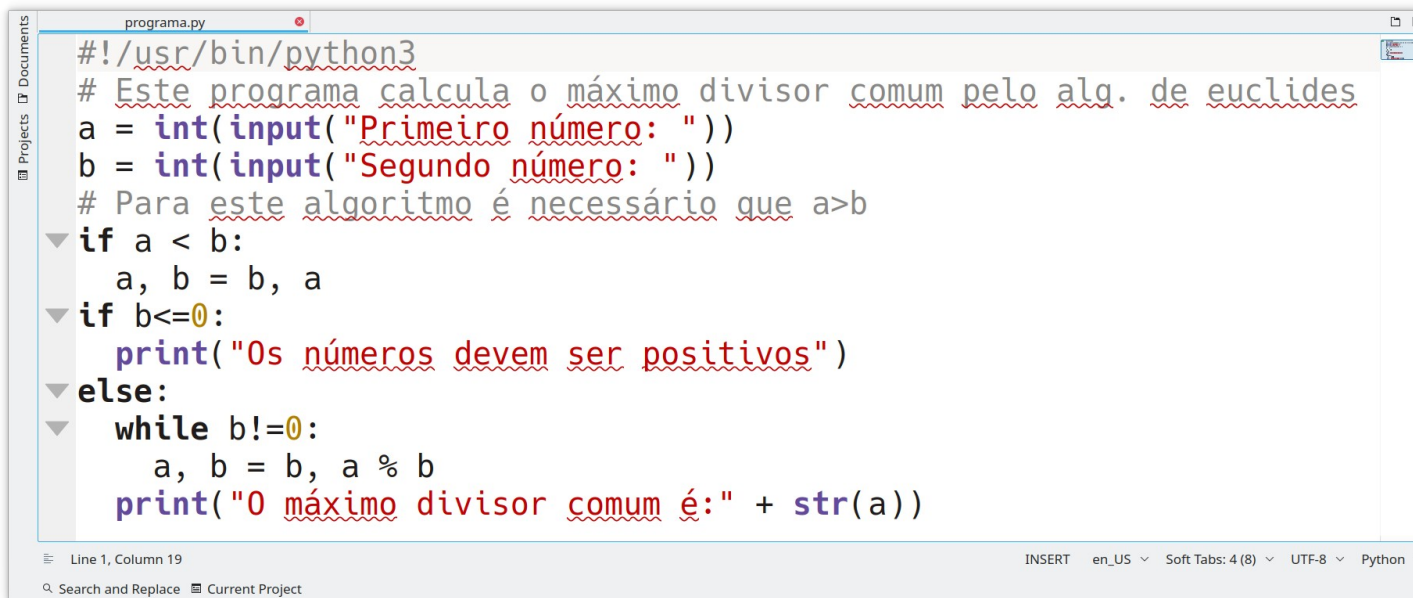
Invocar o interpretador:

```
thiago@betelgeuse:~$ python3 programa.py
Primeiro número: 125
Segundo número: 75
O máximo divisor comum é:25
thiago@betelgeuse:~$
```

Programação em Python

Execução:

Em ambientes -ux o próprio script pode ser executado diretamente!

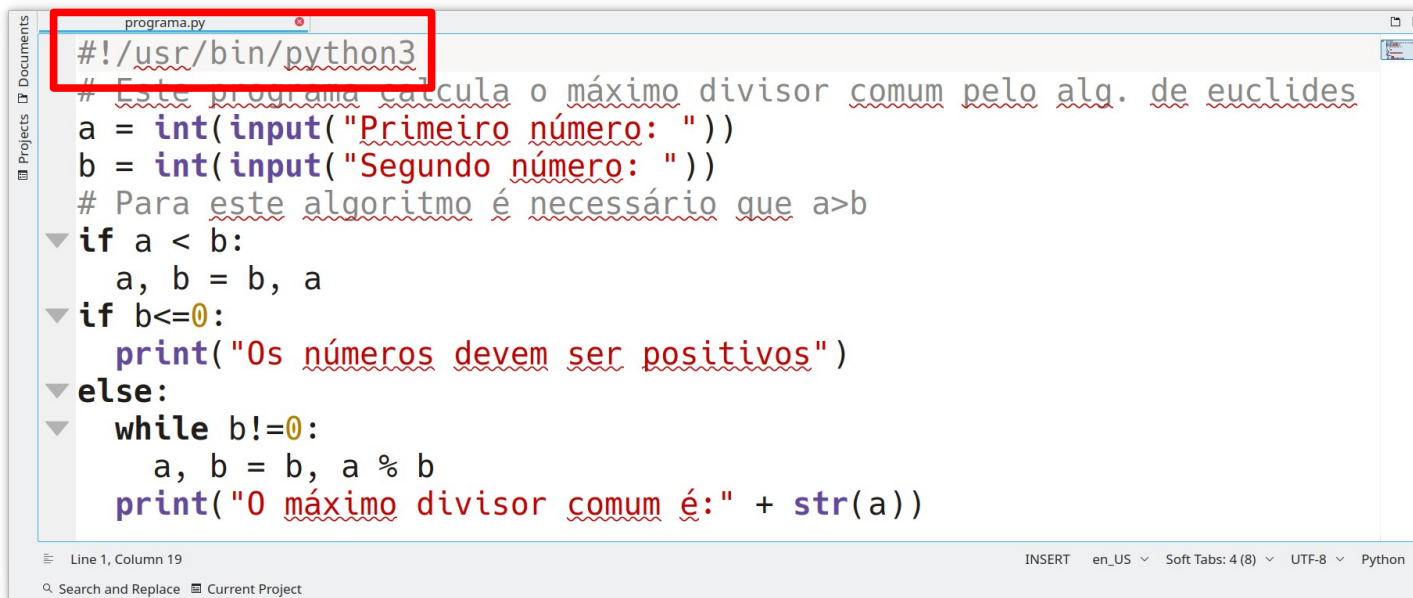
A screenshot of a code editor window titled 'programa.py'. The code is written in Python and implements a function to find the Greatest Common Divisor (GCD) using the Euclidean algorithm. The code includes comments in Portuguese and uses color-coded syntax. The editor interface shows a sidebar on the left with 'Projects' and 'Documents' tabs. The bottom status bar indicates 'Line 1, Column 19', 'INSERT' mode, 'en_US' locale, 'Soft Tabs: 4 (8)', 'UTF-8' encoding, and 'Python' interpreter.

```
#!/usr/bin/python3
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

Programação em Python

Execução:

Em ambientes -ux o próprio script pode ser executado diretamente!



```
#!/usr/bin/python3
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

Programação em Python

Execução:

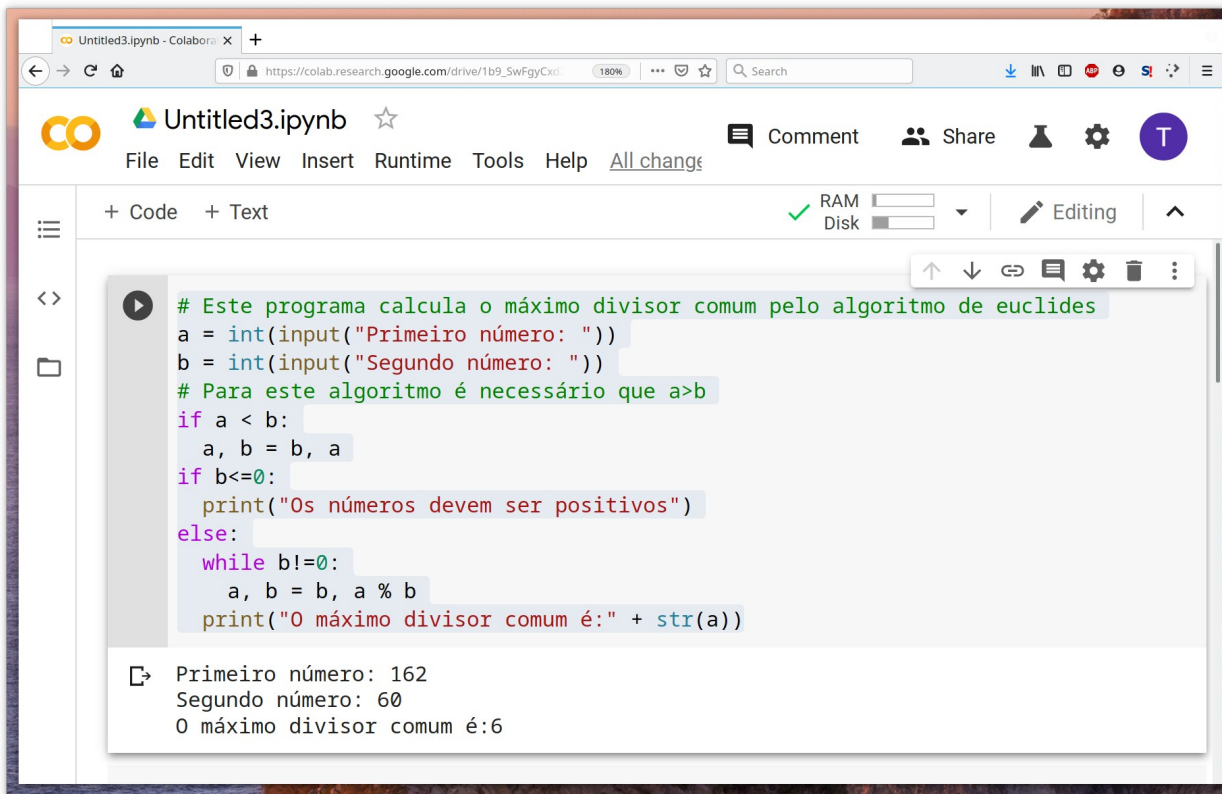
Em ambientes -ux o próprio script pode ser executado diretamente!

```
thiago@betelgeuse:~$ ./programa.py
Primeiro número: 563
Segundo número: 221
O máximo divisor comum é:1
thiago@betelgeuse:~$
```

Programação em Python

Execução:

É possível inserir programas em células do jupyter:



```
# Este programa calcula o máximo divisor comum pelo algoritmo de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

Primeiro número: 162
Segundo número: 60
O máximo divisor comum é:6

Programação em Python

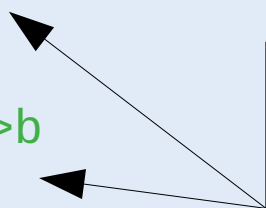
Elementos do programa

```
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

Programação em Python

Elementos do programa

```
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

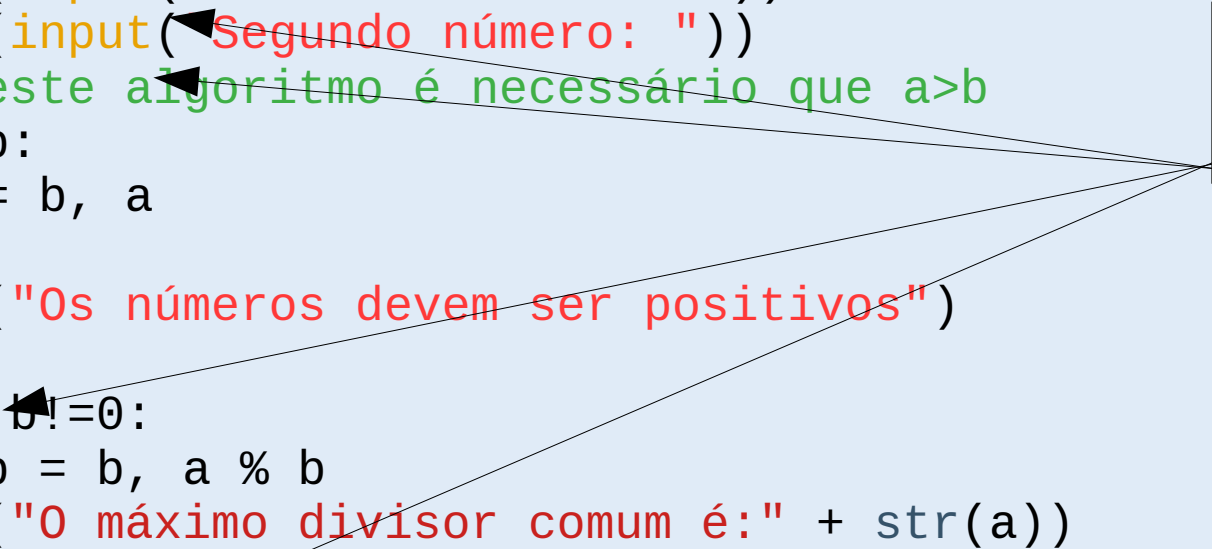


Comentários
(ignorados pelo
interpretador)

Programação em Python

Elementos do programa

```
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```



Chamadas a funções (subprogramas)

Programação em Python

Elementos do programa

```
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

Construtores de objetos

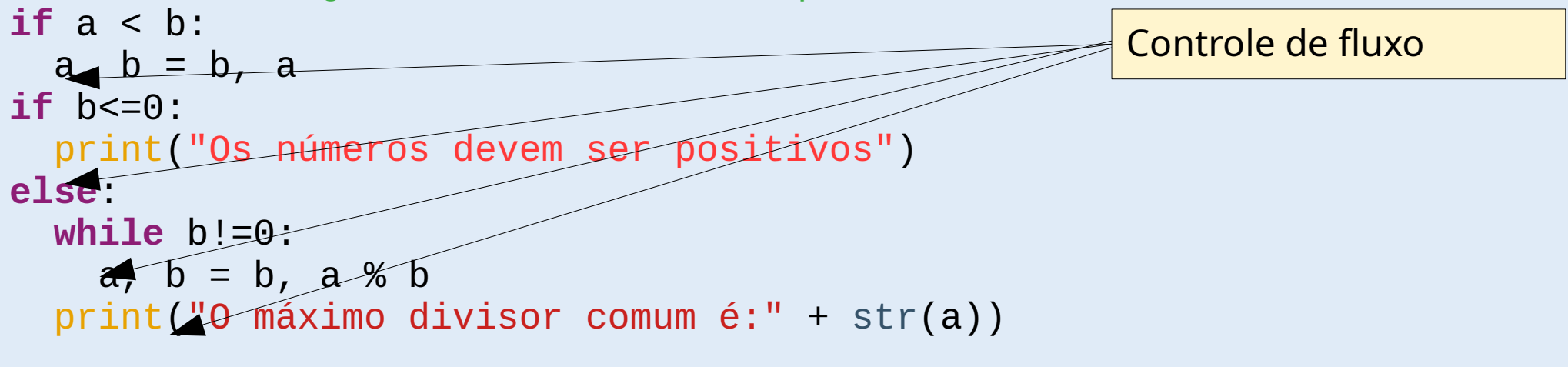


Programação em Python

Elementos do programa

```
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

Controle de fluxo



Programação em Python

Elementos do programa

```
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

Blocos delimitados por
indentação

Programação em Python

Elementos do programa

```
# Este programa calcula o máximo divisor comum pelo alg. de euclides
a = int(input("Primeiro número: "))
b = int(input("Segundo número: "))
# Para este algoritmo é necessário que a>b
if a < b:
    a, b = b, a
if b<=0:
    print("Os números devem ser positivos")
else:
    while b!=0:
        a, b = b, a % b
    print("O máximo divisor comum é:" + str(a))
```

Blocos delimitados por
indentação

Comentários

São marcados pelo caractere # e vão até o final da linha

```
# Inicializa o valor de a
```

```
a = 1
```

Podem se seguir a instruções

```
a = 1 # Inicializa o valor de a
```

Atenção! Docstrings não são comentários

```
""" Isso é uma docstring """
```

```
a = 1
```

Controle de fluxo

if... elif... else

```
if <condição 1>:  
    <bloco 1>  
elif <condição 2>:  
    <bloco 2>  
elif <condição 3>:  
    <bloco 3>  
else  
    <bloco alternativo>
```

Controle de fluxo

while

```
while <condição>  
    <bloco>
```

Executa o bloco enquanto a condição for verdadeira.

Controle de fluxo

for

```
for <variável> in <coleção enumerável>  
    <bloco>
```

Executa o bloco atribuindo à variável valores sucessivos da coleção

ex:

```
a = 0
```

```
for i in [1, 2, 3]
```

```
    a += i
```

```
print(a)
```


Controle de fluxo

for e range

A função range é particularmente apropriada para o laço for:

```
a = 0
for i in range(3)
    a += i
print(a)
```

Controle de fluxo

for e range

`range(n)` Enumera os inteiros de 0 a $n-1$

`range(n, m)` Enumera os inteiros de n a $m-1$

`range(n, m, s)`

Enumera os inteiros de n , $n+2s$, $n+3s$... até o maior valor de $n+ks < m$

Controle de fluxo

for e range

`range(n)` Enumera os inteiros de 0 a $n-1$

`range(n, m)` Enumera os inteiros de n a $m-1$

`range(n, m, s)`

Enumera os inteiros de n , $n+2s$, $n+3s$... até o maior valor de $n+ks < m$

Controle de fluxo

break, continue, pass

break força a saída *imediata* do bloco.

```
# Procura uma posição em a igual a b
for i in range(len(a)):
    if b == a[i]:
        break
```

com o break a execução atual do bloco

```
# Gera todos os primos menores que 100
primos = []
for i in range(2, 100):
    primo = True
    for div in primos:
        if (i % div) == 0:
            primo = False
            break
    if primo:
        primos.append(i)
```

Chamadas a funções/métodos

“Funções” em python são distintas do conceito matemático (verdade para toda linguagem imperativa estruturada)

```
a = iter([1, 2, 3, 4, 5])
```

```
next(a) → 1
```

```
next(a) → 2
```

```
next(a) → 3
```

O antigo termo “subprograma” evita esta confusão, mas não é mais usual.

Python

Orientad

... mas s

```
class program {  
    Public static main(String args[]) {  
        System.out.println("Hello World!");  
    }  
}
```

"Hello World" em Java

```
print("Hello World!\n")
```

"Hello World" em Python

Python

```
#include <string.h>
#include <memory.h>

#define ERR_OK 0
#define ERR_ARG 1

int remove_elemento(int **vetor, unsigned *tamanho, unsigned posicao) {
    if(posicao >= tamanho) return ERR_ARG;
    if(posicao < tamanho - 1)
        memmove(*vetor+posicao, *vetor+posicao+1, tamanho-posicao-1);
    *tamanho--;
    *vetor = realloc(*vetor, *tamanho);
    return ERR_OK;
}
```

Eliminar uma posição de vetor em C

```
def remove_elemento(vetor, posicao):
    del vetor[posicao]
```

Eliminar uma posição de vetor em Python

Python

Ling

```
a = 1
for i in range(2,10):
    a = a * i
...
print(a)
```

Calcula 10! (estilo imperativo)

```
from functools import reduce
from operator import mul
print(reduce(mul, range(1,10)))
```

Calcula 10! (estilo funcional)