Código Bom

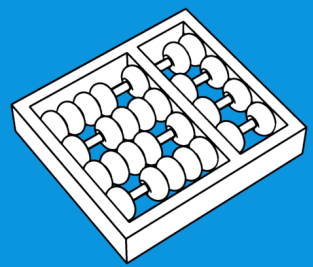
MC102 - Algoritmos e Programação de Computadores

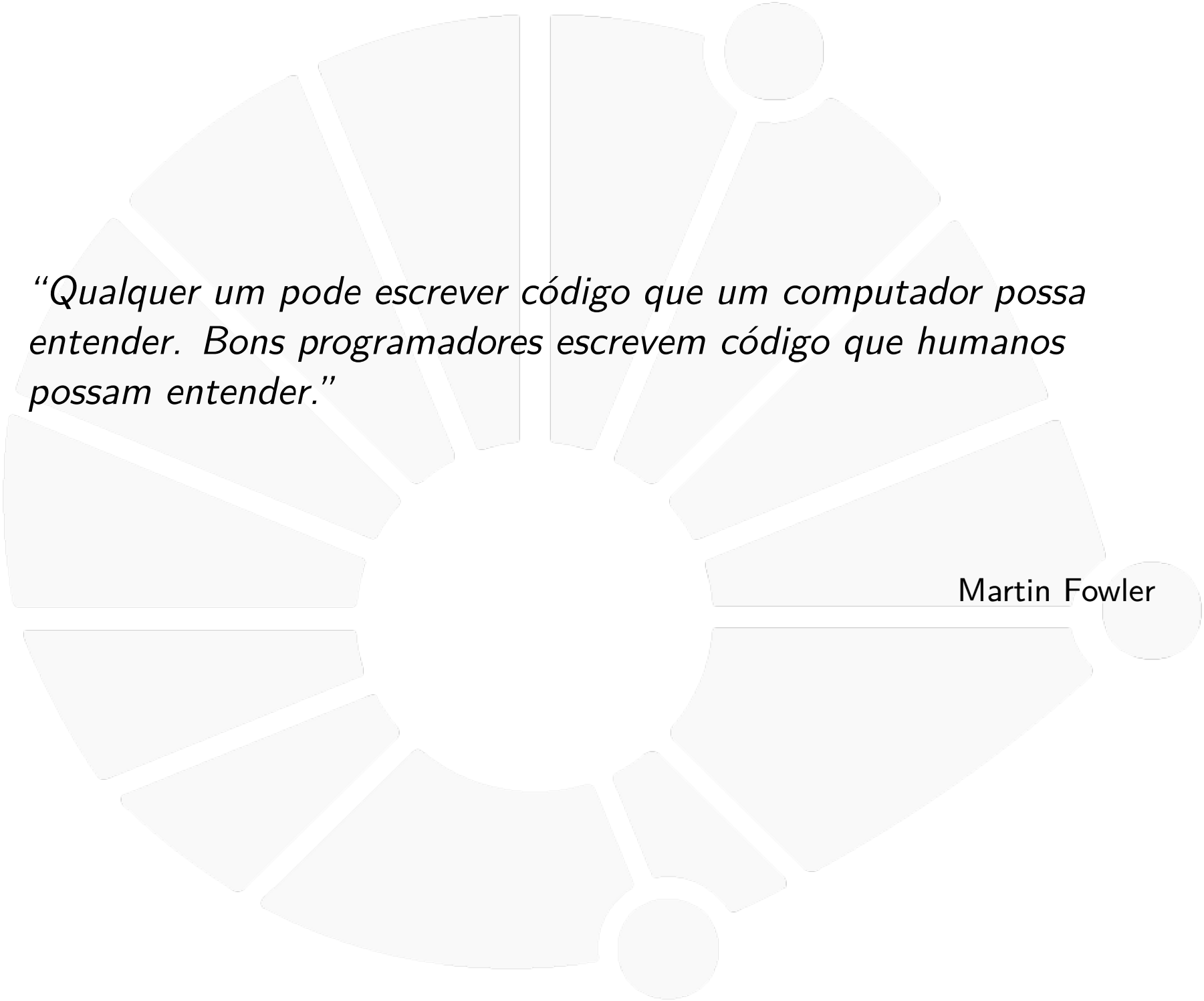
04/25

12

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP

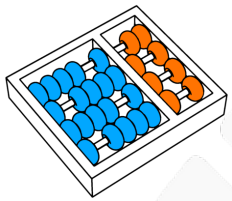




"Qualquer um pode escrever código que um computador possa entender. Bons programadores escrevem código que humanos possam entender."

Martin Fowler

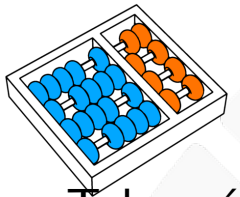
Dúvidas da aula anterior



Dúvidas selecionadas

- Não entendi exatamente a utilidade de uma função lambda.
- Tem alguma diferença entre usar o finally e escrever o código que seria identado no finally fora do try? (Como a parte que está no finally iria acontecer de qualquer forma, não seria possível só colocar essa parte depois do try?)
- Não entendi bem quando você falou das variáveis inteiras apontarem para o número.
- Não entendi como funciona o "raise", nem o "as". Poderia explicar novamente?
- Vimos alguns erros comuns quando vimos o tratamento de exceções com o senhor, mas ao passo que vamos criando nossas próprias rotinas dentro do código, erros novos vão aparecer e gostaríamos de criar esses erros... Isso seria possível? Como fazer?
- Da para chamar funções no Lambda?
- Dentro do except, é possível lançar um outro erro?
- Porque quando uso lista=list(map(int, input().split()) os elementos da lista são reconhecidos como tipo int, mas quando faço lista=[map(int, input().split())] eles são reconhecidos como tipo map?
- Não entendi como posso criar a minha própria biblioteca e deixar ela disponível em outros códigos.
- Ao invés de escrever "except ValueError as e"posso só escrever "except as e"pra pegar qualquer erro?
- Para que serve o else na função except?
- Todo try precisa de um except ou posso usar somente o try?
- Entendi que quando executo um programa em python existe uma variavel padrão chamada __name__ que pode ou não ter o valor "__main__, mas não entendi porque preciso usar isso quando estou importando funções de um programa para outro.

Código bom?!



O que é um código bom?

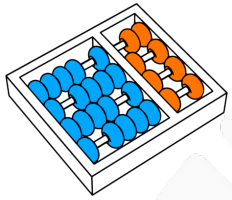
Talvez é mais fácil olhar primeiro o que não é. . .

```
1  def imprime(n):
2      for i in range(3,n,2):
3          eh=True
4          k=3
5          while k<i and eh:
6
7
8              eh=i%k!=0
9
10
11
12             k+=1
13         if eh: print(i)
```

O que essa função faz?

- Temos que pensar um bom tempo para perceber.

Estilo



Estilo

Quando programamos é importante ter um estilo de escrita claro e consistente:

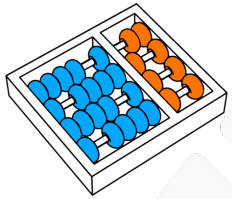
- Também que esteja de acordo com o que o resto da equipe usa.

É comum usarmos um **linter** para verificar regras de estilo:

- O linter lê o código e indica problemas de legibilidade.

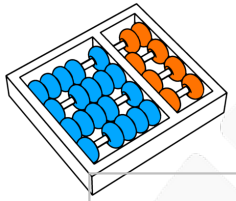
Em Python, uma opção é usar o **Flake8**.

- Vamos ver o que ele diz do nosso código original.



Flake8 no nosso código

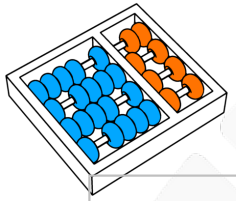
```
ruim1.py:2:21: E231 missing whitespace after ','
ruim1.py:2:23: E231 missing whitespace after ','
ruim1.py:3:10: E225 missing whitespace around operator
ruim1.py:4:10: E225 missing whitespace around operator
ruim1.py:5:16: E225 missing whitespace around operator
ruim1.py:6:1: W293 blank line contains whitespace
ruim1.py:7:1: W293 blank line contains whitespace
ruim1.py:8:13: E303 too many blank lines (2)
ruim1.py:8:14: E225 missing whitespace around operator
ruim1.py:8:16: E228 missing whitespace around modulo
    operator
ruim1.py:8:18: E225 missing whitespace around operator
ruim1.py:9:1: W293 blank line contains whitespace
ruim1.py:10:1: W293 blank line contains whitespace
ruim1.py:11:1: W293 blank line contains whitespace
ruim1.py:12:13: E303 too many blank lines (3)
ruim1.py:12:14: E225 missing whitespace around operator
ruim1.py:12:17: W291 trailing whitespace
ruim1.py:13:13: E701 multiple statements on one line (colon)
ruim1.py:13:23: W291 trailing whitespace
ruim1.py:16:12: W292 no newline at end of file
```



Nova versão

```
1  def imprime(n):
2      for i in range(3, n, 2):
3          eh = True
4          k = 3
5          while k < i and eh:
6              eh = (i % k != 0)
7              k += 1
8          if eh:
9              print(i)
```

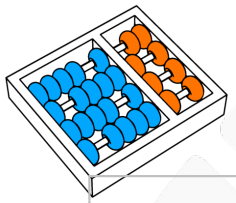
- Não há quebras de linhas excessivas.
- Os operadores binários estão cercados por espaços.
- Temos espaço após as vírgulas.
- Não temos o corpo do **if** na mesma linha.
- Usamos um parêntese na linha 6 para deixar mais claro.
- Flake8 não aponta mais erros.
- Mas dá para ser melhor...



Outra versão

```
1  def imprime_primos(n):
2      for numero in range(3, n, 2):
3          eh_primo = True
4          divisor = 3
5          while divisor < numero and eh_primo:
6              eh_primo = (numero % divisor != 0)
7              divisor += 1
8          if eh_primo:
9              print(numero)
```

- Nome da função dá uma pista do que ela faz.
- Nomes das variáveis dão pistas do que são.
- Linha 6 é mais fácil de ler.
  - Pode ficar ainda mais fácil com um **if**.
- Bons nomes já ajudam a documentar o código.
- Mas poderia ser melhor ainda...

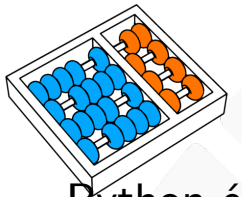


E mais outra versão

```
1  def eh_primo(numero):
2      ''' Devolve se o numero dado é primo ou não.'''
3      divisor = 3
4      while divisor < numero:
5          if numero % divisor == 0:
6              return False
7          divisor += 1
8      return True
9
10 def imprime_primos(n):
11     ''' Imprime os primos menores ou iguais a n.'''
12     for numero in range(3, n, 2):
13         if eh_primo(numero):
14             print(numero)
```

- Temos uma função para saber se é primo.
- **imprime_primos** fica mais fácil de entender.
- A docstring também ajuda:
  - Não são os **n** primeiros primos.
  - Mas sim os menores ou iguais a **n**.

Tipagem dinâmica



Tipagem Dinâmica

Python é uma linguagem de tipagem dinâmica:

- Uma variável pode guardar qualquer tipo.

Ex:

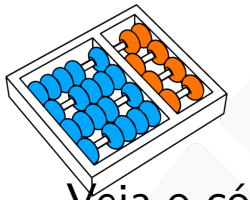
```
1  x = 10
2  x = str(x) + '!'
3  print(x)          # imprime 10!
```

Isso permite algumas coisas interessantes:

```
1  def soma(x, y):
2      return x + y
3
4  print(soma(2, 3))       # imprime 5
5  print(soma('a', 'b'))   # imprime ab
```

Porém, algumas checagens de bugs são perdidas com isso.

- Ademais, o código pode ser mais difícil de entender.



Dica de Tipo — Type Hinting

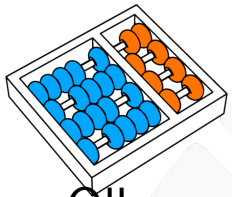
Veja o código abaixo:

```
1  def eh_primo(numero: int) -> bool:
2      ''' Devolve se o numero dado é primo ou não.'''
3      divisor: int = 3
4      while divisor < numero:
5          if numero % divisor == 0:
6              return False
7          divisor += 1
8      return True
9
10 def imprime_primos(n: int) -> None:
11     ''' Imprime os primos menores ou iguais a n.'''
12     for numero in range(3, n, 2):
13         if eh_primo(numero):
14             print(numero)
```

Damos algumas **dicas** dos tipos esperados das variáveis:

- O `:` nos dá a dica do tipo esperado.
- O `->` diz o tipo de retorno da função.

E o que fazer com isso?



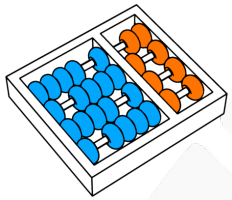
Erro de Execução por causa de tipo

Olhemos esse código:

```
1  from datetime import datetime
2
3  ...
4
5  if datetime.now().hour == 3:
6      imprime_primos(9.5)
7  else:
8      imprime_primos(9)
```

Se não for 3 da manhã, ele roda sem erros... mas, às 3 da manhã ele dá o seguinte erro:

```
Traceback (most recent call last):
  File "ruim2.py", line 22, in <module>
    imprime_primos(9.5)
  File "ruim2.py", line 16, in imprime_primos
    for numero in range(3, n, 2):
TypeError: 'float' object cannot be interpreted as an
    integer
```



Erros de Execução

Dependendo do caminho percorrido no código podemos ter um erro de execução:

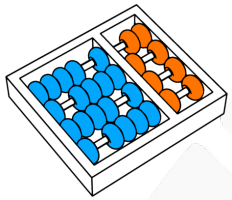
- O difícil é que alguns caminhos podem ser raros.

Linguagens estaticamente tipadas conseguem evitar alguns desses erros:

- Nós sabemos exatamente o tipo da variável.
- Então sabemos que ela não pode assumir certos valores.

Na linguagem C, detectaríamos esse erro muito antes:

- Um **int** não pode assumir o valor **9.5**.
- Teríamos um erro de compilação.
- Independente do caminho a ser percorrido na execução.



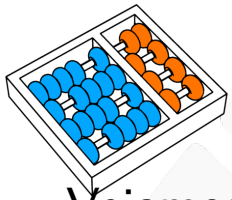
MyPy

O MyPy é um programa que faz checagem de tipo para Python:

- Baseada nas dicas de tipo dadas.

```
ruim2.py:22: error: Argument 1 to "imprime_primos" has
    incompatible type "float"; expected "int"
Found 1 error in 1 file (checked 1 source file)
```

Esse erro encontrado não depende da execução do código.



MyPy

Vejamos o seguinte código:

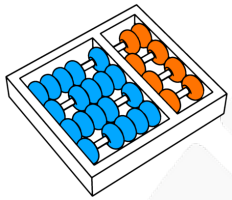
```
1 def soma_inteiros(x: int, y: int) -> int:
2     return x + y
3
4 print(soma_inteiros(2, 3))
5 print(soma_inteiros('a', 'b'))
```

O MyPy encontra o seguinte erro:

```
mypy1.py:5: error: Argument 1 to "soma_inteiros" has
    incompatible type "str"; expected "int"
mypy1.py:5: error: Argument 2 to "soma_inteiros" has
    incompatible type "str"; expected "int"
Found 2 errors in 1 file (checked 1 source file)
```

Mas o código roda no Python sem erros!

- A dica de tipo é só uma dica...
- O Python não deixa de ser dinamicamente tipado.



Sobre tipagem

A biblioteca **typing** dá suporte as dicas:

- Tem várias coisas para aprender lá.
- E depende da versão do Python.

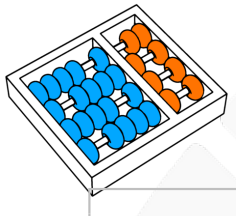
Você precisa instalar o MyPy:

- **pip install mypy**

Você pode executar o MyPy fazendo:

- **mypy meuarquivo.py**
- **mypy minhapasta**
- **mypy .**

Correção

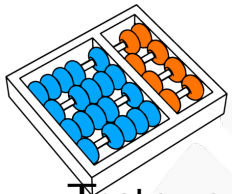


É bonito, mas faz o que promete?

```
1  def eh_primo(numero):
2      ''' Devolve se o numero dado é primo ou não.'''
3      divisor = 3
4      while divisor < numero:
5          if numero % divisor == 0:
6              return False
7          divisor += 1
8      return True
9
10 def imprime_primos(n):
11     ''' Imprime os primos menores ou iguais a n.'''
12     for numero in range(3, n, 2):
13         if eh_primo(numero):
14             print(numero)
```

O código está certo?

- Precisamos saber o algoritmo:
  - O que é divisor, primo, como verifica primalidade, etc.
- Depois precisamos saber se o código implementa o algoritmo.



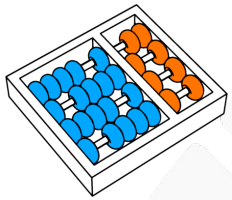
Testando

Testar aumenta a confiança que o código está correto:

```
1  def eh_primo(numero: int) -> bool:
2      ''' Devolve se o numero dado é primo ou não.'''
3      divisor: int = 3  # errado propositalmente
4      while divisor < numero:
5          if numero % divisor == 0:
6              return False
7          divisor += 1
8      return True
9
10 assert eh_primo(2) is True
11 assert eh_primo(3) is True
12 assert eh_primo(4) is False
13 assert eh_primo(17) is True
14 assert eh_primo(42) is False
```

O resultado nos ajuda a achar o erro:

```
Traceback (most recent call last):
  File "test1.py", line 12, in <module>
    assert eh_primo(4) is False
AssertionError
```

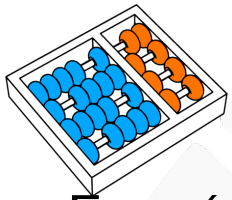


Testes de Unidade

O ruim de usar **assert** é que paramos no primeiro erro:

- Nosso código poderia ser grande.
- E estar em vários arquivos.
- E ter vários erros. . .

Vamos usar algo melhor — **pytest**.



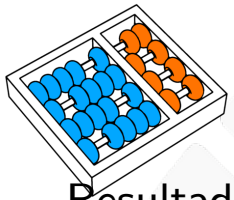
Nosso primeiro teste

Esse é o arquivo **test_primo.py**:

```
1  from primos import eh_primo
2
3  def test_eh_primo():
4      primos = [2, 3, 5, 7, 11, 13, 17]
5      for p in primos:
6          assert eh_primo(p) is True
7      compostos = [4, 6, 8, 9, 10, 12, 14, 15, 16]
8      for c in compostos:
9          assert eh_primo(c) is False
```

Para rodar o teste, executamos **pytest** no terminal:

- Executa todos os arquivos que comecem com **test_**.
- Considera que cada função começando **test_** é um teste.

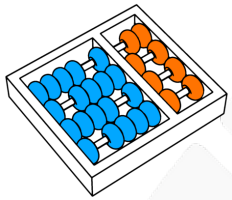


pytest

Resultado:

```
============================= test session starts ==============================
platform darwin -- Python 3.9.10, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/ana/Desktop/mc102/codigo_bom/
plugins: Faker-8.10.1
collected 1 item

test_primo.py F                                                          [100%]

=================================== FAILURES ===================================
_______________________________ test_é_primo ________________________________

    def test_eh_primo():
        primos = [2, 3, 5, 7, 11, 13, 17]
        for p in primos:
            assert eh_primo(p) is True
        compostos = [4, 6, 8, 9, 10, 12, 14, 15, 16]
        for c in compostos:
>           assert eh_primo(c) is False
E           assert True is False
E            +  where True = eh_primo(4)

test_primo.py:9: AssertionError
=========================== short test summary info ============================
FAILED test_primo.py::test_eh_primo - assert True is False
============================== 1 failed in 0.05s ===============================
```

O código está errado pois está dizendo que 4 é primo.

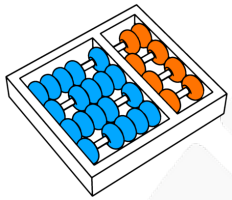


Voltando ao código

```
1  def eh_primo(numero):
2      ''' Devolve se o numero dado é primo ou não.'''
3      divisor = 3
4      while divisor < numero:
5          if numero % divisor == 0:
6              return False
7          divisor += 1
8      return True
```

O problema está na linha 3...

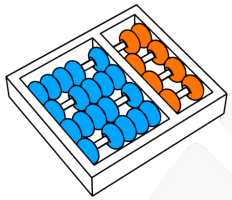
- Começamos com **divisor** igual a **3**.
- Assim, devolvemos **True** para 4.



Corrigindo

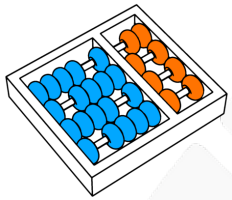
Também alteramos a função de impressão:

```python
def eh_primo(numero: int) -> bool:
    ''' Devolve se o numero dado é primo ou não.'''
    divisor: int = 2  # corrigido
    while divisor < numero:
        if numero % divisor == 0:
            return False
        divisor += 1
    return True

def primos_ate_n(n: int) -> list[int]:
    ''' Devolve a lista dos primos menores ou iguais a n.'''
    lista: list[int] = []
    for numero in range(3, n, 2):
        if eh_primo(numero):
            lista.append(numero)
    return lista

def imprime_primos(n: int) -> None:
    ''' Imprime os primos menores ou iguais a n.'''
    for numero in primos_ate_n(n):
        print(numero)
```



Aumentando o teste

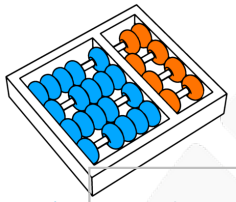
```python
from primos import eh_primo, primos_ate_n

def test_eh_primo():
    primos = [2, 3, 5, 7, 11, 13, 17]
    for p in primos:
        assert eh_primo(p) is True
    compostos = [4, 6, 8, 9, 10, 12, 14, 15, 16]
    for c in compostos:
        assert eh_primo(c) is False

def test_primos_ate_n():
    primos = [2, 3, 5, 7, 11, 13, 17]
    assert primos_ate_n(17) == primos
```



Resultado

```
============================= test session starts ==============================
platform darwin -- Python 3.9.10, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/ana/Desktop/mc102/codigo_bom/
plugins: Faker-8.10.1
collected 2 items

test_primo.py .F                                                         [100%]

=================================== FAILURES ===================================
_____________________________ test_primos_até_n _____________________________

    def test_primos_ate_n():
        primos = [2, 3, 5, 7, 11, 13, 17]
>       assert primos_ate_n(17) == primos
E       assert [3, 5, 7, 11, 13] == [2, 3, 5, 7, 11, 13, ...]
E         At index 0 diff: 3 != 2
E         Right contains 2 more items, first extra item: 13
E         Use -v to get the full diff

test_primo.py:13: AssertionError
=========================== short test summary info ============================
FAILED test_primo.py::test_primos_ate_n - assert [3, 5, 7, 11, 13] == [2, 3, ...
========================== 1 failed, 1 passed in 0.06s =========================
```

Ele não encontrou 2 e 17 na lista.

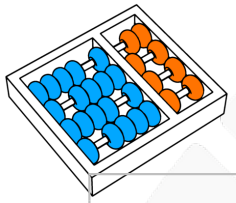


Voltando ao código

```
1  def eh_primo(numero: int) -> bool:
2      ''' Devolve se o numero dado é primo ou não.'''
3      divisor: int = 2
4      while divisor < numero:
5          if numero % divisor == 0:
6              return False
7          divisor += 1
8      return True
9
10 def primos_ate_n(n: int) -> list[int]:
11     ''' Devolve a lista dos primos menores ou iguais a n.'''
12     lista: list[int] = []
13     for numero in range(2, n + 1):  # corrigido
14         if eh_primo(numero):
15             lista.append(numero)
16     return lista
```

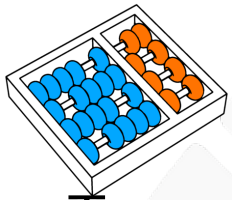
Agora sim o código está correto!

- Os testes anteriores passam!
- Mas será mesmo que está correto?
- Qual o resultado de **eh_primo(1)**? e **eh_primo(-4)**?



Versão Final

```python
def eh_primo(numero: int) -> bool:
    ''' Devolve se o numero dado é primo ou não.'''
    if numero <= 1:  # negativos, 0 e 1 não são primos
        return False
    divisor: int = 2
    while divisor < numero:
        if numero % divisor == 0:
            return False
        divisor += 1
    return True

def primos_ate_n(n: int) -> list[int]:
    ''' Devolve a lista dos primos menores ou iguais a n.'''
    lista: list[int] = []
    for numero in range(2, n + 1):
        if eh_primo(numero):
            lista.append(numero)
    return lista
```



Sobre testes

Testes é uma disciplina:

- Existem tipos diferentes de teste.
- O que vimos é basicamente teste de unidade:
  - Testamos pequenas partes do nosso código.
- Buscamos ter testes que cubra todo o código:
  - Tem como verificar isso.
- É preciso cuidado ao criar os testes.

Você precisa instalar o pytest:

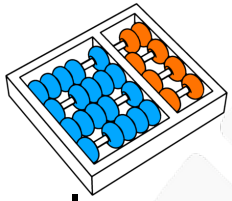
- **pip install pytest**

Ele tem outras funcionalidades:

- E existem concorrentes como o **unittest**

Em geral há uma estrutura de pastas a ser seguida.

Eficiência



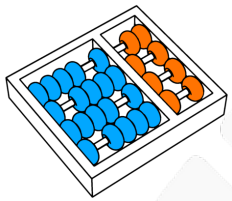
Tempo de Execução

Importamos **cProfile** e executamos
cProfile.run("primos_ate_n(100000)")

```
         109595 function calls in 25.820 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   25.820   25.820 <string>:1(<module>)
        1    0.012    0.012   25.820   25.820 teste_tempo.py:15(primos_ate_n)
    99999   25.807    0.000   25.807    0.000 teste_tempo.py:3(eh_primo)
        1    0.000    0.000   25.820   25.820 {built-in method builtins.exec}
     9592    0.001    0.000    0.001    0.000 {method 'append' of 'list' objects
     }
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
     Profiler' objects}
```

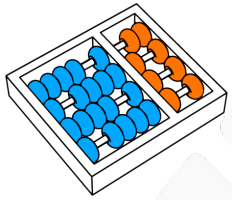
Demorou 25 segundos para rodar:

- Praticamente todo o tempo é executando **eh_primo**.
- Se melhorarmos **eh_primo**, melhoramos muito o código.



Novo código

```python
import cProfile

def eh_primo(numero: int) -> bool:
    ''' Devolve se o numero dado é primo ou não.'''
    if numero <= 1:  # negativos, 0 e 1 não são primos
        return False
    divisor: int = 2
    while divisor * divisor <= numero:  # melhor
        if numero % divisor == 0:
            return False
        divisor += 1
    return True

def primos_ate_n(n: int) -> list[int]:
    ''' Devolve a lista dos primos menores ou iguais a n.'''
    lista: list[int] = []
    for numero in range(2, n + 1):
        if eh_primo(numero):
            lista.append(numero)
    return lista

cProfile.run("primos_ate_n(100000)")
```

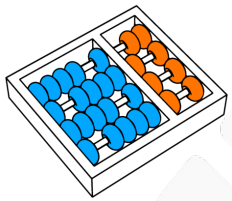


Tempo de Execução

```
         109595 function calls in 0.181 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.181    0.181 <string>:1(<module>)
        1    0.009    0.009    0.181    0.181 teste_tempo2.py:15(primos_ate_n)
    99999    0.172    0.000    0.172    0.000 teste_tempo2.py:3(eh_primo)
        1    0.000    0.000    0.181    0.181 {built-in method builtins.exec}
     9592    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects
      }
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
     Profiler' objects}
```

Rodamos em 0.181 segundos!

- Ao invés de 25.820 segundos!

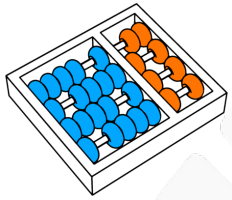


Usando um algoritmo melhor

```python
import cProfile

def primos_ate_n(n: int) -> list[int]:
    ''' Devolve a lista dos primos menores ou iguais a n.

        Implementa o Crivo de Eratóstenes.
    '''
    primos: list[int] = []
    eh_primo: list[bool] = []
    for _ in range(n + 1):  # _ é variável não usada
        eh_primo.append(True)
    p = 2
    while p <= n:
        if eh_primo[p]:
            primos.append(p)
            # se p é primo, então seus múltiplos não são
            for k in range(p * p, n + 1, p):
                eh_primo[k] = False
        p += 1
    return primos

cProfile.run("primos_ate_n(100000)")
```

O código ainda poderia ser mais “Pythonico”:

- Mas precisamos aprender mais coisas para isso.



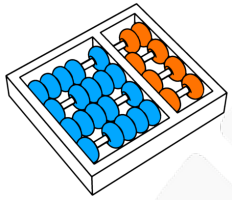
Tempo de Execução

```
         109597 function calls in 0.033 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.033    0.033 <string>:1(<module>)
        1    0.028    0.028    0.032    0.032 ultima_versao.py:4(primos_ate_n)
        1    0.000    0.000    0.033    0.033 {built-in method builtins.exec}
   109593    0.005    0.000    0.005    0.000 {method 'append' of 'list' objects
     }
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.
     Profiler' objects}
```

Rodamos em 0.033 segundos!

- Ao invés de 0.181 segundos!
- Porém estamos criando uma lista bem grande na memória:
  - Gastamos por volta de **8 * n** bytes.
  - 1Mb a cada 125 mil.
- Estamos trocando tempo por espaço.

Conclusão



Então o que é um bom código?

Um bom código é:

- Correto:
  - Implementa o algoritmo corretamente.
  - Testes ajudam a acreditar nisso.
- Rápido:
  - Rápido é relativo, depende da necessidade.
  - Melhor ser rápido onde importa.
  - Profilling ajuda a identificar gargalos.
- Fácil de ler:
  - Segue um estilo.
  - Tem bons nomes de variáveis, funções, etc.
  - Tem comentários úteis e focados.
  - Tem uma boa documentação.

Código Bom

MC102 - Algoritmos e Programação de Computadores

04/25

12

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP