Programação Script

Módulos e Classes

Aula 12

Prof. Felipe A. Louza

Roteiro

Roteiro

Módulos

Em **Python** um módulo é um arquivo com definições e comandos para serem usados em outros programas.

- Há diversos módulos do **Python** que fazem parte da biblioteca padrão.
- Já utilizamos alguns deles:

```
1 >>> import math
2 >>> math.sqrt(16)
3 4
```

Módulos

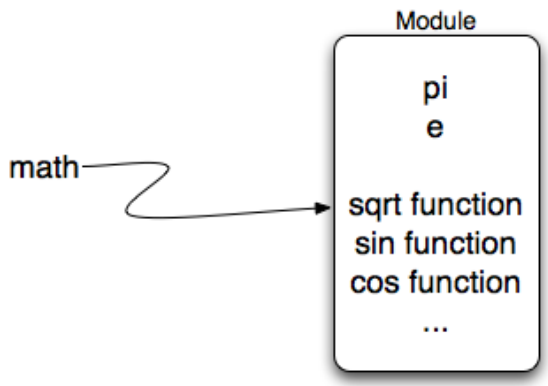
Na verdade, módulos são objetos de dados, assim como qualquer outro tipo em **Python**.

- Objetos do tipo `module` contém outros elementos do **Python** (*funções e variáveis*).

```
>>> import math
```

```
>>> type(math)
<class 'module'>
```

```
>>> math.pi
3.141592653589793
```



  - A instrução `import` cria um novo nome, `math`, que faz referência a um objeto `module`.
  - Para utilizar algo do módulo, usamos a notação de *ponto* (*namespace*).

Módulos

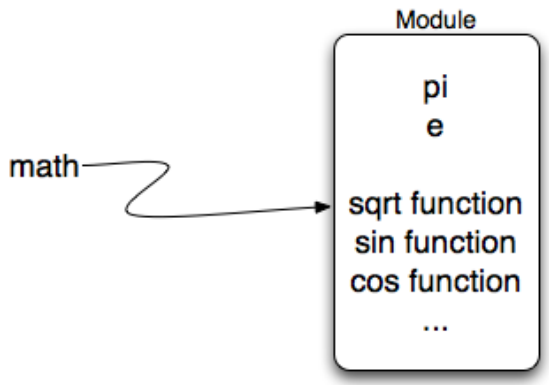
Na verdade, módulos são objetos de dados, assim como qualquer outro tipo em **Python**.

- Objetos do tipo `module` contém outros elementos do **Python** (*funções e variáveis*).

```
>>> import math
```

```
>>> type(math)
<class 'module'>
```

```
>>> math.pi
3.141592653589793
```



  - A instrução `import` cria um novo nome, `math`, que faz referência a um objeto `module`.
  - Para utilizar algo do módulo, usamos a notação de *ponto* (*namespace*).

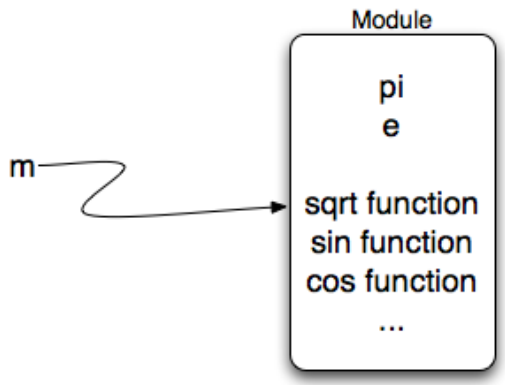
Módulos

- Podemos importar um módulo e renomear o nome do objeto `module`:

```
>>> import math as m
```

```
>>> type(m)
<class 'module'>
```

```
>>> m.pi
3.141592653589793
```



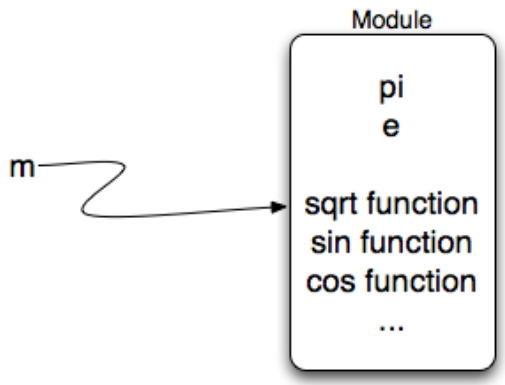
Módulos

- Podemos importar um módulo e renomear o nome do objeto `module`:

```
>>> import math as m
```

```
>>> type(m)
<class 'module'>
```

```
>>> m.pi
3.141592653589793
```



  - Cuidado, ao tentar acessar `math.pi` temos um erro.

```
>>> math.pi
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    math.pi
NameError: name 'math' is not defined
```

Módulos

- Outra possibilidade é importar `funções/variáveis` específicas do `modulo`:

```
1 >>> from math import pi, sqrt
2 >>> pi
3 3.141592653589793
4 >>> sqrt(4)
5 2.0
```

Módulos

- Outra possibilidade é importar `funções/variáveis` específicas do `modulo`:

```
1 >>> from math import pi, sqrt
2 >>> pi
3 3.141592653589793
4 >>> sqrt(4)
5 2.0
```

  - É permitido importar quase tudo[1] que um módulo define, exemplo:

    `from math import *`

  - O problema é que essa abordagem piora a legibilidade do código, nesses casos, é melhor `import math`.

[1] Exceto `declarações` que começam com `_` (underscore).

Módulos

- Outra possibilidade é importar `funções/variáveis` específicas do `modulo`:

```
1 >>> from math import pi, sqrt
2 >>> pi
3 3.141592653589793
4 >>> sqrt(4)
5 2.0
```

  - É permitido importar quase tudo[1] que um módulo define, exemplo:

    `from math import *`

  - O problema é que essa abordagem piora a legibilidade do código, nesses casos, é melhor `import math`.

[1] Exceto `declarações` que começam com `_` (underscore).

Módulos

- Podemos listar tudo o que um módulo define:

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__',
 '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees',
 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10',
 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod',
 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan',
 'tanh', 'tau', 'trunc']
```

Módulos

- Podemos listar tudo o que um módulo define:

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__',
 '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees',
 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10',
 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod',
 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan',
 'tanh', 'tau', 'trunc']
```

  - https://docs.python.org/3/library/math.html

Módulos

Outros módulos interessantes:

- `random`:

```
1 >>> import random
2 >>> random.randint(3, 9) # retorna um valor entre 3 e 9
3 4
```

  - https://docs.python.org/3/library/random.html

- `time`:

```
1 >>> import time
2 >>> time.time() # retorna o quantos segundos passados desde 01/01/1970
3 1621450935.0984983
```

  - https://docs.python.org/3/library/time.html

01/01/1970: início dos tempos para sistemas UNIX.

Módulos

Outros módulos interessantes:

- `random`:

```
>>> import random
>>> random.randint(3, 9) # retorna um valor entre 3 e 9
4
```

  - https://docs.python.org/3/library/random.html

- `time`:

```
>>> import time
>>> time.time() # retorna o quantos segundos passados desde 01/01/1970
1621450935.0984983
```

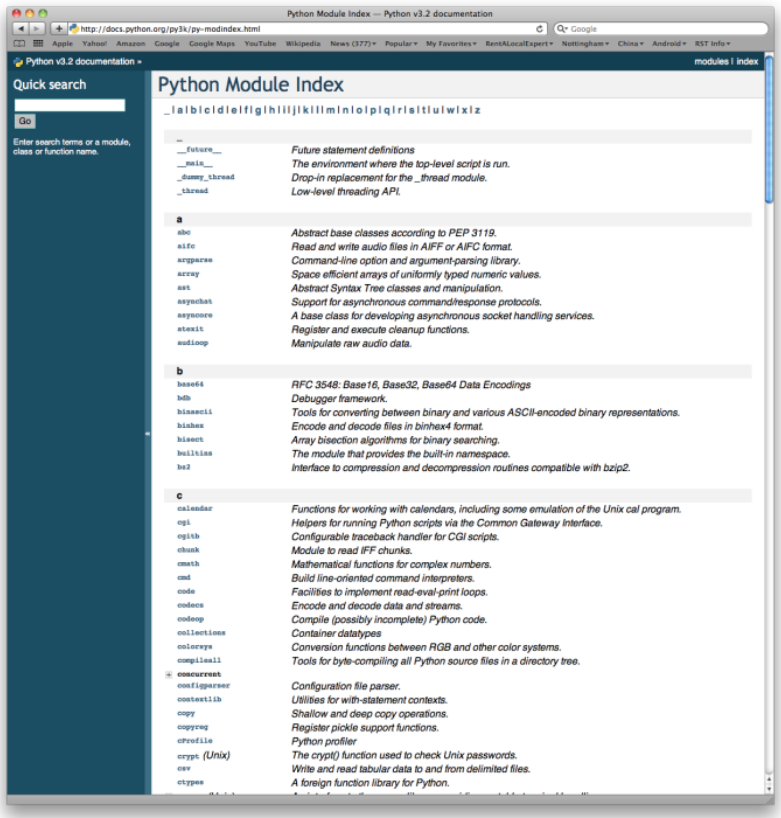
  - https://docs.python.org/3/library/time.html

01/01/1970: início dos tempos para sistemas UNIX.

Módulos

Existem muitos outros módulos disponíveis na biblioteca padrão:

- `https://docs.python.org/3/py-modindex.html`



Módulos

Podemos criar os nossos próprios módulos:

- Qualquer arquivo `.py` é um módulo em **Python**.

arquivo.py

```
1 def palindromo(s):
2   s = s.lower()
3   s = s.replace(" ", "")
4   r = s[::-1]
5   if(s == r):
6     return True
7   else:
8     return False
```

principal.py

```
1 import arquivo as arq
2
3 def main():
4   s = input("Digite s: ")
5   if(arq.palindromo(s) == True):
6     print(s,"é um palindromo")
7
8 main()
```

- Outro arquivo pode carregar esse módulo com o comando `import`.

Módulos

Podemos criar os nossos próprios módulos:

- Qualquer arquivo `.py` é um módulo em **Python**.

arquivo.py

```
1 def palindromo(s):
2   s = s.lower()
3   s = s.replace(" ", "")
4   r = s[::-1]
5   if(s == r):
6     return True
7   else:
8     return False
```

principal.py

```
1 import arquivo as arq
2
3 def main():
4   s = input("Digite s: ")
5   if(arq.palindromo(s) == True):
6     print(s,"é um palindromo")
7
8 main()
```

- Outro arquivo pode carregar esse módulo com o comando `import`.

Módulos

- Os arquivos precisam estar no mesmo diretório, caso contrário, o **Python** não encontra o módulo.

```
1 >>> import arquivo
2 Traceback (most recent call last):
3   File "<pyshell#0>", line 1, in <module>
4     import arquivo
5 ModuleNotFoundError: No module named 'arquivo'
```

  - Nesse caso, precisamos adicionar o "caminho" para o arquivo na variável `sys.path`.

Módulos

O caminho a um arquivo pode ser especificado de duas formas:

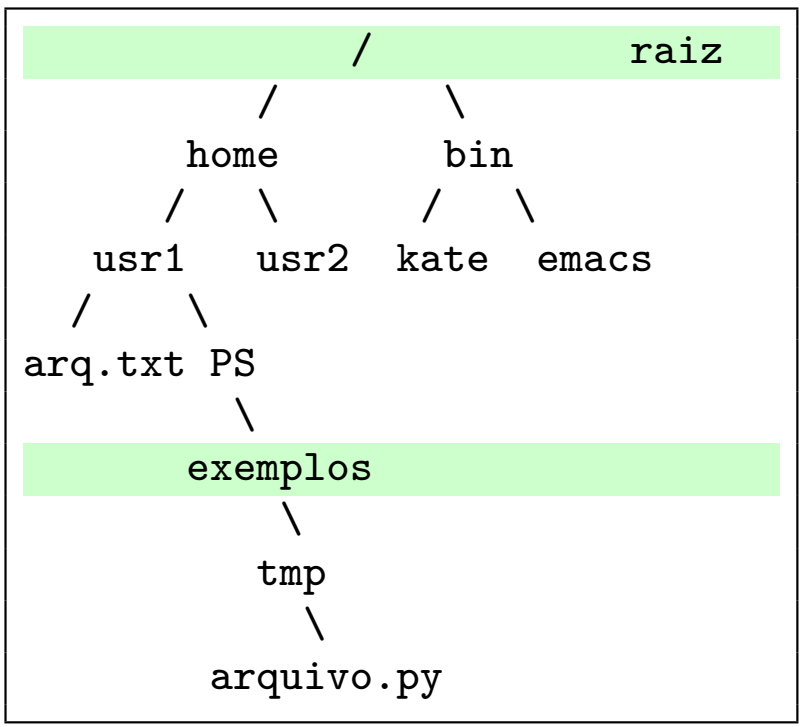
- Caminho absoluto: desde o diretório raiz.

```
1 /home/usr1/PS/exemplos/tmp/arq.py
```

- Caminho relativo: a partir do diretório corrente.

```
1 ./tmp/arq.py
```

hierarquia de diretórios



No **Linux**, para ver qual é o diretório corrente, use o comando `pwd`.

Módulos

- Para fornecer um módulo em um subdiretório, precisamos primeiro importar o módulo `sys`:

```
>>> import sys
>>> sys.path
['', '/home/usr1/PS/exemplos',
 '/usr/bin', '/usr/lib64/python38.zip', '/usr/lib64/python3.8',
 ...
 '/usr/lib64/python3.8/site-packages',
 '/usr/lib/python3.8/site-packages']
```

  - `sys.path` contém uma lista de strings que determina os caminhos de busca de módulos conhecidos pelo interpretador **Python**.

```
>>> sys.path.append("./tmp") # o modulo esta em ./tmp/arquivo.py
>>> import arquivo
>>> dir(arquivo)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'palindromo']
```

Módulos

- Para fornecer um módulo em um subdiretório, precisamos primeiro importar o módulo sys:

```
>>> import sys
>>> sys.path
['', '/home/usr1/PS/exemplos',
 '/usr/bin', '/usr/lib64/python38.zip', '/usr/lib64/python3.8',
 ...
 '/usr/lib64/python3.8/site-packages',
 '/usr/lib/python3.8/site-packages']
```

  - sys.path contém uma lista de strings que determina os caminhos de busca de módulos conhecidos pelo interpretador **Python**.

```
>>> sys.path.append("./tmp") # o modulo esta em ./tmp/arquivo.py
>>> import arquivo
>>> dir(arquivo)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
 '__name__', '__package__', '__spec__', 'palindromo']
```

Módulos

- Quando importamos um módulo, as variáveis/funções são inicializadas/declaradas e os comandos são executados.

arquivo.py

```
1  exemplo = "reviver"
2
3  def main():
4    s = input("Digite s: ")
5    if(palindromo(s)):
6      print(s,"é um palindromo")
7
8  def palindromo(s):
9    s = s.lower()
10   s = s.replace(" ", "")
11   r = s[::-1]
12   if(s == r):
13     return True
14   else:
15     return False
```

principal.py

```
1  import arquivo as arq
2
3  print(arq.exemplo)
4
5  print(arq.palindromo("ana"))
6
7  arq.main()
```

Módulos

- O problema é quando temos algum comando como a chamada da função `main()`.

arquivo.py

```
1  ...
2
3  def main():
4    s = input("Digite s: ")
5    if(palindromo(s)):
6      print(s,"é um palindromo")
7
8  def palindromo(s):
9  ...
10
11 main()
```

principal.py

```
1 import arquivo as arq
2
3 #executa o comando arq.main()
```

  - Precisamos verificar se estamos executando o código como um script ou **importando de outro lugar**.

Módulos

- Podemos fazer isso com o valor da variável global `__name__`.

arquivo.py

```
1  ...
2
3  def main():
4    s = input("Digite s: ")
5    if(palindromo(s)):
6      print(s,"é um palindromo")
7
8  def palindromo(s):
9  ...
10
11 if(__name__ == "__main__"):
12   main()
```

principal.py

```
1 import arquivo as arq
2
3 #apenas inicializa variáveis
4 #e declara funções
```

  - Quando `__name__` é igual à `__main__` o código está **sendo executado** como um script.

Módulos

As variáveis/funções são inicializadas/declaradas e os comandos são executados **uma única vez** ao importar um módulo.

- Quando modificamos um módulo precisamos recarregá-lo no interpretador **Python**.

arquivo.py

```
1  ...
2
3  def main():
4    s = input("Digite s: ")
5    if(palindromo(s)):
6      print(s,"é um palindromo")
7
8  def palindromo(s):
9  ...
10
11 if(__name__ == "__main__"):
12   main()
```

principal.py

```
1 import arquivo as arq
2
3 import importlib
4 importlib.reload(arq)
```

Módulos

Vamos redefinir a organização básica de um programa em **Python**:

```
1  import bibliotecas
2
3  variáveis globais
4
5  def main():
6    variáveis locais
7    Comandos Iniciais
8
9  def fun1(Parâmetros):
10   variáveis locais
11   Comandos
12
13 ...
14
15 if(__name__ == "__main__"):
16   main()
```

- O nosso programa pode **ser executado** como um script, ou **importado** como um módulo.

Módulos

Algumas vantagens:

- Permite o reaproveitamento de código (minimiza erros e facilita alterações)
- Separa o programa em partes que possam ser logicamente compreendidas de forma isolada.
- Reuso de código.

Mais sobre módulos:

- `https://docs.python.org/3/tutorial/modules.html`

Módulos

Algumas vantagens:

- Permite o reaproveitamento de código (minimiza erros e facilita alterações)
- Separa o programa em partes que possam ser logicamente compreendidas de forma isolada.
- Reuso de código.

Mais sobre módulos:

- `https://docs.python.org/3/tutorial/modules.html`

Roteiro

Exemplo 1: Verificando um palindromo

Escreva um programa `principal.py`, que recebe um `texto` passado pelo usuário via linha de comando e responde se o `texto` é ou não um **palindromo**.

```
$ python3 principal.py reviver
reviver é um palindromo
```

Exemplo 1: Verificando um palindromo

- Vamos reutilizar o módulo `arquivo.py`.

`arquivo.py`

```
1  exemplo = "reviver"
2
3  def main():
4    s = input("Digite s: ")
5    if(palindromo(s)):
6      print(s,"é um palindromo")
7
8  def palindromo(s):
9    s = s.lower()
10   s = s.replace(" ", "")
11   r = s[::-1]
12   if(s == r): return True
13   else: return False
14
15 if(__name__ == "__main__"):
16     main()
```

`principal.py`

```
1  import arquivo as arq
2
3  def main():
4    ...
5    ...
6    ...
7
8  if(__name__ == "__main__"):
9      main()
```

Exemplo 1: Verificando um palindromo

- O módulo sys permite analisar os parâmetros passados pelo usuário via linha de comando.

```
1 import arquivo as arq
2 import sys
3
4 def main():
5   for i in range(0,len(sys.argv)):
6     print(sys.argv[i])
7
8 if(__name__ == "__main__"):
9     main()
```

  - sys.argv é uma lista de strings com os argumentos passados via linha de comando.

```
1 $ python3 argv.py abc de 123
2 argv.py
3 abc
4 de
5 123
```

Exemplo 1: Verificando um palindromo

- Para testar um texto passado via linha de comando, precisamos concatenar as strings

`sys.argv[1], sys.argv[2], ...`.

```
1  import sys
2  import arquivo as arq
3
4  def main():
5    s = ""
6    l = len(sys.argv)
7    for i in range(1,l):
8      s += sys.argv[i]
9    print(s)
10   ...
```

Exemplo 1: Verificando um palindromo

- Por fim, basta chamar a função do módulo `arq`:

```
1  import sys
2  import arquivo as arq
3
4  def main():
5    s = ""
6    l = len(sys.argv)
7    for i in range(1,l):
8      s += sys.argv[i]
9
10   if(arq.palindromo(s)):
11     print(s,"é um palindromo")
12   else:
13     print(s,"não é um palindromo")
14
15 if(__name__ == "__main__"):
16   main()
```

Exemplo 2: Módulo `circulo.py`

Escreva um módulo `circulo.py` que define funções para calcular a **área**, o **perímetro** e o **diametro** de uma circunferência.

Escreva um programa em **Python** que recebe o raio via linha de comando e exibe um *menu* para o usuário. Exemplo:

```
$ python3 principal.py 10
Qual operação deseja realiza?
1) área
2) perímetro
3) diametro
4) sair
_
```

- Quando o usuário escolher a opção, exiba o resultado.

Exemplo 2: Módulo `circulo.py`

- Primeiro, escrevemos `circulo.py`.

`circulo.py`

```
1  import math
2
3  def area(r):
4    return math.pi*r*r
5
6  def perimetro(r):
7    return 2*math.pi*r
8
9  def diametro(r):
10   return 2*r
```

`principal.py`

```
1  import circulo
2  import sys
3
4  def main():
5    ...
6    ...
7    ...
8
9  if(__name__ == "__main__"):
10     main()
```

Exemplo 2: Módulo `circulo.py`

- Primeiro, escrevemos `circulo.py`.

`circulo.py`

```
1  import math
2
3  def area(r):
4    return math.pi*r*r
5
6  def perimetro(r):
7    return 2*math.pi*r
8
9  def diametro(r):
10   return 2*r
```

`principal.py`

```
1  import circulo
2  import sys
3
4  def main():
5    ...
6    ...
7    ...
8
9  if(__name__ == "__main__"):
10     main()
```

Exemplo 2: Módulo `circulo.py`

- Vamos definir uma função para o menu:

```
1  import circulo
2  import sys
3
4  def menu():
5    print("Qual operação deseja realiza?")
6    print("1) área")
7    print("2) perímetro")
8    print("3) diametro")
9    print("4) sair")
10   return int(input())
11
12 def main():
13   ....
14   ....
15   ....
16
17
18 if(__name__ == "__main__"):
19     main()
```

Exemplo 2: Módulo `circulo.py`

- Precisamos verificar se o usuário passou o valor do raio:

```
1  import circulo
2  import sys
3
4  def menu():
5    ...
6
7  def main():
8    if(len(sys.argv)!=2):
9      print("Informe o raio")
10     return None
11   raio = int(sys.argv[1])
12
13   ...
14
15 if(__name__ == "__main__"):
16     main()
```

Exemplo 2: Módulo `circulo.py`

- Finalmente, executamos a função principal:

```
1  import circulo
2  import sys
3
4  def menu():
5    ...
6
7  def main():
8    ...
9
10   op = menu()
11   while(op != 4):
12     if(op == 1): print(circulo.area(raio))
13     elif(op == 2): print(circulo.perimetro(raio))
14     elif(op == 3): print(circulo.diametro(raio))
15     else: print("Operação inválida")
16     op = menu()
17     print()
18
19 if(__name__ == "__main__"):
20     main()
```

Roteiro

Classes

Python é uma linguagem de programação orientada a objetos (**POO**):

- **POO** é um outro[2] paradigma de desenvolvimento de software muito utilizado em sistemas grandes e complexos.
- O foco é na definição de objetos que contem tanto os dados quanto as funcionalidades.

Vamos ver (apenas) alguns conceitos de **POO** em **Python**.

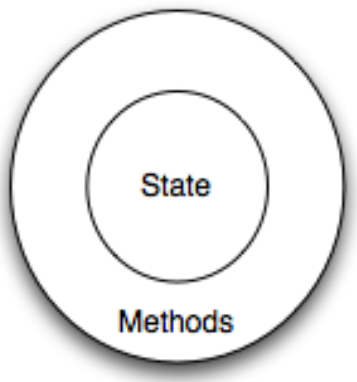
[2] Diferente de programação script/procedural.

Classes

Em **Python** todo valor é na verdade um **objeto**.

- Seja uma `string`, `lista`, ou mesmo um `inteiro`.
- Dizemos que um objeto possui um **estado** (*atributos*) e uma **coleção de funções** (*métodos*) que ele pode executar.

```
1 >>> texto = "laranja"
2 >>>
3 >>> texto.islower()
4 True
5 >>> texto.__class__
6 <class 'str'>
```



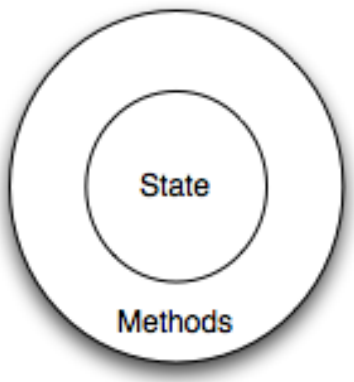
– O "tipo" de um **objeto** é chamado de **classe**.

Classes

Em **Python** todo valor é na verdade um **objeto**.

- Seja uma `string`, `lista`, ou mesmo um `inteiro`.
- Dizemos que um objeto possui um **estado** (*atributos*) e uma **coleção de funções** (*métodos*) que ele pode executar.

```
1 >>> texto = "laranja"
2 >>>
3 >>> texto.islower()
4 True
5 >>> texto.__class__
6 <class 'str'>
```



  - O "tipo" de um **objeto** é chamado de **classe**.

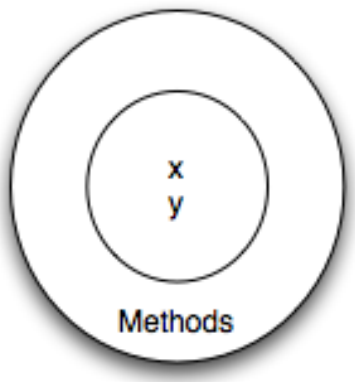
Classes

Podemos criar as nossas próprias `classes`:

- Como exemplo, considere o conceito de **um ponto** com duas dimensões `(x, y)`.

```
>>> class Ponto:
    pass #classe vazia

# novo objeto do tipo ponto,
# ou instância
>>> p = Ponto()
>>> p.x = 0
>>> p.y = 0
```



  - O **Python** permite adicionar os atributos dinamicamente.
  - As definições de classes podem aparecer em qualquer lugar em um programa.

Classes

- Mas, outras **instâncias** da classe `Ponto` podem não ter os **atributos** `x` e `y`:

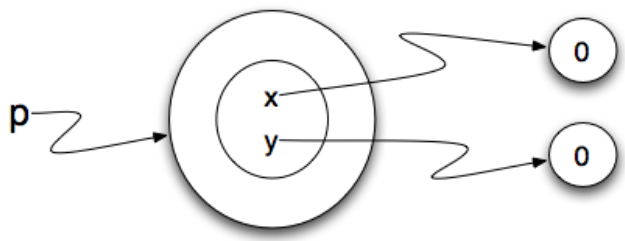
```
1 >>> q = Ponto()
2 >>> q.x
3 Traceback (most recent call last):
4   File "<pyshell#34>", line 1, in <module>
5     q.x
6 AttributeError: 'Ponto' object has no attribute 'x'
```

  - O **Python** permite adicionar os atributos dinamicamente.

Classes

- Vamos criar nossas próprias `classes` e definir os atributos em um método especial, chamado de construtor: `__init__()`.

```
1  >>> class Ponto:
2      def __init__(self):
3        self.x = 0
4        self.y = 0
5
6  # novo objeto do tipo ponto,
7  # ou instância
8  >>> p = Ponto()
9  >>> p.x
10 0
11 >>> p.y
12 0
```



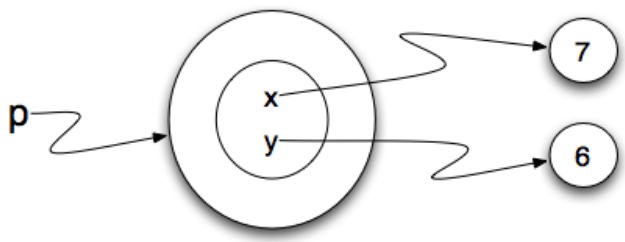
  - O construtor é chamado automaticamente sempre que uma nova instância é criada.
  - O parâmetro `self` faz referencia o objeto recém-criado.

Classes

- Podemos passar os valores de `x` e `y` para o construtor:

```
>>> class Ponto:
    def __init__(self, a, b):
      self.x = a
      self.y = b

# novo objeto do tipo ponto,
# ou instância
>>> p = Ponto(7, 6)
>>> p.x
7
>>> p.y
6
```

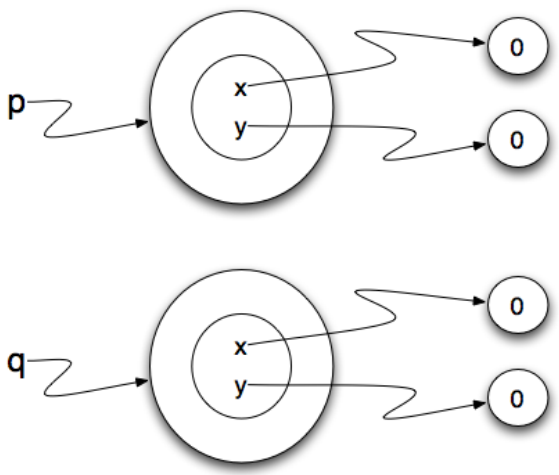


  - Essa é uma boa prática.

Classes

- Cada instância é um **novo objeto** na memória:

```
1  >>> class Ponto:
2      def __init__(self, a, b):
3        self.x = a
4        self.y = b
5
6  # novo objeto do tipo ponto,
7  # ou instância
8  >>> p = Ponto(0, 0)
9  >>> q = Ponto(0, 0)
10 >>>
11 >>> p is q
12 False
```



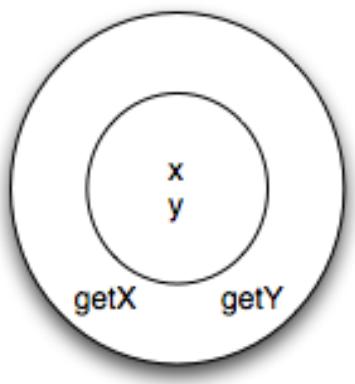
  - A classe em si não é uma instância de um ponto, mas contém as informações de como criar pontos.

Classes

Podemos acrescentar métodos que são operações úteis para trabalhar com os **objetos** da classe.

- Um método se comporta como uma função, mas ele é chamado de uma **instância específica**.

```
1  >>> class Ponto:
2      def __init__(self, a, b):
3        self.x = a
4        self.y = b
5
6      def getX(self):
7          return self.x
8
9      def getY(self):
10         return self.y
```



  - **Todo método** definido que opere em objetos dessa classe terá `self` como seu primeiro parâmetro.

Classes

- Ao executar um método, temos o equivalente à chamada da função passando o objeto como parâmetro.

```
>>> class Ponto:
    def __init__(self, a, b):
      self.x = a
      self.y = b

    def getX(self):
        return self.x

    def getY(self):
        return self.y
```

```
>>> p = Ponto(7, 6)
>>> p.getX()
7
>>> Ponto.getX(p)
7
```

  - O `self` serve como referência para o objeto em si, que por sua vez permite o acesso aos dados no interior do objeto.

Classes

- Outros métodos: `distanceFromOrigin()`

```
>>> class Ponto:
    def __init__(self, a, b):
      self.x = a
      self.y = b

    ...
    ...

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

>>> p = Ponto(7,6)
>>> print(p.distanceFromOrigin())
9.219544457292887
```

Classes

Podemos *sobreescrever* a função que converte um **objeto** para `str`.

```
>>> class Ponto:
    def __init__(self, a, b):
      self.x = a
      self.y = b

    ...

    def __str__(self):
        return "({}, {})".format(str(self.x), str(self.y))

>>> p = Ponto(7, 6)
>>> print(p)
(7, 6)
>>> str(p)
'(7, 6)'
```

- Vamos incluir um método especial: `__str__`.

Classes

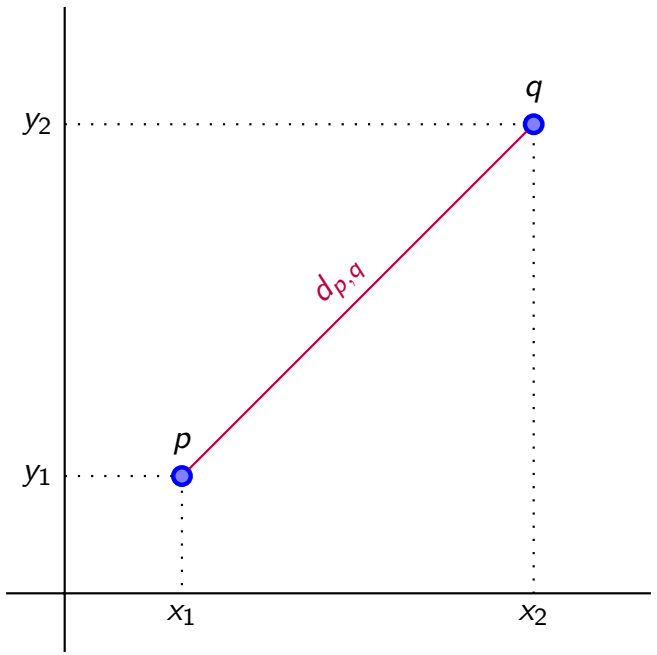
- Existem muitos outros conceitos de **POO**: herança, polimorfismo, classe abstrata, interface, ...

Roteiro

Exemplo 3: Distância entre pontos

Adicione na classe `Ponto` um **método** que calcula a distância euclidiana entre a instância p e outro ponto q.

```
>>> p = Ponto(4,3)
>>> q = Ponto(0,0)
>>> p.distance(q)
5.0
```



Exemplo 3: Distância entre pontos

- O método recebe como parâmetro outro **objeto** do tipo `Ponto`.

```
>>> class Ponto:
  def __init__(self, a, b):
    self.x = a
    self.y = b

  ...
  ...

  def distance(self, q):
    distx = q.getX() - self.getX()
    disty = q.getY() - self.getY()
    dist = (distx**2 + disty**2) ** 0.5
    return dist
```

Exemplo 3: Distância entre pontos

- Podemos testar com o método `distanceFromOrigin()`

```
>>> p = Ponto(4,3)
>>> q = Ponto(0,0)
>>> p.distance(q)
5.0
>>> p.distanceFromOrigin()
5.0
```

Exemplo 4: Lista randomizada

Escreva uma **classe** que define uma lista com n numeros inteiros aleatórios no intervalo $[0, m]$.

```
>>> lista = ListaAleatoria(10, 100)
...
[14, 41, 95, 67, 4, 96, 67, 68, 81, 91]
```

Exemplo 4: Lista randomizada

- Vamos importar o módulo `random`:

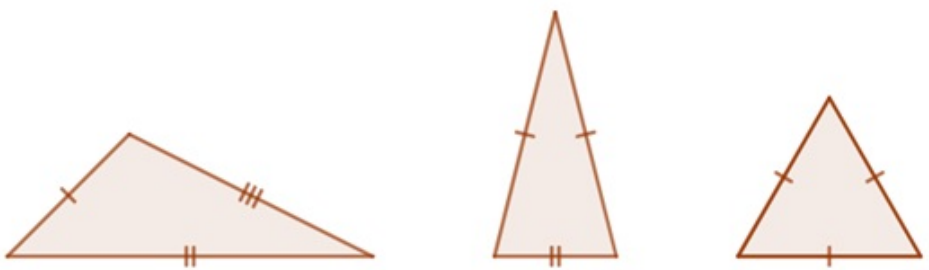
```
1  import random
2
3  class ListaAleatoria:
4
5    def __init__(self, n, m):
6      self.lista = []
7      for i in range(n):
8        self.lista.append(random.randint(1, m))
9
10   def getLista(self):
11     return self.lista
```

```
1  >>> l = ListaAleatoria(10, 100)
2  >>> l.getLista()
3  [14, 41, 95, 67, 4, 96, 67, 68, 81, 91]
```

Exemplo 5: Tipos de triângulos

Escreva uma **classe** que define um triângulo com o tamanho dos seus lados.
A função `print()` deve informar o tipo do triângulo.

```
>>> t = Triangulo(5, 5, 1)
>>> print(t)
'isósceles'
```



Exemplo 5: Tipos de triângulos

```
1  class Triangulo:
2    def __init__(self, a, b, c):
3      self.a = a
4      self.b = b
5      self.c = c
6
7    def tipo_lado(self):
8      if(self.a == self.b == self.c):
9        return "equilátero"
10     elif(self.a == self.b or self.b == self.c or self.a == self.c):
11       return "isósceles"
12     else:
13       return "escaleno"
```

- Precisamos sobrescrever o método `__str__`:

```
1    def __str__(self):
2        return self.tipo_lado()
```

Exemplo 5: Tipos de triângulos

```
1  class Triangulo:
2    def __init__(self, a, b, c):
3      self.a = a
4      self.b = b
5      self.c = c
6
7    def tipo_lado(self):
8      if(self.a == self.b == self.c):
9        return "equilátero"
10     elif(self.a == self.b or self.b == self.c or self.a == self.c):
11       return "isósceles"
12     else:
13       return "escaleno"
```

- Precisamos sobrescrever o método `__str__`:

```
1    def __str__(self):
2        return self.tipo_lado()
```

Fim

Dúvidas?

Leitura complementar

Leitura complementar:

1. https://docs.python.org/3/tutorial/modules.html
2. https://docs.python.org/3/tutorial/classes.html

Roteiro

Referências

1. Materiais adaptados dos slides do Prof. Eduardo C. Xavier, da Universidade Estadual de Campinas.
2. panda.ime.usp.br/pensepy/static/pensepy/index.html