Introdução a Python

IPL 2021

MIT BRAZIL

UNICAMP

Boas práticas de programação

– Facilita alterações
– Reduz número de erros
– Melhora legibilidade

- Idealmente, software deveria ter três características principais:

SFB (safe from bugs): *correctness* (comportamento atual correto) e *defensiveness* (futuro comportamento correto)

ETU (easy to understand): código deve comunicar seu comportamento para programadores futuros que precisam entendê-lo e modificá-lo

RFC (ready for change): software sempre muda. Alguns designs permitem mudança facilmente, enquanto outros causam trabalho extra

Diretrizes de estilo

- Embora bom estilo seja ultimamente subjetivo, existem algumas noções razoavelmente bem aceitas:

Nomes importam
- Nomes para funções devem descrever o que fazem
- Nomes de variáveis devem descrever o que representam

DRY (Don't repeat yourself)
- Pedaços de código não devem conter lógica redundante: use um loop, função ou variável
- Se copiando/colando trechos: espaço para uma função
- Se usando o mesmo cálculo curto: salve o intermediário

Generalidade
- Defina funções de forma geral e deixe as entradas tratarem de casos específicos
- Por exemplo, defina uma função **pow** de exponenciação genérica ao invés de funções **square**, **cube**, …

Planeje para mudança
- Crie programas fáceis de mudar
- Não use funções muito longas, mas sim funções sequenciais, para ter acesso a intermediários

Reduz quantidade de código: **menos linhas é mais!**

Resolvendo um problema

1. Entenda o problema

- Qual problema você quer resolver?
- Qual a **entrada** e a **saída**?
- Quais são exemplos de entrada/saída para testar?

2. Faça um plano

- Quais **operações** precisam ser realizadas para produzir a saída? Como testar essas operações?
- Que informações, além das entradas, vamos precisar? Como colocá-las em código?
- Já viu algo parecido com esse problema antes?
- O problema pode ser dividido em **partes modulares mais simples** de resolver?
- O plano cobre **todas as entradas possíveis** e dá as saídas corretas?

3. Implemente o plano

- Traduza seu plano para Python
- Tente implementar as operações importantes como **funções individuais** para criar modularidade e evitar repetição
- Conforme avança pelo código, **verifique cada etapa**. Tenha certeza que a etapa é correta antes de continuar para minimizar a chance de erros difíceis de corrigir no futuro

4. Olhe para trás e revise

- **Teste o programa** com os casos que descobriu no começo para checar que é correto: para cada caso, execute-o e garanta que o resultado é correto
- Há outra maneira de resolver o problema? Quais as vantagens e desvantagens em relação a sua solução?
- O resultado é útil para algum outro problema? A estrutura é aplicável em outros problemas?
- Procure oportunidades para **melhorar o estilo do seu código**: nomes de funções e variáveis, cálculos repetidos, funções ou trechos que podem ser generalizados (e.g. quebrar uma operação grande em partes menores)

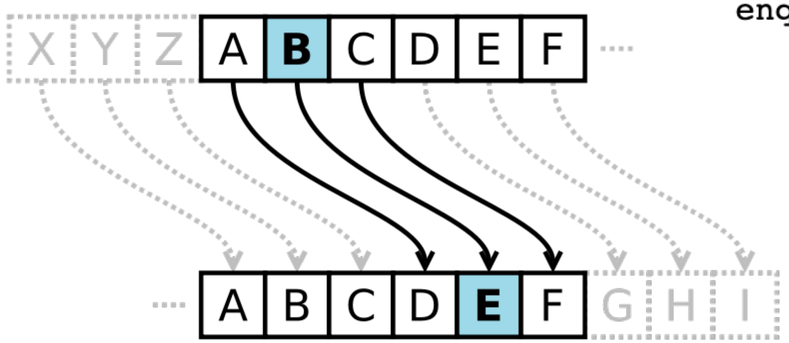
Exemplo: cifra de César

1. Entenda o problema

- Produz uma string criptografia deslocando caracteres por um valor **shift**
- Letras, números, pontuação são todos tratados diferentemente: um bom exemplo é **“376 tuna fish?!”**, que dá **“598 vwpc hkuj?!”** com **shift=2**
- Precisamos também testar casos com **shift** menor que -26, entre -26 e -10, entre -10 e 0, entre 10 e 26, e maior que 26 para garantir que a função funciona corretamente para números e letras para qualquer valor de **shift**

2. Faça um plano

- Dificuldade: cada tipo tem um tratamento diferente. Então separamos cada (letra, número, pontuação) em uma função própria
- Para construir a saída, precisamos manter a saída deslocada “até agora”. Podemos usar uma string vazia e adicionar caracteres conforme avançamos pela mensagem original
- Aqui escreveríamos pseudocódigo antes de converter nossa solução para Python:




```
defina o resultado como uma string vazia.
enquanto ainda tenhamos mais caracteres de entrada a considerar, faça o seguinte:
    pegue o próximo caractere da entrada
    se o caractere for uma letra:
        mude o caractere de acordo com as regras de letra
    caso contrário, se o caractere for um número:
        mude o caractere de acordo com as regras numéricas
    case contrário,
        não mude o caractere
    em todos os casos, adicione o caractere deslocado ao final da string de resultado
retornar a string resultante
```


MIT BRAZIL UNICAMP

Exemplo: cifra de César

3. Implemente o plano

```python
from string import ascii_lowercase, digits


def handle_letter(char, shift):
    """
    Helper function for Caesar Cipher. Handles the case the character is a letter.

    Args:
        char: length-one string, letter character
        shift: shift value of caesar cipher

    Returns:
        new letter that should be placed in cryptographed message
    """
    orig_num_value = ascii_lowercase.find(char)
    new_value = (orig_num_value + shift) % len(ascii_lowercase)
    return ascii_lowercase[new_value]


def handle_number(char, shift):
    """
    Helper function for Caesar Cipher. Handles the case the character is a number.

    Args:
        char: length-one string, number character
        shift: shift value of caesar cipher

    Returns:
        new number (as str) that should be placed in cryptographed message
    """
    orig_num_value = int(char)
    new_value = (orig_num_value + shift) % len(digits)
    return str(new_value)

```

```python
def caesar_cipher(msg, shift):
    """
    Implements a simple Caesar
    English alphabet, 0-9 digi

    Args:
        msg: message to be "cr
        shift: integer shift v

    Returns:
        cryptographed message t
    """
    msg = msg.lower()  # passar string para minúsculas
    out = ""

    for char in msg:
        if char in ascii_lowercase:
            out += handle_letter(char, shift)
        elif char in digits:
            out += handle_number(char, shift)
        else:
            out += char

    return out

```

4. Olhe para trás e revise

- Aqui usaríamos nossos casos de teste, checando os resultados um por um
- Poderíamos também notar problemas estilísticos (por exemplo, código repetido), a que podemos voltar para melhorar o programa

Classes e instâncias

- Objetos são centrais em Python e consistem de um tipo (determina as operações possíveis) e um valor
- Alternativamente, podemos chamar o tipo de **classe** do objeto e dizer que o objeto em si é uma **instância**
  - Por exemplo, **int** é uma classe e algumas de suas instâncias são **32**, **3294**, **-234**
- Até aqui só trabalhamos com tipos integrados de Python
- Porém, podemos criar *nossos próprios tipos (ou classes) de objetos* em Python
  - Permite representarmos objetos mais específicos de forma muito mais conveniente: podemos salvar *atributos* importantes e criar operações específicas para esse tipo

Vetor

- Precisamos armazenar as coordenadas do vetor
- Operações incluem soma de vetores, produto escalar, magnitude

Triângulo

- Precisamos armazenar as coordenadas dos vértices
- Operações incluem calcular área, classificar quanto a lado/ângulo

Música

- Precisamos armazenar letra, autor, áudio, duração, álbum, ano
- Operações incluem tocar a música, mostrar a letra ou outros dados

Definindo classes

- Vamos usar o exemplo de um ponto (coordenadas) para criar nossa primeira classe
- Originalmente, poderíamos representar pontos como:
  - duas variáveis separadas **x** e **y**
  - elementos de uma tupla ou lista
- Alternativamente, podemos *criar um novo tipo* para representar pontos. Essa abordagem é mais complexa, mas tem várias vantagens que veremos

```
1    x, y = 10, 20
2
3    coords = (10, 20)
4
5
6    class Point:
7        pass
8
```

Ambos os métodos começam a ficar bastante complicados quando tivermos vários pontos!

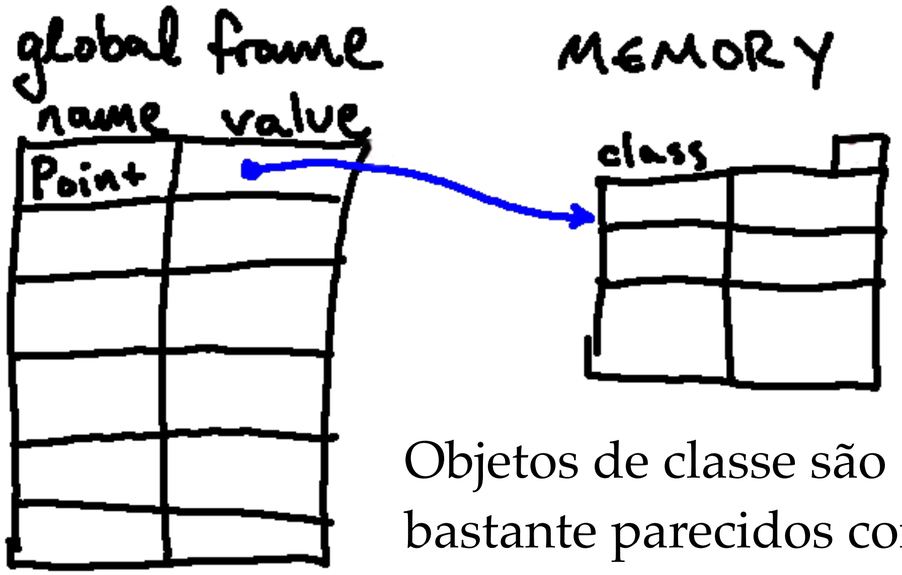
Classes são mais robustas e escalam mais facilmente

Sintaxe de criação de classes

- Novas classes são definidas com a palavra-chave **class**, seguida pelo nome dado e um corpo indentado

Classes em diagramas de ambiente

- A definição leva Python a criar um *objeto de classe* para representar o tipo e associar o nome **Point** a ele no quadro em que a classe foi definida



Objetos de classe são bastante parecidos com quadros ou dicionários

MIT BRAZIL

UNICAMP

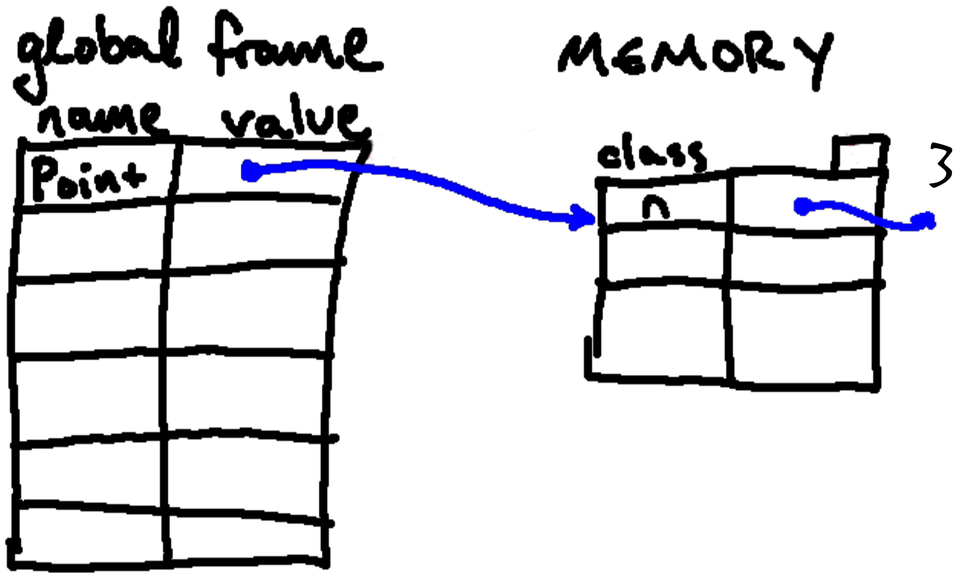
Criando classes

- Podemos definir variáveis e funções dentro de uma definição de classe: quando o objeto de classe é criado, Python executa o corpo da classe *dentro desse ambiente*
  - Podemos definir variáveis dentro da definição de classe, geralmente para valores que são comuns a todas as instâncias de uma classe
  - Essas variáveis são chamadas de **atributos**

Notação de Ponto

- Podemos procurar e/ou modificar atributos dentro de uma classe usando a mesma notação de ponto de módulos, por exemplo **Point.n**
  - Python primeiro procura **Point** e então procura o nome **n** dentro desse objeto, encontrando **2**
- Também podemos realizar atribuições usando notação semelhante, como **Point.n = 3**

```
1   class Point:
2       n = 2
3
4
5   print(Point)  # objeto de classe
6   print(Point.n)  # 2
7
8   Point.n = 3
9   print(Point.n)  # 3
10
```



MIT BRAZIL

UNICAMP

Criando instâncias

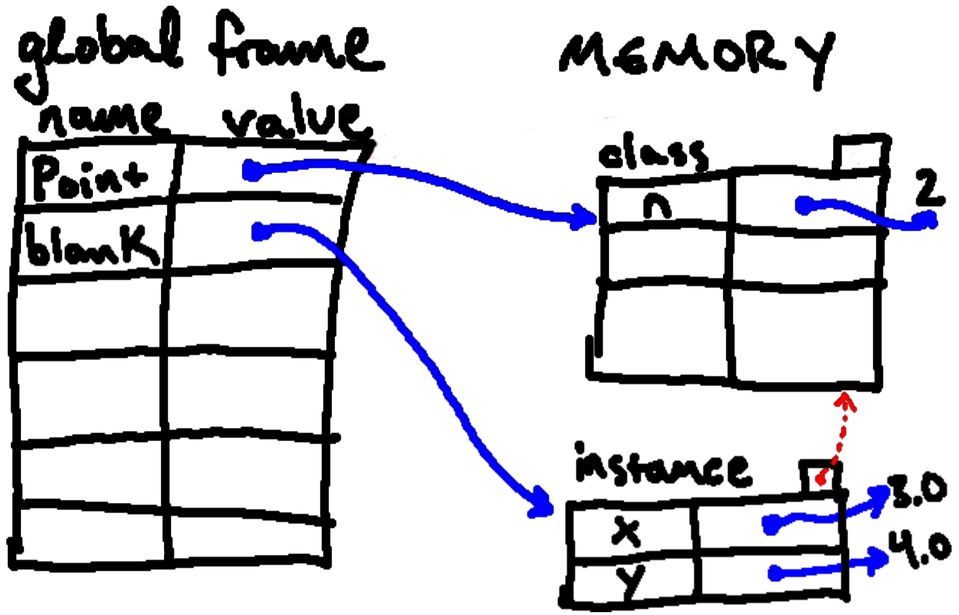
- O objeto de classe é uma fábrica para a criação de objetos. Para criar um Ponto, chamamos **Point** como se fosse uma função: **blank = Point( )**
  - O valor de retorno é um objeto de tipo **Point**
- Esse processo é chamado de *instanciação* e o objeto gerado é uma *instância* da classe

Representação de instâncias

- Instâncias serão representadas de forma semelhante, mas serão rotuladas como instâncias e seus ponteiros parentais apontarão *para o objeto de classe* do tipo associado

Notação de ponto

- Podemos procurar e/ou modificar atributos dentro de uma instância da mesma forma que faríamos com classes
- Esses elementos são chamados de **atributos** ou variáveis de instância

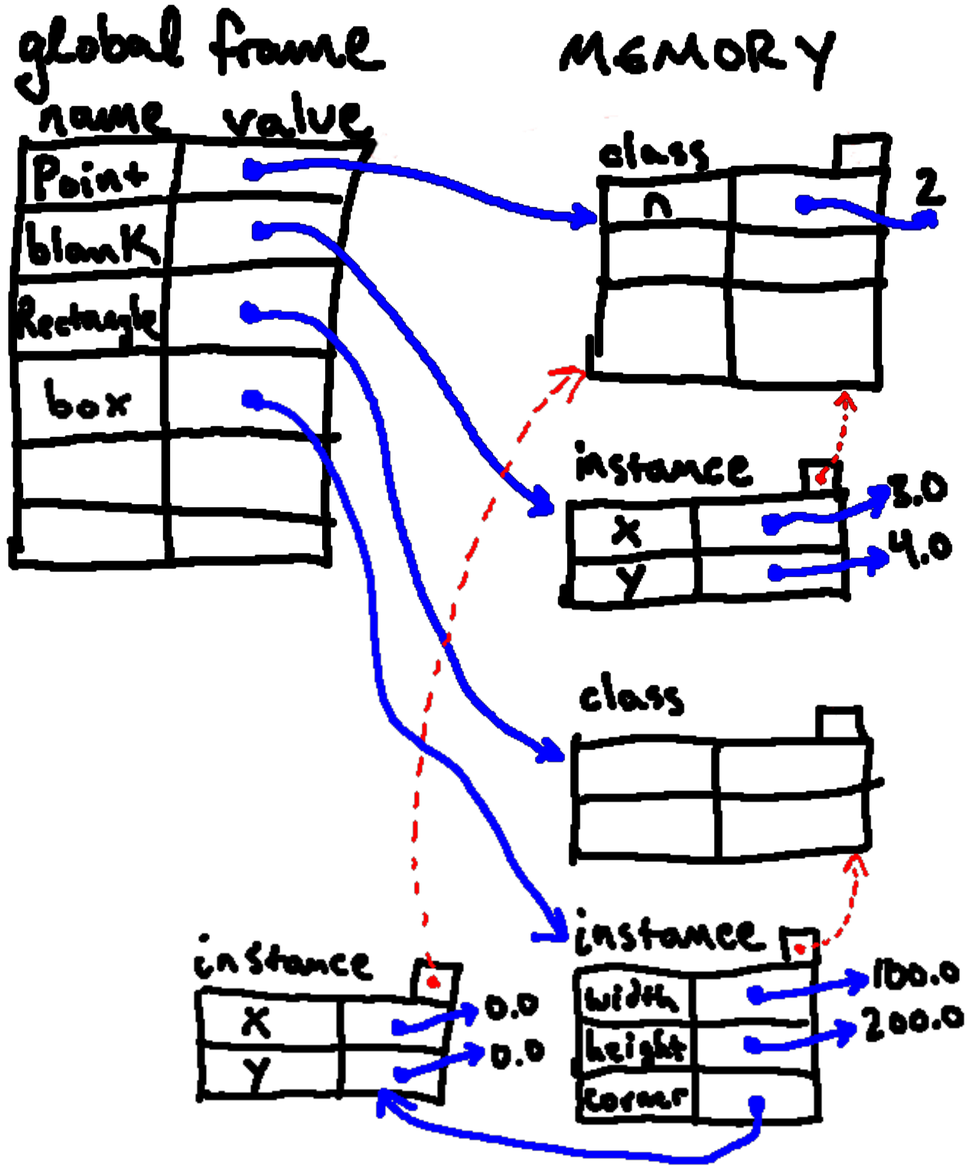



```python
class Point:
    n = 2


print(Point)  # objeto de classe
print(Point.n)  # 2

blank = Point()
blank.x = 3.0
blank.y = 4.0

print(blank.y)  # 4.0
x = blank.x
print(x)  # 3.0

print((blank.x ** 2 + blank.y ** 2) ** 0.5)

print(blank.n)  # 2 (encontra na classe)

a = 10
# print(blank.a)  AttributeError, não definida
# Python *não* procurará no quadro global,
# mas parará no objeto de classe

```

Exemplo

```python
class Point:
    n = 2


print(Point)  # obj. classe
print(Point.n)  # 2

blank = Point()
blank.x = 3.0
blank.y = 4.0

print(blank.y)  # 4.0
x = blank.x
print(x)  # 3.0

print(blank.n)  # 2
```

```python

class Rectangle:
    pass


box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

print(box.width)  # 100.0
print(box.corner.x)  # 0.0
print(box.corner.n)  # 2
# ambos AttributeError abaixo
# print(box.corner.box)
# print(box.x)
```



MIT BRAZIL

UNICAMP

Classes e funções

- Instâncias de classes definidas pelo programador podem ser tratadas como objetos primitivos
- Então podemos fazer funções que operam em instâncias ou que retornam novas instâncias da classe

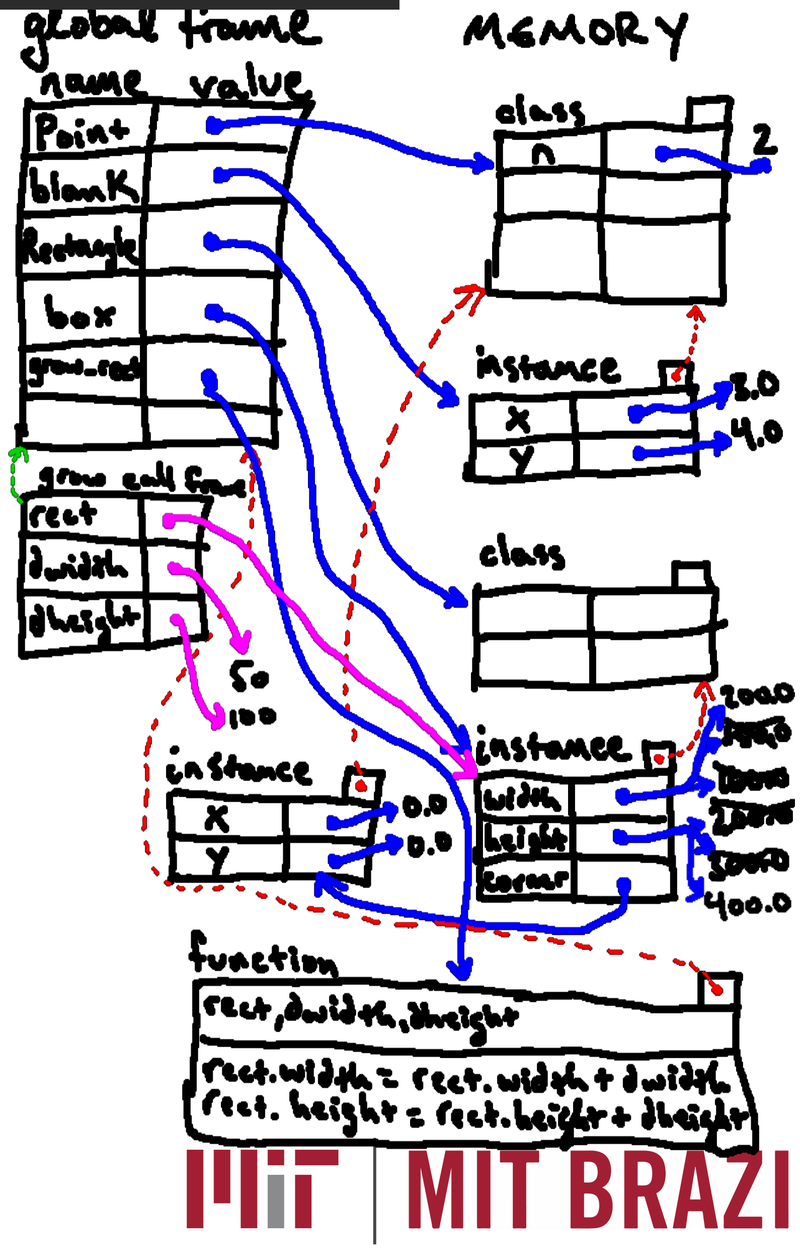
Funções que modificam objetos

- Objetos são mutáveis: podemos alterar o estado fazendo uma atribuição a um de seus atributos
- Podemos escrever funções que modificam objetos diretamente

Funções puras

- Ao invés de modificar a instância passada como argumento, retornam um novo resultado

```
9   def grow_rectangle(rect, dwidth, dheight):
10      rect.width = rect.width + dwidth
11      rect.height = rect.height + dheight
    grow_rectangle(box, 50, 100)
```




```
14  def shifted_rectangle(rect, dx, dy):
15      new_rect = Rectangle()  # cria uma nova instância
16      # altura e largura devem ser iguais
17      new_rect.width = rect.width
18      new_rect.height = rect.height
19      # o canto do retângulo é diferente, contudo
20      # fazemos uma nova instância Point para o canto
21      new_rect.corner = Point()
22      new_rect.corner.x = rect.corner.x + dx
23      new_rect.corner.y = rect.corner.y + dy
24
25      return new_rect
26
```


MIT BRAZIL

UNICAMP