Introdução a Python

IPL 2021

MIT BRAZIL

UNICAMP

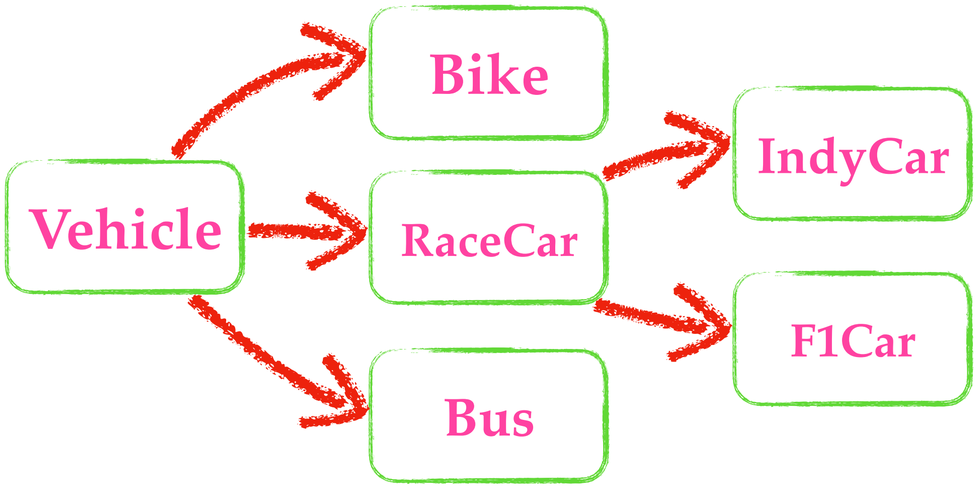
Herança (*inheritance*)

- Às vezes queremos criar uma classe que se comporte de várias maneiras como outro tipo de classe, mas com alguns comportamentos diferentes
- Uma maneira de fazer isso seria copiar toda a definição da classe original e fazer algumas modificações
- Porém, Python oferece uma forma mais simples e direta de realizar essa tarefa: **inheritance**

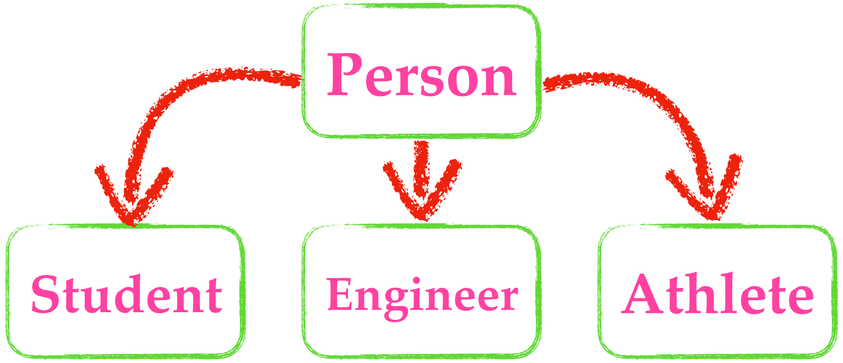
Sintaxe

- Para que uma nova classe herde de outra, colocaremos o nome da classe existente entre parênteses após o nome da nova classe na definição

Veículos



Grupos de Pessoas




```python
class Vehicle:
    speed = 10
    dangerous = False

    def __init__(self, brand, color):
        self.brand = brand
        self.color = color


class RaceCar(Vehicle):
    speed = 50
    dangerous = True


class F1Car(RaceCar):
    speed = 100
```

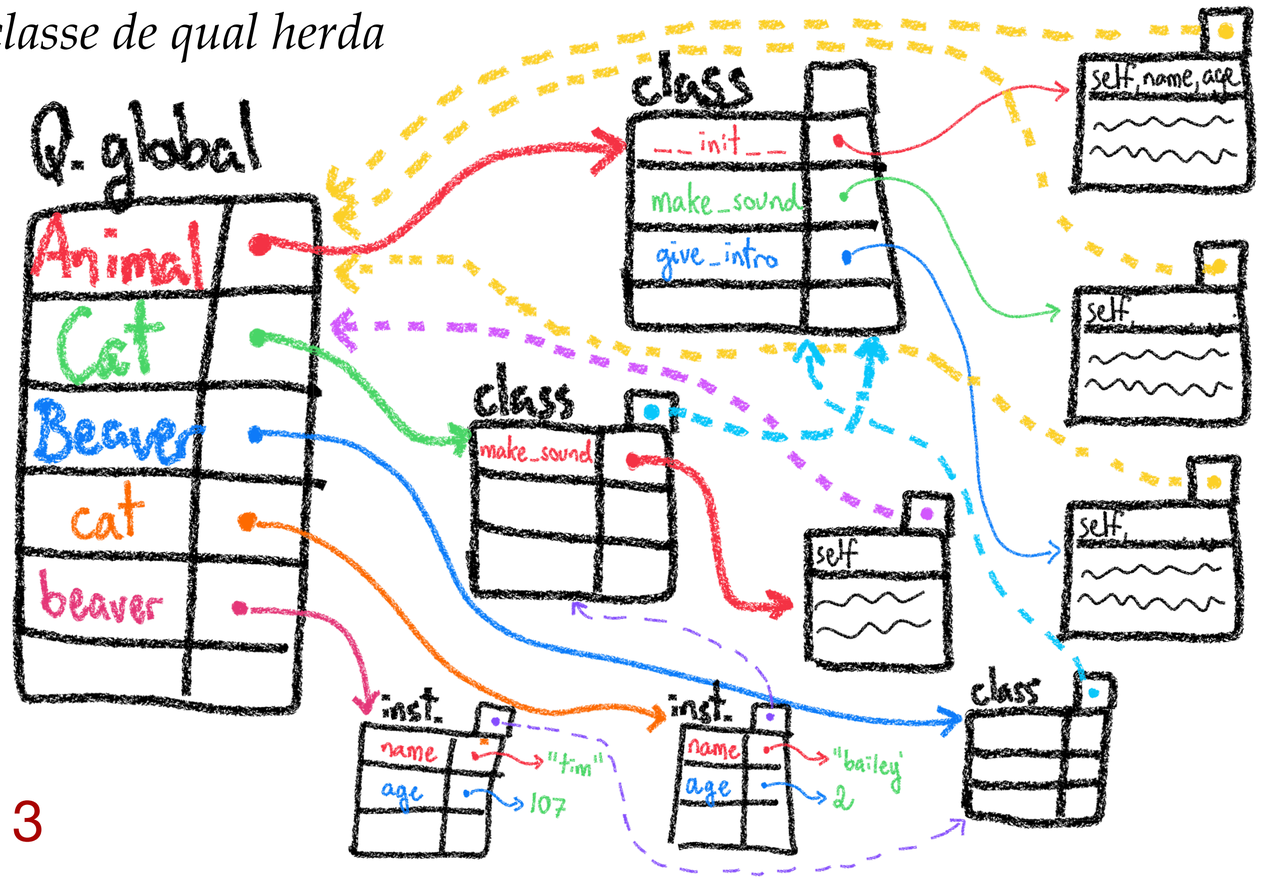

MIT BRAZIL

UNICAMP

Herança em diagramas

Diagramas de Ambiente

- O processo de definir uma classe herdeira se comporta da mesma maneira que qualquer outra, com uma sutil diferença
- *Essa nova classe terá um ponteiro parental para o objeto de classe de qual herda*




```python
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def make_sound(self):
        return "Vai saber que som esse bicho faz"

    def give_intro(self):
        return f"Este/a é {self.name}, de {self.age} anos"


class Cat(Animal):
    def make_sound(self):
        return "meow"


class Beaver(Animal):
    pass


cat = Cat("bailey", 2)
beaver = Beaver("tim", 107)

print(cat.make_sound())  # meow
print(cat.give_intro())  # Este/a é bailey, de 2 anos

print(beaver.make_sound())  # Vai saber que som esse bicho faz
print(beaver.give_intro())  # Este/a é tim, de 107 anos
```

Consequências de herança

- Por causa do ponteiro parental, quando procuramos um nome dentro da nova classe e não o encontramos, *continuamos procurando dentro da classe anterior*
  - A noção de herdar valores vem daí

Comportamento da classe nova vs. original

- Para métodos e atributos que não queremos modificar, isso permite que encontremos os objetos dentro da classe anterior
- Ao mesmo tempo, esse comportamento permite *sobrescrever* valores da classe original que desejamos alterar, já que Python procurará primeiro na nova classe

Quando definir na nova classe

- Devemos definir um atributo ou método na classe nova apenas quando seu comportamento for diferente daquele definido na classe original
- Para comportamentos repetidos, não escrevemos nada na nova classe para evitar repetição e deixamos o ponteiro parental realizar o trabalho

```python
class Question:
    def __init__(self, prompt, sol):
        self.prompt = prompt
        self.solution = sol

    def check(self, provided_answer):
        return provided_answer == self.solution

    def ask(self):
        print(self.prompt)
        response = input(">")
        if self.check(response):
            return "Yay! That's correct"
        else:
            return "Incorrect"


class CaseInsensitiveQuestion(Question):
    def check(self, provided_answer):
        return self.solution.lower() == provided_answer.lower()

```

Método **check** é sobrescrito na nova classe, mas **ask** e **__init__** são mantidos exatamente como são já que não foram definidos na nova classe

super()

- Não definir um método/atributo na nova classe faz com que o objeto da classe original seja usado; definir um método/atributo sobrescreve o atributo da classe original
- Mas e se quisermos usar um método da classe original como parte de um método da nova classe?
  - Poderíamos copiar/colar o código do método da classe original e estendê-lo com novas linhas
  - Poderíamos referenciar o método da classe original **cls_orig** usando seu nome: **cls_orig.method_name**
  - Mas e se mudarmos o nome da classe? Teríamos que trocar as referências um por um? Não parece ser bom estilo (não é genérico)

super()

- Podemos usar **super()** para nos referirmos à classe original da qual a classe atual herda para criar generalidade
- Além de generalidade, permite operações interessantes com modelos de *multiple inheritance* (classe que herda de mais de uma classe ao mesmo tempo)

```
1   class LoggingDict(dict):
2       def __setitem__(self, key, value):
3           # novo comportamento no nosso método especial
4           print(f"Associando chave {key} a {value}")
5           # agora realizamos função como faríamos
6           # em um dicionário integrado
7           super().__setitem__(key, value)
8
9
10  d = LoggingDict()
11  d[1] = 2  # imprime "Associando chave 1 a 2"
12  d[10] = True  # imprime "Associando chave 10 a True"
13
14  print(d)  # {1: 2, 10: True}
15  # ^ imprime mesmo que dict pois __str__ não foi mudada
16
17  print(type(d))  # LoggingDict
18
```

Se tivéssemos escrito
dict.__setitem__(key, value)
na linha 7 acima, teríamos problemas se decidíssemos trocar de **dict** para **OrderedDict** como classe original, por exemplo

Como *multiple inheritance* funciona:
https://www.programiz.com/python-programming/multiple-inheritance
Discussão sobre o poder de **super()**:
https://rhettinger.wordpress.com/2011/05/26/super-considered-super/

Other goodies (1/n)

Argumentos opcionais

Fonte muito perigosa (difícil de achar) de bugs! Não usar a menos que você saiba *muito* bem o que está fazendo

- Sintaxe tem dois componentes principais:
  - *Valor padrão* e *símbolo de atribuição* = conectando nome do parâmetro opcional e o valor padrão
- Se nenhum argumento para o parâmetro opcional **delta** for passado, então o valor padrão **1e-6** será adotado
- Se **delta** for especificado, assumirá o valor passado como argumento (*substituindo* o valor padrão)
- Se a função tiver parâmetros obrigatórios e opcionais, todos os obrigatórios precisam vir primeiro, seguidos pelos opcionais

Um aviso

- Valores padrão são definidos quando a função é *definida*, não chamada. Portanto, é perigoso criar valores padrão mutáveis, já que todas as chamadas usarão exatamente o mesmo objeto (que pode ter sido modificado por chamadas anteriores!)
  - Abordagem para contornar: colocar o valor padrão para **None** e usar uma condição **if** para substituir por objeto mutável dentro

```python
def isclose(x, y, delta=1e-6):
    return abs(x - y) <= delta


def append_x(x, lst=[]):
    lst.append(x)
    print(lst)


append_x(10)  # imprime [10]
append_x(20)  # imprime [10, 20]
append_x(30)  # imprime [10, 20, 30]


def append_x_fixed(x, lst=None):
    if lst is None:
        lst = []

    lst.append(x)
    print(lst)


append_x_fixed(10)  # imprime [10]
append_x_fixed(20)  # imprime [20]
append_x_fixed(30)  # imprime [30]
```

Other goodies (2/n)

Expressões condicionais

- Instruções condicionais são frequentemente utilizadas para escolher um de dois valores
- Python fornece uma forma mais concisa de expressar essa operação usando uma *expressão condicional*:
  - x = **valor_se_cond_verd if condição else valor_se_cond_falsa**

Comparações múltiplas em linha

- Anteriormente, usamos operadores de comparação (< <= == != >= >) como binários, mas na verdade são *n*-ários. Por exemplo:
  - x < y < z é equivalente a x < y **and** y < z
- Também funciona para combinações de comparadores diferentes:
  - x > y < z é equivalente a x > y **and** y < z

f-strings

- Recurso simples para incorporar valor de objetos dentro de strings (por exemplo, para imprimir valor de variáveis com texto)
- Sintaxe: **f"textotextotexto {nome_de_variável} textotextotexto"**
  - Equivalente a **"textotexto " + str(variável) + " textotexto"**

```python
def choose_highest(x, y):
    return x if x > y else y


def factorial(n):
    return 1 if n <= 1 else\
        n * factorial(n - 1)


print(10 < 20 < 30)  # True
print(10 > 20 < 30)  # False
print(10 <= 10 > 5)  # True

x, y, z = 10, 20, 30
print(f"x: {x}, y: {y}, z: {z}")
# x: 10, y: 20, z: 30

print(f"x / y = {x / y}")
# x / y = 0.5

```

Other goodies (3/n)

Capturando exceções

Podemos especificar *quais tipos* de exceção capturar no bloco **except** (escrever só **except** captura todos)

- É razoavelmente comum encontrar erros difíceis de prever quando escrevemos código
  - Por exemplo, um **ZeroDivisionError** se 0 for passado como denominador em uma divisão depende das entradas
- Em vez de *prever* erros, às vezes é melhor que nosso código aprenda a lidar com erros específicos por meio de instruções que devem ser especificamente executadas se um problema ocorrer
- Isso é exatamente o que **try/except** faz, com sintaxe semelhante a uma ramificação **if/else**
  - Python tenta executar o código no corpo de **try**. Se nenhum erro (*exceção*) ocorrer, o código simplesmente ignora o bloco **except**
  - Se ocorrer uma exceção, ele para de executar o bloco **try** e começa a executar o ramo de **except**
- A ideia de **try/except** é capturar a exceção, o que te dá a opção de corrigir o problema dentro do código, criar uma exceção de propósito em algum momento, imprimir uma mensagem de erro mais útil (personalizada), ou pelo menos encerrar o programa propriamente

```python
def divide(x, y):
    try:
        result = x / y
        return result
    except ZeroDivisionError:
        return None


def list_len(lst):
    length = 0
    try:
        while True:
            _ = lst[length]
            length += 1
    except IndexError:
        return length


print(list_len([1, 2, 3, 4]))  # 4
print(list_len([]))  # 0
```

Usar a forma genérica (sem especificar o tipo de exceção esperado) é má prática, pois pode esconder erros diferentes daqueles que esperamos, levando a problemas no futuro

Other goodies (4/n)

Argumentos nomeados

- Também podemos especificar parâmetros relacionados a cada um dos argumentos usando **kwarg=value**
- Argumentos não-nomeados (posicionais) precisam vir antes dos nomeados

*args e **kwargs

- ***args** e ****kwargs** são recursos que permitem que funções aceitem um número *arbitrário* de argumentos posicionais
- ***args** fornece todos os argumentos passados *via uma tupla*
- ****kwargs** fornece todos os argumentos nomeados *via um dicionário* exceto aqueles correspondendo a parâmetros formais (definidos)
- Ambos podem ser combinados com argumentos normais para permitir uma mistura de argumentos fixados e alguns variáveis

Operadores * e ** ao contrário

- Podemos também usar a sintaxe de * e ** ao contrário
- ***seq** é usado para *desempacotar sequência de argumentos quando chamando uma função*: **f(*[1, 2, 3])** é equivalente a **f(1, 2, 3)**
- ****dict** é usado para *desempacotar um dicionário de argumentos*: **f(**{"a": 1, "b": 3, "c": 5})** é equivalente a **f(a=1, b=3, c=5)**

```python
def foo(*args):
    print(f"Valor: {args}\n\
Tipo: {type(args)}\n")


def bar(**kwargs):
    print(f"Valor: {kwargs}\n\
Tipo: {type(kwargs)}\n")


def quux(sth, *args, **kwargs):
    print(f"Argumento normal: {sth}\n\
 Args. posicionais: {args}\n\
Args. nomeados: {kwargs}\n")


foo(100, -14, True, 2)
# Valor: (100, -14, True, 2)
# Tipo: <class 'tuple'>

bar(a=10, b="uhum", c=False)
# Valor: {"a": 10, "b": "uhum", "c": False}
# Tipo: <class 'dict'>

quux(12, 15, 17, 24, a=29, b=20, c=34)
# Argumento normal: 12
# Args. posicionais: (15, 17, 24)
# Args. nomeados: {'a': 29, 'b': 20, 'c': 34}

```

```python
def weird_math(x, y, z):
    print(x + y * z)


weird_math(1, 2, 3)  # 7
weird_math(*[1, 2, 3])  # 7 (eq)

weird_math(y=2, x=1, z=3)  # 7
weird_math(**{"y": 2,
              "x": 1,
              "z": 3})  # 7 (eq.)

```

Linha 36: operador * usado para desempacotar lista

Linha 39: operador ** usado para desempacotar dict

```python
first, *rest = [1, 2, 3, 4]
print(first)  # 1
print(rest)  # [2, 3, 4]

first, *mid, last = [1, 2, 3, 4]
print(first)  # 1
print(mid)  # [2, 3]
print(last)  # 4

```

Operador * também pode ser usado no lado esquerdo de atribuições, dando uma lista ao invés de uma tupla

https://stackoverflow.com/questions/36901/what-does-double-star-asterisk-and-star-asterisk-do-for-parameters

Recurso de todas os recursos relacionados a definir funções:

http://docs.python.org/dev/tutorial/controlflow.html#more-on-defining-functions