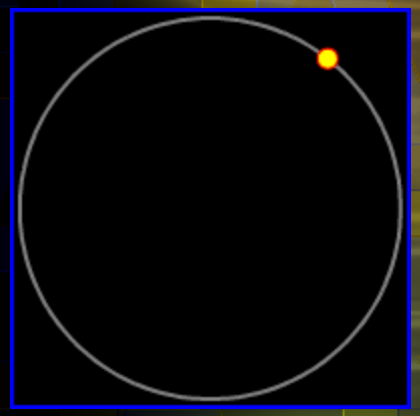
Tutorial

Adaptatividade em Python

Python

- Multiparadigma
- Uso geral (científico, GUI, Web, games, etc.)
- Metaprogramação e reflexão
  - “Python is more dynamic, does less error-checking” (P. Norvig, ao comparar Python com LISP)
- Em tempo de execução...
  - Código (arquivos/strings)
  - AST
  - Bytecode

E muito mais! =)



Parte 1

Geradores, iteradores e iteráveis

“Talk is cheap. Show me the code.”
(Linus Torvalds)

Avaliação tardia (deferred) e geradores

- Yield
  - Return-like
  - Estado “suspenso”
- Avaliação no uso, não na declaração
- “Fluxo de controle” em um objeto
- Iteráveis sem tamanho definido
  - I/O
  - “Algoritmo” sem término definido?


```
# Função geradora
def count(start=0, step=1):
    while True:
        yield start
        start += step

# Geradores são iteradores
gen = count()
next(gen)
next(gen)
next(gen)
gen_impares = count(1, step=2)
next(gen_impares)
next(gen_impares)
next(gen_impares)
next(gen)

# Expressão geradora
g = (el ** 2 for el in count(3)
                if el % 3 == 0)
next(g)
next(g)
next(g)

# Iteradores são iteráveis
for el in count(5, 3):
    print(el)
    if el >= 30:
        break
```

Isso tem pronto: itertools.count

```
In [1]: def count(start=0, step=1):
   ...:     while True:
   ...:         yield start
   ...:         start += step
   ...:

In [2]: gen = count()

In [3]: next(gen)
Out[3]: 0

In [4]: next(gen)
Out[4]: 1

In [5]: next(gen)
Out[5]: 2

In [6]: gen_impares = count(1, step=2)

In [7]: next(gen_impares)
Out[7]: 1

In [8]: next(gen_impares)
Out[8]: 3

In [9]: next(gen_impares)
Out[9]: 5

In [10]: next(gen)
Out[10]: 3
```

```
In [11]: g = (el ** 2 for el in count(3)
   ...:               if el % 3 == 0)
   ...:

In [12]: next(g)
Out[12]: 9

In [13]: next(g)
Out[13]: 36

In [14]: next(g)
Out[14]: 81

In [15]: for el in count(5, 3):
   ....:     print(el)
   ....:     if el >= 30:
   ....:         break
   ....:
5
8
11
14
17
20
23
26
29
32
```

Iterador/iterável com um pouco de orientação a objetos

```python
from collections import Iterator

class Counter(Iterator):

    def __init__(self, start=0, step=1):
        self.value = start
        self.step = step
        self.finish = False

    def next(self):
        if self.finish:
            raise StopIteration
        result = self.value
        self.value += self.step
        return result

    __next__ = next # Compatibilidade Python 2/3
```

- “Dunders” (Double UNDERscore)
  - __init__
    - Inicializador (“construtor”)
  - __next__ (next no Python 2)
    - Devolve o próximo elemento
  - __iter__
    - Iterável: Devolve um novo iterador
    - Iterador: Devolve a si próprio

```python
g = Counter()
next(g)
next(g)
g.value = 13
next(g)
next(g)
g.step = 7
next(g)
next(g)
next(g)
```

Mudança de comportamento do iterável dentro de seu laço

```python
counter = Counter(start=-5, step=7)
for el in counter:
    print(el)
    counter.step -= 1
    counter.finish = counter.value < -10
```

```
In [1]: %paste
from collections import Iterator
# [...]
## -- End pasted text --

In [2]: g = Counter()

In [3]: next(g)
Out[3]: 0

In [4]: next(g)
Out[4]: 1

In [5]: g.value = 13

In [6]: next(g)
Out[6]: 13

In [7]: next(g)
Out[7]: 14

In [8]: g.step = 7

In [9]: next(g)
Out[9]: 15

In [10]: next(g)
Out[10]: 22

In [11]: next(g)
Out[11]: 29
```

Classe “Counter”

```
In [12]: counter = Counter(start=-5, step=7)

In [13]: for el in counter:
    ....:     print(el)
    ....:     counter.step -= 1
    ....:     counter.finish = counter.value < -10
    ....:
-5
2
8
13
17
20
22
23
23
22
20
17
13
8
2
-5
```

```
# [...] Lembram do Counter.next?
  def next(self):
      if self.finish:
          raise StopIteration
      result = self.value
      self.value += self.step
      return result
```

Lazy evaluation
Avaliação “preguiçosa”

- “Tardia” + “cache”:
  - “Por que fazer antes o que pode ser deixado para a última hora?”
  - +
  - Valores computados uma única vez
- Uso único?
  - Único caso equivalente à avaliação tardia (“deferred”)
- Iteradores/Geradores: cópias (T)
  - itertools.tee
  - audiolazy.thub
  - Fluxo de dados e “Dataflow programming”
- Funções (+ imutabilidade): decorators (cache)
  - functools.lru_cache (Python 3)
  - audiolazy.cached

```python
# coding: utf-8
from itertools import count, takewhile, tee

def prime_gen():
    """ Gerador de números primos """
    yield 2
    primes = []
    for value in count(start=3, step=2):
        iter_primes = takewhile(lambda x: x * x <= value, primes)
        if all(value % p != 0 for p in iter_primes):
            primes.append(value)
            yield value

primes, primes_copy = tee(prime_gen(), 2)

for idx, p in enumerate(primes, 1):
    print(u"{:>5}º primo: {}".format(idx, p))
    if idx == 200:
        break

for idx, p in enumerate(primes_copy, 1):
    print(u"{:>5}º primo ao quadrado: {}".format(idx, p ** 2))
    if idx == 200:
        break
```

```
    1º primo: 2
    2º primo: 3
    3º primo: 5
    4º primo: 7
    5º primo: 11
[...]
  198º primo: 1213
  199º primo: 1217
  200º primo: 1223
```

```
    1º primo ao quadrado: 4
    2º primo ao quadrado: 9
    3º primo ao quadrado: 25
    4º primo ao quadrado: 49
    5º primo ao quadrado: 121
[...]
  198º primo ao quadrado: 1471369
  199º primo ao quadrado: 1481089
  200º primo ao quadrado: 1495729
```

Baseado em:
https://gist.github.com/danilobellini/7233352

```python
# coding: utf-8
from audiolazy import count, takewhile, thub

def prime_gen():
    """ Gerador de números primos """
    yield 2
    primes = []
    for value in count(start=3, step=2):
        stream_primes = takewhile(lambda x: x * x <= value, primes)
        if all(value % stream_primes != 0):
            primes.append(value)
            yield value

primes = thub(prime_gen(), 2)

for idx, p in enumerate(primes, 1):
    print(u"{:>5}º primo: {}".format(idx, p))
    if idx == 200:
        break

for idx, p in enumerate(primes, 1):
    print(u"{:>5}º primo ao quadrado: {}".format(idx, p ** 2))
    if idx == 200:
        break
```

```
    1º primo: 2
    2º primo: 3
    3º primo: 5
    4º primo: 7
    5º primo: 11
[...]
  198º primo: 1213
  199º primo: 1217
  200º primo: 1223
```

```
    1º primo ao quadrado: 4
    2º primo ao quadrado: 9
    3º primo ao quadrado: 25
    4º primo ao quadrado: 49
    5º primo ao quadrado: 121
[...]
  198º primo ao quadrado: 1471369
  199º primo ao quadrado: 1481089
  200º primo ao quadrado: 1495729
```

Mesma coisa, mas com audiolazy.thub no lugar de itertools.tee, e takewhile devolvendo audiolazy.Stream

Outros exemplos

- Geradores para I/O
  - AudioLazy: mcfm.py, robotize.py, animated_plot.py
  - Turing(1936): a-machine VS c-machine
- Avaliação lazy/preguiçosa por decorator

Algoritmo para cada valor/bloco de “entrada” (saída em tempo finito); causalidade

```
import sys # Para funcionar em Python 2 e 3
if sys.version_info.major == 2:
    from cachetools import lru_cache
else:
    from functools import lru_cache

@lru_cache(maxsize=1000)
def fib(n):
    return n if n <= 1 else fib(n - 2) + fib(n - 1)

print(fib(500))
```

```
139423224561697880139724382870407283950070256587697307264108962948325571622863290691557658876222521294125
```

Parte 2

“Primeira classe” e closures

Design patterns? (design strategies?)

- Slides “Design Patterns in Dynamic Programming” (P. Norvig)
  - “16 of 23 patterns have qualitatively simpler implementation in Lisp or Dylan than in C++ for at least some uses of each pattern”
- Defaults != patterns != standards != defaults
  - http://info.abril.com.br/noticias/rede/gestao20/software/a-lingua-portuguesa-brasileira-e-pessima-standard-vs-pattern/
- 2013-2014: Grupo de estudos para discutir aplicabilidade em linguagens dinâmicas:
  - https://garoa.net.br/wiki/Design_patterns_em_linguagens_din%C3%A2micas
  - Simplificados com funções/“tipos” de primeira classe:
    - Command, strategy, template method, visitor, abstract factory, factory method, flyweight, proxy, chain of responsibility, state
- Exemplos (código) em várias linguagens
  - http://en.wikibooks.org/wiki/Computer_Science_Design_Patterns

Função de primeira classe

- Funções como valores/objetos
- Exemplo: pattern strategy para operador binário com o símbolo como parâmetro sem usar “classes”
  - Baseado em (o link possui uma versão em C e várias em Python): https://github.com/danilobellini/design_patterns

```
def add(x, y): return x + y
def sub(x, y): return x - y
def mod(x, y): return x % y

ops = {
    "+": add,
    "-": sub,
    "%": mod,
}

def apply_op(symbol, x, y):
    return ops[symbol](x, y)

print(apply_op("+", 2, 3))   # 5
print(apply_op("-", 5, 7))   # -2
print(apply_op("%", 22, 13)) # 9
```

Alternativas

```
ops = {
    "+": lambda x, y: x + y,
    "-": lambda x, y: x - y,
    "%": lambda x, y: x % y,
}
```

```
template = "lambda x, y: x %s y"
ops = {s: eval(template % s) for s in "+-*/%"}
```

[Lexical] Closure

- “Função” + “contexto”

  Blocos “instanciáveis” (e.g. funções anônimas)

  + (com)

  “Variáveis livres” (definidas para cada “instância”)

- Caso típico: aplicação parcial

```
def adder(a):
    return lambda b: b + a

add2 = adder(2)
sub1 = adder(-1)

print(add2(15))       # 17
print(sub1(4))        # 3
print(add2(sub1(8))) # 9
```

“b” é o único parâmetro da função anônima devolvida, “a” é uma variável livre

Quantidades e nomes de parâmetros em funções/métodos

- def/lambda
  - Valores default com “=”
    - Pertencem à função/método
    - Mutabilidade → influencia em todas as chamadas
  - Argumentos extras
    - Posicionais
      - * unário → tupla
    - Nominados (keyword)
      - ** unário → dicionário

- Chamada (todo item é opcional)
  1. Argumentos posicionais
  2. (1x) * unário, continuação dos argumentos posicionais com qualquer iterável
  3. Pares “chave=valor” explícitos
  4. (1x) ** unário, utiliza todos os pares “chave=valor” de um dicionário
- * e ** unários da chamada não são os mesmos recebidos pela função como argumentos
  - Verificação dos nomes, quantidades, excessos, ausências e repetições (TypeError)

```
def smaller_first(pair):
    k, v = pair
    return len(k), k
def show_it_all(a, b, c=None, *args, **kwargs):
    for k, v in sorted(locals().items(), key=smaller_first):
        print("{:>6}: {}".format(k, v))
```

```
squares = [i ** 2 for i in range(10)]
ascii = {ch: hex(ord(ch)) for ch in "Place"}
```

```
>>> show_it_all(0, -1)
   a: 0
   b: -1
   c: None
args: ()
kwargs: {}
```

```
>>> show_it_all(b=0, a=-1)
   a: -1
   b: 0
   c: None
args: ()
kwargs: {}
```

```
>>> show_it_all(*squares)
   a: 0
   b: 1
   c: 4
args: (9, 16, 25, 36, 49, 64, 81)
kwargs: {}
```

```
>>> show_it_all(*"First")
   a: F
   b: i
   c: r
args: ('s', 't')
kwargs: {}
```

```
>>> show_it_all(1, 2, 3, 4, 5, d=415)
   a: 1
   b: 2
   c: 3
args: (4, 5)
kwargs: {'d': 415}
```

```
>>> show_it_all(-5, *squares, args=415)
   a: -5
   b: 0
   c: 1
args: (4, 9, 16, 25, 36, 49, 64, 81)
kwargs: {'args': 415}
```

```
>>> show_it_all(b="Second", **ascii)
   a: 0x61
   b: Second
   c: 0x63
args: ()
kwargs: {'P': '0x50', 'e': '0x65', 'l': '0x6c'}
```

```
# TypeError
show_it_all("First", **ascii)   # "a" 2x
show_it_all(*squares, **ascii)  # "a" 2x
show_it_all(1, 2, 3, 4, 5, b=5) # "b" 2x
show_it_all(b=5, e=0, **ascii)  # "e" 2x
show_it_all(**ascii) # "b" ausente
show_it_all(0)       # "b" ausente
show_it_all()        # "a" ausente
```

```
def smaller_first(pair):
    k, v = pair
    return len(k), k
def show_it_all(a, b, c=None, *args, **kwargs):
    for k, v in sorted(locals().items(), key=smaller_first):
        print("{:>6}: {}".format(k, v))
```

Quanta coisa...

Decorator

```python
# coding: utf-8
from functools import wraps, reduce

def double_of(func):
    """Decorator que dobra o resultado da função."""
    @wraps(func) # Copia informações de func
    def wrapper(*args, **kwargs):
        return 2 * func(*args, **kwargs)
    return wrapper

sum_twice = double_of(sum)
print(sum_twice([2, 5, 3])) # 20

@double_of
def prod_twice(data):
    return reduce(lambda x, y: x * y, data, 1)

print(prod_twice([2, 5, 3])) # 60

# A menos dos nomes, o acima é o mesmo que:
def prod(data):
    return reduce(lambda x, y: x * y, data, 1)
prod2x = double_of(prod) # Decorator!

print(prod2x([2, 5, 3])) # 60
```

- Função
- 1 parâmetro
  - Função ou classe
- 1 valor de saída
  - Normalmente função ou classe
- Uso com o @
  - Usa o nome do próprio objeto “decorado”

Métodos com “self” explícito?

- Currying!
  - Aplicação parcial do primeiro parâmetro

```
class MsgKeeper(object):
    def __init__(self, msg):
        self.msg = msg
    def show(self):
        print(self.msg)

foo = MsgKeeper("foo")
bar = MsgKeeper("bar")

foo.show()
bar.show()
MsgKeeper.show(foo)
MsgKeeper.show(bar)
```

- Aninhamento
  - “self” é um nome arbitrário
- Classes de primeira classe (+ closure)
  - Exemplos:
    - audiolazy.StrategyDict
      - Docstring dinâmica e por instância
    - dose (tratamento de eventos do watchdog)

Classe de primeira classe e Monkeypatch

```python
class MsgKeeper(object):
    def __init__(self, msg):
        self.msg = msg
    def show(self):
        print(self.msg)

# Definido fora do namespace da classe
def prefixed_keeper(self):

    class PrefixedKeeper(MsgKeeper):

        @property
        def msg(this):
            return self.msg + this._msg

        @msg.setter
        def msg(this, value):
            this._msg = value

    return PrefixedKeeper

turn = MsgKeeper("turn")
```

O método prefixed_keeper não existia quando turn foi instanciado!

Monkeypatch: atualização da classe em tempo de execução

```python
# Monkeypatch
MsgKeeper.prefixed_keeper = prefixed_keeper
# (poderia ser definido dentro da classe)

turno = turn.prefixed_keeper()("o")
turnon = turno.prefixed_keeper()("n")

turn.show()   # turn
turno.show()  # turno
turnon.show() # turnon
turn.msg = "Pyth"
turnon.show() # Python
```

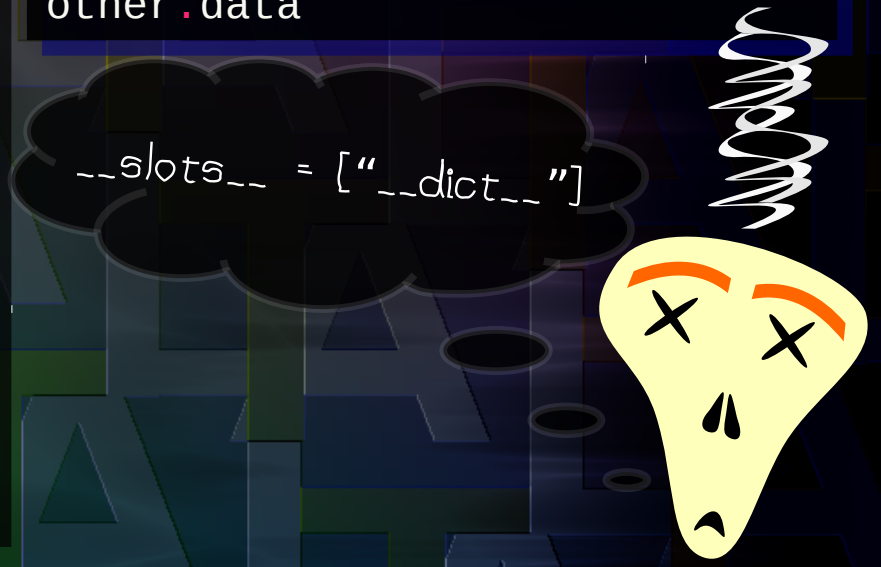
Parte 3

Namespaces, dicionários e escopo (léxico)

Dicionário da classe e dicionário da instância

- Tudo em Python é um objeto
- Namespaces são representados como dicionários
- Objetos e classes [normalmente] possuem um namespace de atributos
  - “vars(obj)”
    - Dicionário (__dict__)
    - Nem sempre é aplicável
      - Tipos built-ins, __slots__
  - “dir(obj)”
    - Lista de nomes (strings) dos atributos, incluindo os da classe/herança (exceto os da metaclasse)
    - Sempre aplicável

```
class A(object):
    data = "Testing"
    def __init__(self, data):
        self.data = data

other = A("Other")

# Digitar no IPython:
A.data
other.data
type(other) # other.__class__
type(other).data
vars(other)
vars(A)

# E se apagarmos da instância?
del other.data
vars(other)
other.data
```



Resolução dinâmica de atributos

```
In [1]: class A(object):
   ...:     data = "Testing"
   ...:     def __init__(self, data):
   ...:         self.data = data
   ...:

In [2]: other = A("Other")

In [3]: A.data
Out[3]: 'Testing'

In [4]: other.data
Out[4]: 'Other'

In [5]: type(other)
Out[5]: __main__.A

In [6]: type(other).data
Out[6]: 'Testing'

In [7]: vars(other)
Out[7]: {'data': 'Other'}

In [8]: vars(A)
Out[8]:
<dictproxy {'__dict__': <attribute '__dict__' of 'A' objects>,
 '__doc__': None,
 '__init__': <function __main__.__init__>,
 '__module__': '__main__',
 '__weakref__': <attribute '__weakref__' of 'A' objects>,
 'data': 'Testing'}>
```


```
In [9]: del other.data

In [10]: vars(other)
Out[10]: {}

In [11]: other.data
Out[11]: 'Testing'
```

Docstrings, help

```
Dicas para uso do IPython

TAB          -> Code completion
? ao final   -> Docstring + extras
?? ao final  -> Código-fonte
```

- Strings são imutáveis
- Primeira string (literal) do bloco
  - Módulo
  - Classe
  - Método/Função
- Built-in “help()”
- Dunder __doc__
  - Referência fixa em classes
    - Dinâmico se __doc__ for uma property (e.g. audiolazy.StrategyDict)
  - Referência modificável em módulos e métodos/funções (uso intenso na AudioLazy)
- Documentação no próprio código

```python
# coding: utf-8
def is_palindrome(val):
    """
    Verifica se a representação do valor
    fornecido é um palíndromo
    """
    as_string = repr(val)
    return as_string == as_string[::-1]

# Rodar em um REPL
is_palindrome(123213) # False
is_palindrome(123321) # True

help(is_palindrome) # Exibe a docstring e
                    # outras informações

is_palindrome.__doc__ # Devolve a docstring
#\n
#----Verifica se a representação do valor\n
#----fornecido é um palíndromo\n
#----
```

Locals & globals

- 2 namespaces por contexto (escopo léxico): “local” e “global”
- Funções built-in
  - locals()
  - globals()
  - dir()
    - Sem parâmetro fornece os nomes (chaves) de “locals()”
- Modificar o resultado de “locals()” apenas possui o efeito de mudança do namespace no nível do módulo/script (ou direto no REPL)
  - Manipulação do namespace a partir de strings e valores ao invés de nomes em código
    - Módulos audiolazy.lazy_math e audiolazy.lazy_itertools
  - Criação massiva / automação
  - Uso de dados ao invés de estrutura

```
from math import factorial

for i in range(35):
    locals()["f%d" % i] = factorial(i)

print(f0) # 1
print(f1) # 1
print(f5) # 120
print(f15) # 1307674368000

print(dir()) # Como script, Python 2:
#['__builtins__', '__doc__',
# '__file__', '__name__',
# '__package__', 'f0', 'f1', 'f10',
# 'f11', 'f12', 'f13', 'f14', 'f15',
# 'f16', 'f17', 'f18', 'f19', 'f2',
# 'f20', 'f21', 'f22', 'f23', 'f24',
# 'f25', 'f26', 'f27', 'f28', 'f29',
# 'f3', 'f30', 'f31', 'f32', 'f33',
# 'f34', 'f4', 'f5', 'f6', 'f7',
# 'f8', 'f9', 'factorial', 'i']
```

“Escopo dinâmico”?

```
# coding: utf-8
template = u"{level} ({author}): {msg}"

print(template.format(
    msg = u"Isto não é uma mensagem.",
    level = u"Info",
    author = u"Danilo",
)) # Exibe:
# Info (Danilo): Isto não é uma mensagem.

msg = u"Contexto como parâmetro?"
level = u"Dinâmico"
author = u"Locals!"

print(template.format(**locals())) # Exibe:
# Dinâmico (Locals!): Contexto como parâmetro?
```

- É possível a passagem do resultado de locals() como parâmetro
  - No exemplo, recebe como “**kwargs”, não como valores despejados no locals() externo.
  - Valores mutáveis (e.g. listas) podem ser modificados.

Documentação automática

- audiolazy.StrategyDict
  - Dicionário de estratégias
  - Múltiplas implementações de “coisas similares”
  - Iterável pelas estratégias
  - Mutável
  - Callable (estratégia default)
  - Estratégias acessíveis como atributos e como itens
  - Docstring dinâmica (resumo das docstrings das estratégias)
- audiolazy.format_docstring
  - Decorator p/ fazer docstrings com a partir de template

- Sphinx (reflexão)
  - Geração de documentação em HTML, LaTeX, PDF, man pages, etc.
  - conf.py + reStructuredText + docstrings (em reStructuredText)
  - Equacionamentos matemáticos em LaTeX
  - Integração com Matplotlib

audiolazy.format_docstring

```python
# coding: utf-8
# Código original na AudioLazy (exceto pela docstring):
def format_docstring(template_="{__doc__}", *args, **kwargs):
    def decorator(func):
        if func.__doc__:
            kwargs["__doc__"] = func.__doc__.format(*args, **kwargs)
        func.__doc__ = template_.format(*args, **kwargs)
        return func
    return decorator

# Exemplo
def adder(a):
    @format_docstring(a=a)
    def add(b):
        """Efetua a soma {a} + b para o dado b."""
        return a + b
    return add

add3 = adder(3)
sub2 = adder(-2)
help(add3) # Efetua a soma 3 + b para o dado b.
help(sub2) # Efetua a soma -2 + b para o dado b.
```

Parte 4

Fallback e reflexão (reflection) para itens e atributos

Dunder __missing__

- “Fallback” para tentativa de acessar item inexistente no dicionário
- Necessita de uma nova classe (herança)

```
# coding: utf-8
from __future__ import unicode_literals
# Vamos remover acentos!
class DictDefaultsToKey(dict):
    def __missing__(self, key):
        return key

rev_ascii = {
  "a": "áàãâāäă", "A": "ÁÀÃÂĀÄĂ",
  "e": "éèẽêēëĕ", "E": "ÉÈẼÊĒËĔ",
  "i": "íìĩîīïĭ", "I": "ÍÌĨÎĪÏĬ",
  "o": "óòõôōöŏ", "O": "ÓÒÕÔŌÖŎ",
  "u": "úùũûūüŭ", "U": "ÚÙŨÛŪÜŬ",
  "c": "ç",       "C": "Ç",
  "n": "ñ",       "N": "Ñ",
}

ascii_dict = DictDefaultsToKey()
for k, values in rev_ascii.items():
    ascii_dict.update((v, k) for v in values)

def to_ascii(msg):
    return "".join(ascii_dict[ch] for ch in msg)

# Exemplos
to_ascii("La cédille: ç/Ç (Forçação de barra)")
to_ascii("Qui a décidé d'être naïf?")
to_ascii("ñÑãÃsáÁüúù...")
```

__getitem__, __call__

```python
# -*- coding: utf-8 -*-
from functools import reduce
from operator import add, mul

class SumProd(object):
    def __init__(self):
        self.count = 0

    def __getitem__(self, key): # 1 argumento!
        """Somatório (não-vazio)"""
        self.count += 1
        return reduce(add, key)

    def __call__(self, *key):
        """Produtório (não-vazio)"""
        self.count += 1
        return reduce(mul, key)

sp = SumProd()
print(  sp(3, 4, 5)  ) # __call__    -> 60
print(  sp[3, 4, 5]  ) # __getitem__ -> 12

print(sp.count) # 2
```

- __getitem__
  - “Operador” []
  - 1 único parâmetro (chave), que pode ser uma tupla
- __call__
  - “Operador” ()
  - Permite tratar objetos quaisquer como funções

Dunders get/set/delete para itens e atributos do objeto

- Item
  - __getitem__ : obj[key]
  - __setitem__ : obj[key] = value
  - __delitem__ : del obj[key]
- Atributo
  - __setattr__ : obj.key = value
  - __delattr__ : del obj.key
- Leitura de atributo
  - __getattr__ : obj.key
    - Chamado quando “key not in dir(obj)”
  - __getattribute__ : obj.key
    - [Quase] incondicional
    - Não é chamado se o “key” for um dunder chamado internamente por um built-in

https://docs.python.org/3/reference/datamodel.html

Conteúdo que TODO pythonista deveria conhecer!

Inclui os dunders dos operadores, de descriptors, chamadas por built-ins, context managers, etc.

```python
# coding: utf-8
from random import choice, randint
from string import ascii_lowercase

def mastermind(guess, secret):
    """
    Compara as strings de entrada e devolve um par de inteiros
    (caracteres corretos na posição correta,
     caracteres corretos se desconsiderarmos a posição)

    Origem: https://gist.github.com/danilobellini/5311427
    """
    return sum(1 for g, s in zip(guess, secret) if g == s), \
           sum(min(guess.count(x), secret.count(x)) for x in set(secret))

class NameMastermind(object):
    def __init__(self):
        size = randint(3, 8)
        name = "".join(choice(ascii_lowercase) for el in xrange(size))
        self._name = name
        setattr(self, name, lambda: "Yeah!")
    def __getattr__(self, name):
        return lambda: mastermind(name, self._name)
```

```python
game = NameMastermind()
# Para rodar no REPL...
# NÃO APERTE TAB! Exemplo:
#game.abcd() # -> (0, 0)
#game.efgh() # -> (1, 2)
#game.eeff() # -> (0, 0)
#game.ijkh() # -> (1, 1)
#game.lmno() # -> (0, 1)
#game.lmpq() # -> (0, 1)
#game.lmgq() # -> (0, 2)
#game.lmqg() # -> (0, 2)
#game.rstu() # -> (0, 0)
#game.vwxy() # -> (0, 1)
#game.lgzh() # -> (1, 3)
#game.gmvh() # -> (2, 2)
#game.glwh() # -> 'Yeah!'
```

Funções built-in
hasattr, getattr, setattr

- hasattr
  - Devolve um booleano indicando se o atributo existe
  - Pode chamar __getattr__
    - E se __getattr__ tiver efeito colateral?
- getattr e setattr
  - Particularmente útil para acessar atributos por strings dos nomes ou iterar em módulos

- Módulos também são objetos!
  - Pode-se usar hasattr, getattr, setattr

```
import itertools
if hasattr(itertools, "accumulate"):
    print("Python 3") # A rigor, 3.2+
else:
    print("Python 2") # Talvez 3.0 / 3.1
```

Parte 5

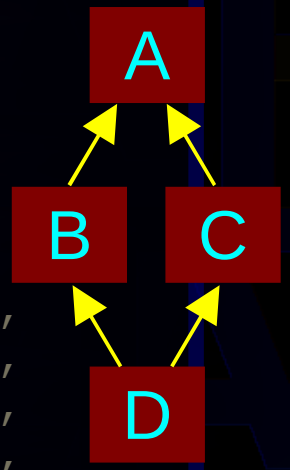
MRO, herança múltipla, __new__, metaclasses

MRO
Method Resolution Order

- Python tem herança múltipla
  - Linearizarização C3
    - Grafo de herança → MRO

```
class A(object): pass
class B(A): pass
class C(A): pass
class D(B, C): pass

classes = [A, B, C, D]
instances = [cls() for cls in classes]

def msg_returner(msg):
    return lambda self: msg

for cls in classes:
    cls.__str__ = msg_returner(cls.__name__)

def show_all():
    print("{} {} {} {}".format(*instances))
```

```
show_all() # A B C D
del C.__str__
show_all() # A B A D
C.__str__ = msg_returner("3")
show_all() # A B 3 D
del D.__str__
show_all() # A B 3 B
del B.__str__
show_all() # A A 3 3
del C.__str__
show_all() # A A A A

print(D.mro())
# [<class '__main__.D'>,
#  <class '__main__.B'>,
#  <class '__main__.C'>,
#  <class '__main__.A'>,
#  <type 'object'>]
```



type, isinstance, issubclass

- type(name, bases, namespace)
  - Devolve tipos/classes

```
A = type("A", (object,), {"__str__": lambda self: "A"})
B = type("B", (A,),      {"__str__": lambda self: "B"})
C = type("C", (A,),      {"__str__": lambda self: "C"})
D = type("D", (B, C),    {"__str__": lambda self: "D"})
# Já com os métodos __str__ no namespace! [...]
```

- isinstance(obj, cls)
  - cls in type(obj).mro()
  - cls pode ser uma tupla
    - any(c in type(obj).mro() for c in cls)
- issubclass(cls1, cls2)
  - cls2 in cls1.mro()
  - cls2 pode ser uma tupla
    - any(c in cls1.mro() for c in cls2)

Lembrando que o 1º elemento da MRO é a própria classe

```
a, b, c, d = A(), B(), C(), D()

print(isinstance(d, A))       # True
print(isinstance(a, (B, C))) # False
print(isinstance(c, (B, C))) # True

print(issubclass(A, B))       # False
print(issubclass(B, A))       # True
print(issubclass(B, (C, D))) # False
```

Construtor __new__ e decorator @classmethod

```python
class MsgKeeper(object):
    def __new__(cls, msg):
        instance = super(MsgKeeper, cls).__new__(cls)
        instance.msg = msg
        return instance if msg != "Certa!" else -1

    @classmethod
    def alt_new(cls, msg):
        return cls(msg[::-1])

obj = MsgKeeper("Minha mensagem!")
print(obj)     # <__main__.MsgKeeper object at 0x...>
print(obj.msg) # Minha mensagem!

print(MsgKeeper("Certa!")) # -1

obj_alt = MsgKeeper.alt_new("Certa!")
print(obj_alt.msg) # !atreC
```

super(MsgKeeper, cls) é [um proxy de] object

- Primeiro elemento é a classe, não a instância “self”
  - __new__(cls, …)
  - O mesmo efeito pode ser obtido com o decorator @classmethod
- Precisa devolver a instância
  - Que sequer precisa ter a ver com “cls”...

Metaclasses

- A classe da classe
  - Normalmente “type” é a classe de todas as classes
    - “type” está para as metaclasses assim como “object” está para as classes
  - Usar novos nomes pode ajudar: “cls” e “mcls” no lugar de “self” é “cls”
- Duas sintaxes
  - Python 2
    - __metaclass__ no namespace
  - Python 3
    - argumento nominado “metaclass=” após as bases da herança
  - Problema sério para compatibilidade em código único
    - six → Cria um nível de herança “dummy” para manter sintaxe única
    - AudioLazy → Função “meta” no lugar das bases da herança permite usar a sintaxe similar à do Python 3 compatível com ambos
- Classes criadas por herança TAMBÉM compartilham a metaclasse

Metaclasse que apenas avisa quando instancia a classe

```python
import sys

class Metaclass(type):
    def __init__(cls, name, bases, namespace):
        print("Initializing class {}\n"
              "bases: {}\n"
              "namespace: {}"
              .format(name, bases, namespace))

if sys.version_info.major == 2: # Python 2
    exec("""class M1(object): __metaclass__ = Metaclass""")
else: # Python 3
    exec("""class M1(object, metaclass=Metaclass): pass""")

# Initializing class M1
# bases: (<class 'object'>,)
# namespace: {'__module__': '__main__', …}

# Similar: M1 = Metaclass("M1", (object,), {})
# (mas o namespace resultante nem '__module__' possui)
```

AbstractOperatorOverloaderMeta

```python
from audiolazy import AbstractOperatorOverloaderMeta, meta

class PairMetaClass(AbstractOperatorOverloaderMeta):
    __without__ = "r" # Sem ops reversos __rbinary__

    def __binary__(cls, opmeth):
        return lambda self, other: \
            cls(opmeth.func(self.x, other.x),
                opmeth.func(self.y, other.y))

    def __unary__(cls, opmeth):
        return lambda self: \
            cls(opmeth.func(self.x),
                opmeth.func(self.y))

class Pair(meta(object, metaclass=PairMetaClass)):
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __str__(self):
        return "({}, {})".format(self.x, self.y)

print(Pair(4, 3) - Pair(7, 12))
print(Pair(41, 5) + Pair(18, 3))
print(Pair("a", "bc") + Pair("de", "f"))
print(Pair([1, 2], "abc") * Pair(2, 3))
```

```python
# Exemplo de vars(opmeth):
{'arity': 2,
 'symbol': '+',
 'func': <function
          _operator.add>,
 'name': 'add',
 'dname': '__add__',
 'rev': False}
```

Sobrecarga massiva de operadores:
21/33 (Python 3)
ou
22/35 (Python 2)
métodos.

```python
# Sintaxe Python 2
class Pair(object):
    __metaclass__ = PairMetaClass
    # [...]
```

```python
# Sintaxe Python 3
class Pair(object, metaclass=PairMetaClass):
    # [...]
```

```
# Resultado:
#
#(-3, -9)
#(59, 8)
#(ade, bcf)
#([1, 2, 1, 2], abcabcabc)
```

Parte 6

Importação: arquivos, módulos e pacotes

Módulos e sys.modules

o módulo “sys” representa/controla o interpretador

- Cada arquivo “*.py” é mapeado em um módulo
- O módulo “chamado” que inicia o interpretador é o script
  - O único que contém __name__ igual a “__main__”
- Todos os módulos importados estão em um dicionário sys.modules, que funciona como um “cache”
  - São importados somente uma vez
    - Isso permite importação circular (exceto com dependência direto no nível do módulo)
  - Podemos manipular sys.modules? **SIM!!!**

Criando módulos dinamicamente: Testes da AudioLazy

```python
# Adaptado da AudioLazy, audiolazy/test/__init__.py
from importlib import import_module, sys
import types, pytest
from _pytest.skipping import XFailed

class XFailerModule(types.ModuleType):
    def __init__(self, name):
        try:
            if isinstance(import_module(name.split(".", 1)[0]),
                          XFailerModule):
                raise ImportError
            import_module(name)
        except (ImportError, XFailed):
            sys.modules[name] = self
            self.__name__ = name

    __file__ = __path__ = __loader__ = ""

    def __getattr__(self, name):
        def xfailer(*args, **kwargs):
            pytest.xfail(reason="Module {} not found"
                                .format(self.__name__))
        return xfailer
```

XFail significa "eXpected to Fail", uma forma de "pular" testes

Permite testar mesmo na indisponibilidade de algum módulo (evita que o ImportError impossibilite testes que não dependam da importação)

__import__ e importlib.import_module

- É possível importar a partir de nomes em strings
  - Grosso modo,

    ```
    import nomedopacote
    ```

    é o mesmo que

    ```
    nomedopacote = __import__("nomedopacote")
    ```

  - __import__ permite especificar um namespace “globals” para importação
  - importlib.import_module e __import__ têm sintaxes diferentes do statement import para importações aninhadas e relativas

```
print(__import__("sys").version_info)
# sys.version_info(major=2, minor=7, micro=8, releaselevel='final', serial=0)
# ou
# sys.version_info(major=3, minor=4, micro=2, releaselevel='final', serial=0)
```

Sobre a importação

```
import os, sys

# Primeiro diretório de sys.path é o
# "." de quando o script foi chamado.

os.chdir("..") # Isto é irrelevante
for name in sys.path:
    print(name)

#/home/danilo/Desktop/WTA/code
#[...]
```

- sys.path
  - Ordem de busca por módulos
  - Lista (editável) de strings
    - e.g. sms-tools usado no curso ASPMA (Coursera) tinha manipulação de sys.path nos scripts dos “assignments”: `sys.path.append('../../software/models/')`
  - “python setup.py develop” → atualiza o sys.path “permanentemente”
    - “python setup.py develop -u” é uninstall...
- A importação pode ocorrer em qualquer instante, não necessariamente no nível do módulo
  - Dependências opcionais
  - Importação dinâmica sem “sintaxe alternativa”
- ImportError
  - Pode ser usado para buscar alternativas em tempo de execução
  - Recursos de diferentes versões do Python

nome/__init__.py ao invés de nome.py? Package!

- Tipos de importação
  - Relativo (e.g. audiolazy/tests/__init__.py)

```
from ..lazy_compat import meta
```

  - Absoluto

```
from audiolazy.lazy_compat import meta
```

- Denotam uma estrutura aninhada (explícita em diretórios)
- Há módulos com "." no nome (chave) em sys.modules, representando esse aninhamento (e.g. o próprio "audiolazy.lazy_compat")
- Packages são módulos
  - Atributo __path__, contendo uma lista (para ser mutável?) com uma string contendo o diretório do package
  - Arquivos dos módulos aninhados podem ser encontrados pelo nome junto ao __path__ do package

Hacking __path__

- try.py
- pkg/
  - __init__.py
  - a.py
  - subpkg/
    - __init__.py
    - b.py
    - a.py

```
from pkg.subpkg.b import hw
hw()
```

Início dos outros 5 arquivos (todos exceto try.py)

```
if "__path__" in locals():
  print(  "Imported package {}\n"
        "  from this path: {}\n  filename: {}"
        .format(__name__, __path__[0], __file__))
else:
  print("Imported module {}\n  filename: {}"
        .format(__name__, __file__))
```

Implementar o hw() nos arquivos a.py diferentemente, e.g.

```
# pkg/a.py
def hw():
    print("HW")
```

```
# pkg/subpkg/a.py
def hw():
    print("Fake HW")
```

b.py importará hw, garantindo que o try.py funcione

```
from ..a import hw
```

```
from pkg.subpkg.b import hw
hw()
```

- try.py
- pkg/
  - __init__.py
  - a.py
  - subpkg/
    - __init__.py
    - b.py
    - a.py

```
# pkg/a.py
def hw():
    print("HW")
```

```
# pkg/subpkg/a.py
def hw():
    print("Fake HW")
```

```
# pkg/subpkg/b.py
from ..a import hw
```

```
Imported package pkg
  from this path: /home/danilo/pkg
  filename: /home/danilo/pkg/__init__.py
Imported package pkg.subpkg
  from this path: /home/danilo/pkg/subpkg
  filename: /home/danilo/pkg/subpkg/__init__.py
Imported module pkg.subpkg.b
  filename: /home/danilo/pkg/subpkg/b.py
Imported module pkg.a
  filename: /home/danilo/pkg/a.py
HW
```

```
# pkg/subpkg/b.py
from .a import hw
```

```
Imported package pkg
  from this path: /home/danilo/pkg
  filename: /home/danilo/pkg/__init__.pyc
Imported package pkg.subpkg
  from this path: /home/danilo/pkg/subpkg
  filename: /home/danilo/pkg/subpkg/__init__.pyc
Imported module pkg.subpkg.b
  filename: /home/danilo/pkg/subpkg/b.py
Imported module pkg.subpkg.a
  filename: /home/danilo/pkg/subpkg/a.pyc
Fake HW
```

Transformando módulo em package criando um __path__ (apenas Python 2)

```
from pkg.subpkg.b import hw
hw()
```

```
# pkg/a.py
def hw():
    print("HW")
```

```
# pkg/subpkg/a.py
def hw():
    print("Fake HW")
```

- try.py
- pkg/
  - __init__.py
  - a.py
  - subpkg/
    - __init__.py
    - b.py
    - a.py

```
# pkg/subkg/b.py
__path__ = ["."]
from ..a import hw
```

Ele importa pkg.subpkg.a (exibe “HW”) no Python 3

```
Imported package pkg
  from this path: /home/danilo/pkg
  filename: /home/danilo/pkg/__init__.py
Imported package pkg.subpkg
  from this path: /home/danilo/pkg/subpkg
  filename: /home/danilo/pkg/subpkg/__init__.py
Imported module pkg.subpkg.b
  filename: /home/danilo/pkg/subpkg/b.py
Imported module pkg.subpkg.a
  filename: /home/danilo/pkg/subpkg/a.py
Fake HW
```

```
from pkg.subpkg.b import hw
hw()
```

- try.py
- pkg/
  - __init__.py
  - a.py
  - subpkg/
    - __init__.py
    - b.py
    - a.py

```
# pkg/a.py
def hw():
    print("HW")
```

```
# pkg/subpkg/a.py
def hw():
    print("Fake HW")
```

Trocando o __path__ de um package

```
# pkg/subkg/b.py
from .. import __path__ as path_pkg
from . import __path__ as path_subpkg
path_pkg[:] = path_subpkg
from ..a import hw
```

```
Imported package pkg
  from this path: /home/danilo/pkg
  filename: /home/danilo/pkg/__init__.pyc
Imported package pkg.subpkg
  from this path: /home/danilo/pkg/subpkg
  filename: /home/danilo/pkg/subpkg/__init__.pyc
Imported module pkg.subpkg.b
  filename: /home/danilo/pkg/subpkg/b.py
Imported module pkg.a
  filename: /home/danilo/pkg/subpkg/a.pyc
Fake HW
```

Detecção da estrutura do pacote

- Importar o pacote não importa os módulos do diretório
- AudioLazy usa o __path__
  - Módulo os (listar arquivos)
  - Detecta os módulos existentes no pacote
  - Importa todos os módulos
  - “Despeja” os nomes relevantes no namespace principal
    - __all__ no módulo “m” → nomes do “`from m import *`”
  - Sumários automáticos nas docstrings dos módulos
  - Uso de exec

```
# __init__.py da AudioLazy
__modules__, __all__, __doc__ = \
  __import__(__name__ + "._internals", fromlist=[__name__]
             ).init_package(__path__, __name__, __doc__)
exec(("from .{} import *\n" * len(__modules__)).format(*__modules__))
```

Macropy
Macros sintáticas no Python

```
from macropy.tracing import macros, trace

def first_primes():
  yield 2
  yield 3
  yield 5
  yield 7

with trace:
    prod = 1
    for i in first_primes():
        prod = prod * (i ** 2)

# Saída fornecida:

#prod = 1
#for i in first_primes():
#    prod = prod * (i ** 2)
#first_primes() -> <generator object first_primes at 0x7f83c08266e0>
#prod = prod * (i ** 2)
#i ** 2 -> 4
#prod * (i ** 2) -> 4
#prod = prod * (i ** 2)
#i ** 2 -> 9
#prod * (i ** 2) -> 36
#prod = prod * (i ** 2)
#i ** 2 -> 25
#prod * (i ** 2) -> 900
#prod = prod * (i ** 2)
#i ** 2 -> 49
#prod * (i ** 2) -> 44100
```

Personaliza a importação e/ou o REPL

```
# Digitar no REPL antes de começar a usar
import macropy.console
```

```
from macropy.string_interp import macros, s
a, b, c = "texto", "sem", "modificar"
s["{b[::-1] + c[:2]} {a}, {c[:-1]}do"]
# Out[1]: 'mesmo texto, modificado'
```

Parte 7

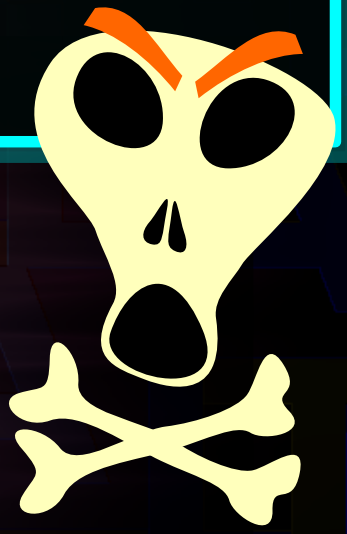
JIT, exec, eval, compile

Built-ins
exec e eval

```
# NÃO RODE ISTO! Equivale a rm -rf /
"""
import os
for root, dirs, files in os.walk("/"):
  for fname in files:
    try:
      os.remove(os.path.join(root, fname))
    except:
      pass
"""
```

- exec
  - Statement(s) como string
  - Efeito colateral
  - Globals e locals
  - Python 2: exec é um statement
  - Python 3: exec é um objeto
- eval
  - Expressão como string
- Já foi utilizado anteriormente em outros slides

Preocupação de segurança ao rodar código arbitrário



Há um uso considerável do recurso na AudioLazy, e.g. as 7 estratégias de window e wsymm são inteiramente geradas por templates, a inicialização usa exec, etc.

JIT (Just In Time)

- Filtros LTI e lineares variantes no tempo da AudioLazy
  - Cria a função do filtro (como uma string) em tempo de execução imediatamente antes de utilizá-la
- PyPy e Numba (LLVM)
  - Compilação JIT eficiente do bytecode Python para LLVM/nativo

```
from audiolazy import z

filt = z ** -2 / (- 2 * z ** -1 + 1)

# Ao usar o filt, ele gerará
def gen(seq, memory, zero):
  m1 , = memory
  d1 = d2 = zero
  for d0 in seq:
    m0 = d2 + --2 * m1
    yield m0
    m1 = m0
    d2 = d1
    d1 = d0
```

Built-in compile

- Gera bytecode Python
- Similar ao eval/exec mas permite múltiplos usos do resultado
  - Uso posterior com o exec
  - É possível criar uma função com o código
    - Talvez seja mais fácil rodar uma vez e manter o valor do resultado

```
# coding: utf-8
from types import CodeType, FunctionType

source = """print('Hello World')"""
fname = "<string>"
mode = "single" # exec   -> module
                # single -> statement
                # eval   -> expression
code = compile(source, fname, mode)

exec(code) # Hello World
print(isinstance(code, CodeType)) # True

# Criando uma função (para não usar o exec)
name = "hw"
defaults = tuple()
hw = FunctionType(code, locals(), name,
                  defaults, None)
hw() # Hello World
```

Mudança de nome de variável livre em closure

```python
# Apenas Python 2
from types import CodeType, FunctionType

def change_internal_name(func, old, new):
  fc = func.func_code
  kws = {el[3:]: getattr(fc, el)
         for el in dir(fc) if el.startswith("co_")}
  kws["names"] = tuple(new if el == old else el
                       for el in kws["names"])
  args = (kws[p] for p in ["argcount", "nlocals",
          "stacksize", "flags", "code", "consts",
          "names", "varnames", "filename", "name",
          "firstlineno", "lnotab", "freevars",
          "cellvars"])
  return FunctionType(CodeType(*args),
            func.func_globals, func.func_name,
            func.func_defaults, func.func_closure)
```

```python
a, b = 32, 42
def test():
    return b
print test()
print change_internal_name(test, old="b", new="a")()
print test()
test = change_internal_name(test, old="b", new="a")
print test()
test = change_internal_name(test, old="a", new="b")
print test()
```

```
# 42 (b)
# 32 (a)
# 42 (b)
# 32 (a)
# 42 (b)
```

Parte 8

AST e bytecode

AST
Abstract Syntax Tree

Exemplo de uso adaptado do setup.py da AudioLazy

```python
import ast

def locals_from_exec(code):
  """ Qualified exec, returning the locals dict """
  namespace = {}
  exec(code, {}, namespace)
  return namespace

def pseudo_import(fname):
  """
  Namespace dict from assignments in the file without
  ``__import__``
  """
  is_d_import = lambda n: isinstance(n, ast.Name
                                     ) and n.id == "__import__"
  is_assign = lambda n: isinstance(n, ast.Assign)
  is_valid = lambda n: is_assign(n) and not any(map(is_d_import,
                                                    ast.walk(n)))
  with open(fname, "r") as f:
    astree = ast.parse(f.read(), filename=fname)
  astree.body = [node for node in astree.body if is_valid(node)]
  return locals_from_exec(compile(astree, fname, mode="exec"))
```

Bytecode

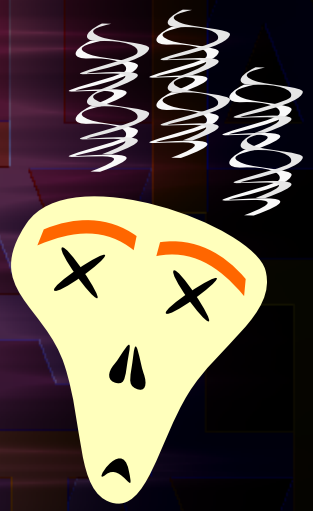
- Expressões em “notação polonesa reversa” (pós-fixa) “Calculadora HP”
- Módulo dis
  - “Disassembly”
  - Exibição do bytecode Python em uma notação amigável
- É possível re-sintetizar funções mudando o bytecode delas

```python
# coding: utf-8
import dis, sys
from types import CodeType, FunctionType
PY2 = sys.version_info.major == 2

# Vamos trocar o operador ** em f:
f = lambda x, y: x ** 2 + y
fc = f.func_code if PY2 else f.__code__
print(f(5, 7)) # 32
dis.dis(f) # "Disassembly" do bytecode Python

# Exibe os valores do bytecode em hexadecimal
code = fc.co_code
if PY2:
    code = map(ord, code)
print(" ".join("%02x" % val for val in code))
# 7c 00 00 64 01 00 13 7c 01 00 17 53
#                  ^^
#                  Valor que queremos mudar

# Armazena os valores do objeto code
kws = {el[3:]: getattr(fc, el)
       for el in dir(fc) if el.startswith("co_")}

# Troca o primeiro BINARY_POWER por BINARY_ADD
kws["code"] = kws["code"].replace(b"\x13", b"\x17", 1)
```

```
# 0 LOAD_FAST     0 (x)
# 3 LOAD_CONST    1 (2)
# 6 BINARY_POWER
# 7 LOAD_FAST     1 (y)
# 10 BINARY_ADD
# 11 RETURN_VALUE
```

Manipulação de bytecode

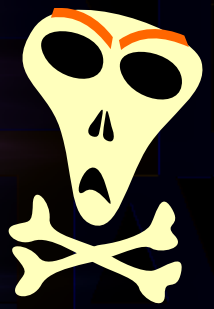
```
# Ordena os valores do objeto code e cria o novo code
args = [kws[p] for p in ["argcount", "nlocals",
        "stacksize", "flags", "code", "consts",
        "names", "varnames", "filename", "name",
        "firstlineno", "lnotab", "freevars",
        "cellvars"]]
if not PY2:
    args.insert(1, 0) # No keyword-only argument
new_code = CodeType(*args)

# Cria a nova função
ftpl = "func_%s" if PY2 else "__%s__"
fparams = ["globals", "name", "defaults", "closure"]
fargs = (getattr(f, ftpl % n) for n in fparams)
new_f = FunctionType(new_code, *fargs)

# Voilà!
dis.dis(new_f)

print(new_f(5, 7)) # 14
# Efetivamente lambda x, y: x + 2 + y
```



```
# 0 LOAD_FAST      0 (x)
# 3 LOAD_CONST     1 (2)
# 6 BINARY_ADD
# 7 LOAD_FAST      1 (y)
# 10 BINARY_ADD
# 11 RETURN_VALUE
```

Parte 9

Contexto, aplicações e futuro



Alguns exemplos de quem utiliza algum esses recursos

- AudioLazy
- Macropy
- py.test
- Django
- Sphinx e plugins (e.g. numpydoc)
- six
- Numpy
- sms-tools (manipula sys.path)
- Weave
- Hy
- Sympy
- Twisted
- Pygments
- PyNes
- Standard Library do Python (e.g. from string import Template)

Etc...

O difícil é achar quem NÃO utiliza recursos como decorators, closures, geradores, * e ** unários, reflection, etc.

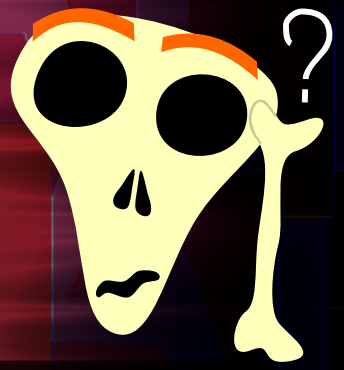
Presente...

- Python (-)
  - “Conflitos sociais”
    - 2 VS 3, mas PEP404 diz que não haverá Python 2.8
    - “dream-python”
  - Global Interpreter Lock (GIL)
  - Desempenho?
- Python (+)
  - PyPI (Python Package Index)
    - 54k+ pacotes/bibliotecas
    - pip, virtualenv
  - Documentação (+ reflexão)
  - Standard library, frameworks prontos, integração com outras linguagens, etc.
- Python e seu uso didático
  - MOOCs (Udacity, Coursera, edX, etc.)
  - Declaração de P. Norvig sobre o AIMA (Livro de inteligência artificial)
  - Universidades
    - “Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities”
      - http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext

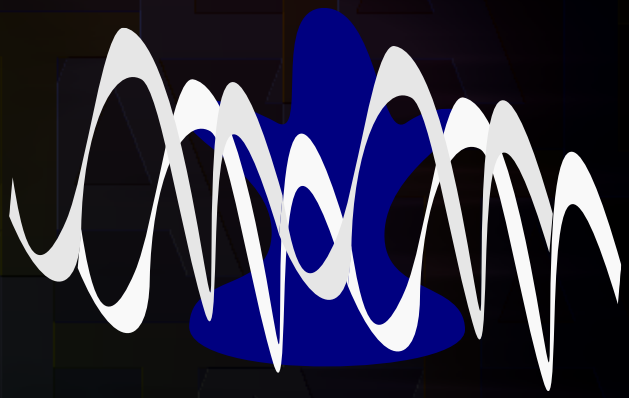
- Outras linguagens (dinâmicas)
  - Ruby
  - JavaScript
  - Erlang
  - Julia
- Functional VS expression-oriented
  - http://richardminerich.com/2012/07/functional-programming-is-dead-long-live-expression-oriented-programming/

… Futuro?

Outras possibilidades (?)



- Descriptors (__get__, __set__, __delete__, @property)
- Uso em suites de testes (e.g. py.test)
- Continuation
- Especificidades dos módulos inspect, ast, dis
- Hibridização com o Sympy (CAS)
  - Usando matemática simbólica para gerar código
- Co-rotinas
- Otimização
  - Geração de código nativo, LLVM (Numba), etc.
  - Mensurar desempenho (empiricamente)
- Garbage collector
- Stack frame / trace
- Python C API
- Novos recursos (e.g. __prepare__ em metaclasses)
- Interpretadores (CPython, Stackless, PyPy, ...)
- …



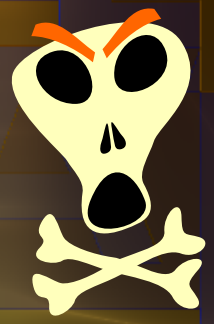
Softwares usados

- Python 2.7.8 e 3.4.2
  - Standard library (itertools, functools, etc.)
- IPython 2.3.0
- Pacotes/bibliotecas Python
  - AudioLazy
  - Pygments (cores no código)
  - Cachetools (lru_cache no Python 2)
  - Matplotlib
- LibreOffice
- Inkscape

Background dos slides feito com AudioLazy + Pylab (Matplotlib + Numpy)

Únicas imagens coletadas da internet (sites oficiais):

- WTA2015 (canto slides)
- Python (2x)
- OSI (Open Source)



Fim!

Dúvidas ou comentários?

Thx!

Links (referências) exibidas no navegador durante a apresentação

- Programação funcional
  - Hughes J. “Why Functional Programming Matters”. The Computer Journal, vol. 32, issue 2, 1989, p 98-107.
    - http://comjnl.oxfordjournals.org/content/32/2/98.short
  - Artigos “Lambda: The Ultimate *” dos criadores do Scheme
    - http://library.readscheme.org/page1.html
- Python
  - Site oficial: https://www.python.org/
    - Documentação oficial: https://docs.python.org
      - Standard Library: https://docs.python.org/3/library/index.html
      - Linguagem: https://docs.python.org/3/reference/index.html
        - Data model: https://docs.python.org/3/reference/datamodel.html
    - PEP index: https://www.python.org/dev/peps/
      - PEP255 (Generators): https://www.python.org/dev/peps/pep-0255

Códigos digitados durante o tutorial

```
$ echo abc | tee apagar.txt
abc
$ cat apagar.txt
abc
```

Shell script: tee

IPython: __getattr__

```
In [1]: %paste
# coding: utf-8
from random import choice, randint
from string import ascii_lowercase

def mastermind(guess, secret):
    """
    Compara as strings de entrada e devolve um par de inteiros
    (caracteres corretos na posição correta,
     caracteres corretos se desconsiderarmos a posição)

    Origem: https://gist.github.com/danilobellini/5311427
    """
    return sum(1 for g, s in zip(guess, secret) if g == s), \
           sum(min(guess.count(x), secret.count(x)) for x in set(secret))

class NameMastermind(object):
    def __init__(self):
        size = randint(3, 8)
        name = "".join(choice(ascii_lowercase) for el in xrange(size))
        self._name = name
        setattr(self, name, lambda: "Yeah!")
    def __getattr__(self, name):
        return lambda: mastermind(name, self._name)

## -- End pasted text --

In [2]: game = NameMastermind()

In [3]: game.abcd()
Out[3]: (0, 1)

In [4]: game.a()
Out[4]: (0, 0)

In [5]: game.b()
Out[5]: (0, 1)

In [6]: game.ab()
Out[6]: (0, 1)

In [7]: game.aab()
Out[7]: (1, 1)

In [8]: game.efgh
Out[8]: <function __main__.<lambda>>

In [9]: game.efgh()
Out[9]: (0, 0)

In [10]: game.efghA()
Out[10]: (0, 0)

In [11]: game.ijkl()
Out[11]: (0, 0)

In [12]: game._name
Out[12]: 'urbrsr'

In [13]: game.urbrst()
Out[13]: (5, 5)

In [14]: game.urbrrs()
Out[14]: (4, 6)

In [15]: game.urbrsr()
Out[15]: 'Yeah!'

In [16]: game2 = NameMastermind()

In [17]: game2._name
Out[17]: 'vxbjqzy'

In [18]: game2.urbrsr()
Out[18]: (1, 1)
```

Algum valor aleatório diferente de urbrsr

```
julia> f(x) = x * x
f (generic function with 1 method)

julia> code_llvm(f, (Float64,))

define double @julia_f_20166(double) {
top:
  %1 = fmul double %0, %0, !dbg !857
  ret double %1, !dbg !857
}

julia> code_native(f, (Float64,))
        .text
Filename: none
Source line: 1
        push    RBP
        mov     RBP, RSP
Source line: 1
        mulsd   XMM0, XMM0
        pop     RBP
        ret

julia> code_native(f, (Int,))
        .text
Filename: none
Source line: 1
        push    RBP
        mov     RBP, RSP
Source line: 1
        imul    RDI, RDI
        mov     RAX, RDI
        pop     RBP
        ret
```

Julia: JIT

IPython: metaclass

```
In [19]: from string import Template

In [20]: Template.__metaclass__
Out[20]: string._TemplateMetaclass
```