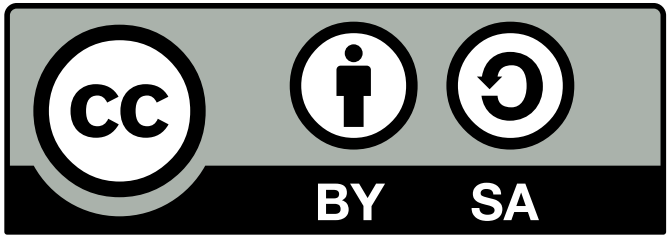
Devenir incollable sur les callables !

Devenir incollable sur les callables !

Antoine “entwanne” Rozo



Alma

Devenir incollable sur les callables !

- Comprendre ce que sont les callables au-delà des fonctions
- Découvrir ce qu'il se passe lors d'un appel
- Petit tour du côté des décorateurs

- https://github.com/entwanne/presentation_callables

Fonctions

Fonctions

- Une fonction est une opération qui calcule un résultat qu'elle renvoie à partir d'arguments qu'on lui donne
- Elle est appelée à l'aide d'une paire de parenthèses

```
>>> abs(5)

>>> abs(-1.2)
```

Fonctions

- Elle travaille sur tous types de valeurs

```
>>> len('Hello')
```

```
>>> all([True, 'foo', 4, {'key': 0}])
```

Fonctions

- Une fonction ne prend pas nécessairement d'arguments

```
>>> input()
```

Fonctions

- Mais renvoie toujours une et une seule valeur

```
>>> print("abc")
```

```
>>> ret = print("abc")
>>> print(ret)
```

```
>>> divmod(7, 2)
```

```
>>> a, b = divmod(7, 2)
```

Fonctions

- Tout appel de fonction peut ainsi être utilisé au sein d'une expression

```
>>> round(3.5) * 2 + abs(-5)
```

Fonctions

- Une fonction ne renvoie pas nécessairement toujours la même valeur pour les mêmes arguments

```
>>> import random
>>> random.randrange(10)
```

```
>>> it = iter(range(10))
>>> next(it)
```

Définition de fonction

- On définit une fonction à l'aide du mot-clé `def`
- La définition associe une fonction à un nom et lui spécifie une liste de paramètres
- Le bloc de code suivant la ligne de définition est le corps de la fonction
- La valeur de retour d'une fonction est renvoyée à l'aide du mot-clé `return`

```
def addition(a, b):
    return a + b
```

```
>>> addition(3, 5)
```

Autres callables

Callables

- Les fonctions sont des callables / appelables (objets que l'on peut appeler à l'aide de parenthèses)
- Mais il n'y a pas que les fonctions qui sont appelables

Types

- Les types sont appelables, pour en créer des instances

```
>>> int('123')
```

```
>>> str()
```

```
>>> range(10)
```

Méthodes

- Les méthodes des objets sont appelables

```
>>> "Hello".replace("l", "_")
```

- De même que les méthodes de classes

```
>>> dict.fromkeys([1, 2, 3])
```

Lambdas

- Les lambdas sont des définitions de fonctions sous forme d'expressions

```
>>> f = lambda x: x+1
>>> f(5)
```

```
>>> (lambda i: i**2)(4)
```

- On parle aussi de fonctions anonymes

Autres

- Les objects `functools.partial` sont des appels partiels de fonctions
- Ils stockent les arguments donnés à la création pour les réutiliser lors de l'appel final

```
>>> import functools
>>> f = functools.partial(max, 0)
>>> f(1, 2)
```

```
>>> f(-1, -2)
```

Arguments des appelables

Arguments

- Les arguments sont les valeurs envoyées à un *callable*

```
>>> addition(3, 5)
```

- Ils peuvent être positionnels ou nommés

```
>>> addition(a=3, b=5)
```

Ordre de placement des arguments

- Les arguments positionnels se placent toujours avant les arguments nommés

```
>>> addition(3, b=5)
```

```
>>> addition(a=3, 5)
```

Arguments et paramètres

- Ne pas confondre les arguments avec les paramètres qui sont les variables dans lesquelles sont reçus les arguments
- Un argument ne peut correspondre qu'à un seul paramètre

Arguments et paramètres

- `a` et `b` sont les paramètres de la fonction `addition` `python def addition(a, b): return a + b`
- `3` et `5` sont les arguments de l'appel `python addition(3, b=5)`
- `5` est un argument nommé associé au nom `b`

Arguments et paramètres

- Les arguments positionnels remplissent les paramètres de la gauche vers la droite
- Tandis que les arguments nommés remplissent les paramètres correspondant à leurs noms

Valeur par défaut

- Un paramètre peut définir une valeur par défaut
- Elle sera utilisée si aucun argument n'est fourni pour ce paramètre

```
def multiplication(a, b=1):
    return a * b
```

```
>>> multiplication(3, 4)
```

```
>>> multiplication(5)
```

Valeur par défaut

- Attention : cette valeur par défaut est définie une seule fois pour la fonction

```
>>> def append(item, dest=[]):
...     dest.append(item)
...     return dest
```

```
>>> append(4, [1, 2, 3])
```

```
>>> append('hello')
```

Différentes sortes de paramètres

- Les paramètres peuvent être de plusieurs sortes :
  - *posititonal-only*, qui ne peuvent recevoir que des arguments positionnels
  - *keyword-only*, qui ne peuvent recevoir que des arguments nommés
  - *positional-or-keyword*, qui peuvent recevoir à la fois des arguments positionnels ou nommés
- Jusqu'ici nos paramètres étaient tous de type *positional-or-keyword*

Différentes sortes de paramètres

- Il est possible à l'aide des délimiteurs / et * de spécifier la sorte de nos paramètres :
  - Les paramètres placés avant / sont *positional-only*
  - Les paramètres placés après * sont *keyword-only*
  - Les autres paramètres sont *positional-or-keyword*
- Ces délimiteurs sont bien sûr optionnels

Différentes sortes de paramètres

```
def function(first, /, second, third, *, fourth):
    ...
```

- `first` est *positional-only*
- `second` et `third` sont *positional-or-keyword*
- `fourth` est *positional-only*

Différentes sortes de paramètres

- À l'appel cela signifie que `first` ne peut pas recevoir d'argument nommé et `fourth` ne peut pas recevoir d'argument positionnel

```
>>> function(1, 2, 3, fourth=4)

>>> function(1, second=2, third=3, fourth=4)

>>> function(1, 2, 3, 4)

>>> function(first=1, second=2, third=3, fourth=4)
```

Ordre de définition des paramètres

- Les délimiteurs font qu'il y a un ordre à respecter pour définir les différents paramètres
  - D'abord *positional-only*, puis *positional-or-keyword* et enfin *keyword-only*
- Les valeurs par défaut des paramètres sont aussi à prendre en compte dans l'ordre de définition

Ordre de définition des paramètres

- Chez les arguments qui peuvent être positionnels (*positional-only* ou *positional-or-keyword*) un paramètre sans valeur par défaut ne peut pas suivre un paramètre qui en a une

```
def function(first, /, second, third=3):
    ...

def function(first, /, second=2, third):
    ...

def function(first=1, /, second, third):
    ...
```

Ordre de définition des paramètres

- Le problème ne se pose pas pour les paramètres *keyword-only* puisque l'ordre des arguments n'y a pas d'importance

```
def function(foo=None, *, bar=True, baz):
    return (foo, bar, baz)
```

```
>>> function(baz=False)
```

Paramètres variadiques

- Il existe aussi des paramètres spéciaux pour récupérer tout un ensemble d'arguments, qu'on appelle paramètres variadiques
  - Ce nom vient du fait qu'ils récupèrent un nombre variable d'arguments...
  - Autrement dit des arguments variadiques

Paramètres variadiques

- Un paramètre préfixé d'un $*$ récupère tous les arguments positionnels restants
  - Il prend alors la place du délimiteur $*$ dans la liste des paramètres
  - C'est-à-dire qu'il se place nécessairement après tous les paramètres qui peuvent recevoir des arguments positionnels
  - Il ne peut y avoir qu'un paramètre préfixé d'un $*$
  - Ce paramètre contiendra alors un tuple des arguments positionnels donnés à la fonction
  - Les arguments positionnels qui correspondent à un paramètre précis ne seront pas inclus dans ce tuple
  - Il est d'usage d'appeler ce paramètre `args`

Paramètres variadiques

```
def my_sum(*args):
    total = 0
    for item in args:
        total += item
    return total

>>> my_sum(1, 2, 3)

>>> my_sum()
```

Paramètres variadiques

- Ils peuvent bien sûr être combinés avec d'autres sortes de paramètres

```
def my_sum(first, /, *args):
    total = first
    for item in args:
        total += item
    return total
```

```
>>> my_sum(1, 2, 3)
```

```
>>> my_sum('a', 'b', 'c')
```

```
>>> my_sum()
```

Paramètres variadiques

- C'est aussi ce qui est utilisé par la fonction `print` par exemple pour accepter un nombre arbitraire d'arguments

```
>>> print(1, 'foo', ['hello', 'world'])
```

Paramètres variadiques

- De manière similaire, le préfixe ** permet de définir un paramètre qui récupère tous les arguments nommés restants
  - Ce paramètre se place nécessairement après tous les autres
  - Il est unique lui aussi
  - Il contient le dictionnaire des arguments nommés passés à la fonction
  - Seuls les arguments nommés ne correspondant à aucun autre paramètre sont présents dans ce dictionnaire
  - Il est d'usage d'appeler ce paramètre `kwargs`

Paramètres variadiques

```
def make_dict(**kwargs):
    return kwargs

>>> make_dict(foo=1, bar=2)
```

Paramètres variadiques

- Combinables eux-aussi avec d'autres sortes de paramètres

```
def make_obj(id, **kwargs):
    return {'id': id} | kwargs
```

```
>>> make_obj('#1', foo='bar')
```

```
>>> make_obj(id='#1', foo='baz')
```

Opérateurs *splat*

- Ces délimiteurs * et ** ne sont pas utilisables uniquement dans les listes de paramètres
- On les retrouve aussi dans les listes d'arguments, sous le nom d'opérateurs *splat*
- Ils ont l'effet inverse de celui en place pour les paramètres

Opérateurs *splat*

- Ainsi `*` appliqué à une liste (ou tout autre itérable) transforme ses éléments en arguments positionnels

```
>>> addition(*[3, 5])
```

```
>>> print(*(i**2 for i in range(10)))
```

Opérateurs *splat*

- Et contrairement aux paramètres, on peut appliquer le *splat* à plusieurs arguments
- On peut aussi préciser d'autres arguments

```
>>> my_sum(*range(5), 10, *range(3))
```

Opérateurs *splat*

- Quant à ** il s'applique à un dictionnaire (ou similaire) et transforme les éléments en arguments nommés
- Les clés du dictionnaires doivent alors être des chaînes de caractères

```
>>> addition(**{'a': 3, 'b': 5})
```

Unpacking

- On retrouve d'ailleurs aussi l'opérateur *splat* dans les opérations d'*unpacking*
- L'*unpacking* consiste à extraire des valeurs depuis un itérable à l'aide d'une assignation spéciale

```
>>> first, *middle, last = *range(5), 8, *range(3)

>>> first

>>> middle

>>> last
```

Signatures de fonctions

Signatures de fonctions

- La signature d'une fonction représente son interface, décrivant comment on peut l'appeler
- La fonction `signature` du module `inspect` permet de récupérer la signature d'une fonction

```
>>> import inspect
>>> inspect.signature(addition)
```

- On voit ainsi que notre fonction `addition` attend deux paramètres `a` et `b`

Signatures

- L'object renvoyé par `inspect.signature` permet d'explorer la signature de la fonction

```
>>> sig = inspect.signature(addition)
>>> sig.parameters
```

```
>>> sig.parameters.keys()
```

```
>>> sig.parameters['a']
```

```
>>> sig.parameters['a'].kind
```

Signatures

- On peut aussi connaître la valeur par défaut d'un paramètre

```
>>> sigmul = inspect.signature(multiplication)
>>> sigmul.parameters['b']
```

```
>>> sigmul.parameters['b'].default
```

Binding

- On peut utiliser une signature pour réaliser un *binding* sur des arguments
- C'est-à-dire faire la correspondance entre les arguments et les paramètres

```
>>> binding = sig.bind(3, b=5)
>>> binding
```

```
>>> binding.arguments
```

```
>>> binding.args, binding.kwargs
```

- C'est une manière de normaliser les arguments passés lors d'un appel
  - Tous ceux qui peuvent être positionnels sont stockés dans `args` et les autres dans `kwargs`

Binding

- Les mêmes vérifications ont lieu que lors d'un appel
- Il faut ainsi préciser tous les arguments nécessaires à l'appel

```
>>> sig.bind(5)
```

Binding

- Il est aussi possible d'appliquer les valeurs par défaut des paramètres sur un *binding*

```
>>> binding = sigmul.bind(10)
>>> binding
```

```
>>> binding.apply_defaults()
>>> binding
```

Modification de signature

- L'objet signature en lui-même est inaltérable
- Mais il possède une méthode `replace` pour en créer une copie modifiée

```
>>> sig.replace(parameters=[])
```

Modification de signature

- Il en est de même pour les objets représentant les paramètres

```
>>> from inspect import Parameter
>>> sig.parameters['a'].replace(kind=Parameter.POSITIONAL_O
```

Modification de signature

- On peut alors dériver notre signature pour n'accepter que les arguments positionnels

```
>>> newsig = sig.replace(parameters=[p.replace(kind=Paramet
>>> newsig
```

```
>>> newsig.bind(1, 2)
```

```
>>> newsig.bind(a=1, b=2)
```

Modification

- On peut ensuite assigner la nouvelle signature à l'attribut `__signature__` de la fonction

```
>>> addition.__signature__ = newsig
```

- Cela change bien la signature renvoyée par `inspect.signature`

```
>>> inspect.signature(addition)
```

- Mais n'affecte pas le comportement réel de la fonction

```
>>> addition(a=1, b=2)
```

Annotations

- La signature d'une fonction comprend aussi les annotations de paramètres et de retour
- Les annotations permettent de préciser les types attendus

```
def addition(a: int, b: int) -> int:
    return a + b
```

```
>>> sig = inspect.signature(addition)
>>> sig
```

Annotations

- Les annotations sont exposées dans les attributs de la signature

```
>>> sig.return_annotation
>>> sig.parameters['a'].annotation
```

Annotations

- Elles sont aussi exposées dans l'attribut spécial `__annotations__` de la fonction

```
>>> addition.__annotations__
```

- Ainsi que via `inspect.get_annotations`

```
>>> inspect.get_annotations(addition)
```

Annotations

- `inspect.get_annotations` est préférable
  - Elle gère les cas d'absence de `__annotations__` sur l'objet et différents problèmes potentiels
  - Elle permet l'évaluation dynamique des annotations

Annotations

```
def addition(a: "int", b: "int") -> "int":
    return a + b

>>> addition.__annotations__

>>> inspect.get_annotations(addition)
```

```
>>> inspect.get_annotations(addition, eval_str=True)
```

Documentation

- La documentation permet d'expliciter le comportement d'une fonction
- Cela se fait en Python à l'aide de docstrings

```
def addition(a: int, b: int) -> int:
    "Return the sum of two integers"
    return a + b
```

- Celle-ci n'est pas exposée dans la signature

Documentation

- La docstring est exposée dans l'argument `__doc__` de la fonction

```
>>> addition.__doc__
```

- Ou via `inspect.getdoc`

```
>>> inspect.getdoc(addition)
```

Documentation

- Cette dernière est préférable pour un meilleur formattage

```
def function():
    """
    Docstring of the function
    on multiple lines
    """

>>> function.__doc__

>>> inspect.getdoc(function)
```

Décorateurs

Décorateurs

- Les décorateurs sont un mécanisme permettant de transformer des callables
- Ils s'appliquent à des fonctions pour en modifier le comportement

```
import functools

@functools.cache
def addition(a, b):
    print(f'Computing {a}+{b}')
    return a + b

>>> addition(3, 5)

>>> addition(1, 2)
```

Décorateurs

- `functools.cache` a remplacé `addition` par une nouvelle fonction avec un mécanisme de cache

- La syntaxe précédente est équivalente à :

```
def addition(a, b):
    print(f'Computing {a}+{b}')
    return a + b

addition = functools.cache(addition)
```

Décorateurs

- On peut aussi appliquer plusieurs décorateurs à la suite

```
@deco1
@deco2
def function():
    ...
```

- Qui est équivalent à :

```
def function():
    ...

function = deco1(deco2(function))
```

Écriture de décorateurs

- Un décorateur est donc un callable qui reçoit un callable et renvoie un callable

```
def decorator(func):
    return func
```

```
@decorator
def addition(a, b):
    return a + b

>>> addition(3, 5)
```

Écriture de décorateurs

- Pour changer le comportement de la fonction décorée, il faut créer une nouvelle fonction reprenant son interface

```
def decorator(func):
    def wrapper(*args, **kwargs):
        print('Calling decorated function')
        return func(*args, **kwargs)
    return wrapper

@decorator
def addition(a, b):
    return a + b

>>> addition(3, 5)
```

Écriture de décorateurs

- `functools.cache` pourrait être réécrit ainsi

```
def cache(func):
    func_cache = {}

    def wrapper(*args, **kwargs):
        # make an hashable key
        key = args, tuple(kwargs.items())
        if key not in func_cache:
            func_cache[key] = func(*args, **kwargs)
        return func_cache[key]

    return wrapper
```

```
@cache
def addition(a, b):
    print(f'Computing {a}+{b}')
    return a + b

>>> addition(3, 5)
```

Décorateurs paramétrés

- Un décorateur ne peut pas être paramétré à proprement parler

- Mais on peut appeler une fonction renvoyant un décorateur

```
@functools.lru_cache(maxsize=1)
def addition(a, b):
    print(f'Computing {a}+{b}')
    return a + b

>>> addition(3, 5)

>>> addition(1, 2)
```

Décorateurs paramétrés

- Un décorateur paramétré est ainsi un callable renvoyant un callable recevant un callable et renvoyant à nouveau un callable

```
def param_decorator(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            print(f'Function decorated with {n}')
            return func(*args, **kwargs)
        return wrapper
    return decorator

@param_decorator(42)
def function():
    ...

>>> function()
```

Outils

Outils

- Python propose différents outils autour des callables
- Afin de les identifier et les manipuler

Builtins

- La fonction `callable` permet de savoir si un objet est *callable*

```
>>> callable(len)
```

```
>>> callable(str)
```

```
>>> callable(str.replace)
```

```
>>> callable(lambda: True)
```

```
>>> callable(5)
```

Module functools

- On a vu `functools.partial` pour l'application partielle
- Elle gère les arguments positionnels et nommés

```
import functools

debug = functools.partial(print, '[DEBUG]', sep=' - ')

debug(1, 2, 3)
```

Module functools

- On a écrit plus tôt des décorateurs qui enrobaient nos fonctions

```
@decorator
def addition(a: int, b: int) -> int:
    "Return the sum of two integers"
    return a + b
```

- Le problème est qu'on perd la signature et la documentation de la fonction initiale

```
>>> inspect.signature(addition)
```

```
>>> inspect.getdoc(addition)
```

Module functools

- On pourrait copier manuellement `__doc__`, `__signature__` et autres

- Mais `functools` fournit une fonction `update_wrapper` pour faire cela plus simplement

```
def decorator(func):
    def wrapper(*args, **kwargs):
        print('Calling decorated function')
        return func(*args, **kwargs)
    functools.update_wrapper(wrapper, func)
    return wrapper
```

Module functools

```
@decorator
def addition(a: int, b: int) -> int:
    "Return the sum of two integers"
    return a + b

>>> inspect.signature(addition)

>>> inspect.getdoc(addition)
```

Module functools

- Cela fonctionne entre-autres en assignant un attribut `__wrapped__` à notre fonction

```
>>> addition.__wrapped__
```

Module functools

- `functools` possède aussi un décorateur `wraps` pour faciliter cela

```
def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print('Calling decorated function')
        return func(*args, **kwargs)
    return wrapper
```

Module operator

- Le module `operator` expose les différents opérateurs du langage
- `operator.call` permet notamment d'appeler un callable (Python 3.11)

```
>>> import operator
>>> operator.call(addition, 1, 2)
```

Module operator

- La fonction `itemgetter` renvoie un callable pour récupérer un élément d'un conteneur donné

```
>>> get_name = operator.itemgetter('name')
>>> get_name({'name': 'John'})
```

```
>>> get_fullname = operator.itemgetter('firstname', 'lastna
>>> get_fullname({'firstname': 'Jude', 'lastname': 'Doe'})
```

Module operator

- Ainsi que la fonction `attrgetter` pour récupérer un attribut

```
>>> get_module = operator.attrgetter('__module__')
>>> get_module(int)
```

Module operator

- Et `methodcaller` pour appeler une méthode sur un objet donné

```
>>> replace = operator.methodcaller('replace', 'o', 'a')
>>> replace('toto')
```

Module inspect

- Et pour rappel le module `inspect` dédié à l'introspection
- `inspect.isfunction`
- `inspect.getsource`

Callables

Callables

- La builtin `callable` permet de tester si un objet est callable

- Celle-ci vérifie qu'un objet possède une méthode `__call__`

- Un callable est ainsi un objet avec une telle méthode

Callables

```
class Adder:
    def __init__(self, add):
        self.add = add

    def __call__(self, x):
        return self.add + x
```

```
>>> add_5 = Adder(5)
>>> callable(add_5)
```

```
>>> add_5(3)
```

Construction de callable

- On peut alors imaginer construire une fonction à partir d'objets callables

```
class Expr:
    def __call__(self, **env):
        raise NotImplementedError
```

Construction de callable

```
class Scalar(Expr):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return repr(self.value)

    def __call__(self, **env):
        return self.value
```

Construction de callable

```
class Operation(Expr):
    def __init__(self, op_func, op_repr, *args):
        self.func = op_func
        self.repr = op_repr
        self.args = args

    def __repr__(self):
        return self.repr(*self.args)

    def __call__(self, **env):
        values = (arg(**env) for arg in self.args)
        return self.func(*values)
```

Construction de callable

```
class Symbol(Expr):
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return self.name

    def __call__(self, **env):
        if self.name in env:
            return env[self.name]
        return self
```

Construction de callable

```
def make_op(op_func, op_fmt):
    return functools.partial(Operation, op_func, op_fmt)

add = make_op(operator.add, '{} + {}'.format)
sub = make_op(operator.sub, '{} - {}'.format)
mul = make_op(operator.mul, '{} * {}'.format)
div = make_op(operator.truediv, '{} / {}'.format)
fdiv = make_op(operator.floordiv, '{} // {}'.format)
mod = make_op(operator.mod, '{} % {}'.format)
pow = make_op(operator.pow, '{} ** {}'.format)
```

Construction de callable

```
>>> expr = add(pow(Symbol('x'), Scalar(2)), Scalar(5))
>>> expr

>>> expr(x=3)
```

Construction de callable

```
def function(func):
    def op_repr(*args):
        return f"{func.__name__}({', '.join(repr(arg) for a
    return functools.partial(Operation, func, op_repr)

>>> import math
>>> expr = function(math.cos)(mul(Symbol('x'), Scalar(math.
>>> expr

>>> expr(x=1)
```

Construction de callable

```
    def ensure_expr(x):
        if isinstance(x, Expr):
            return x
        return Scalar(x)

    def binop(op_func):
        return lambda lhs, rhs: op_func(ensure_expr(lhs), ensur

    def rev_binop(op_func):
        return lambda rhs, lhs: op_func(ensure_expr(lhs), ensur

    Expr.__add__ = binop(add)
    Expr.__radd__ = rev_binop(add)
    Expr.__sub__ = binop(sub)
    Expr.__rsub__ = rev_binop(sub)
    Expr.__mul__ = binop(mul)
    Expr.__rmul__ = rev_binop(mul)
    Expr.__truediv__ = binop(div)
    Expr.__rtruediv__ = rev_binop(div)
```

Construction de callable

```
>>> expr = 3 * Symbol('x') ** 2 + 2
>>> expr

>>> expr(x=10)
```

Construction de fonction

Fonctions VS callables

- Si une fonction est un callable, alors elle possède une méthode `__call__`

- Mais vers quoi pointe cette méthode ?

```
>>> addition.__call__

>>> addition.__call__(3, 5)
```

```
>>> addition.__call__.__call__.__call__(3, 5)
```

- Un autre mécanisme est donc nécessaire

Fonctions VS callables

- Les fonctions possèdent un attribut `__code__`
- Celui-ci contient le code (compilé) de la fonction

```
def function():
    print('hello')

>>> function.__code__
```

- Ce code est un objet exécutable

```
>>> exec(function.__code__)
```

- Du moins pour les fonctions sans paramètres

Construction de fonctions

- Une fonction est donc un enrobage autour d'un objet code

- On peut construire une fonction en créant une instance `FunctionType` du module `types`

```
>>> import types
>>> newfunc = types.FunctionType(function.__code__, globals

>>> newfunc()
```

- Mais comment construire un objet code ?

Construction de fonctions

- La fonction `compile` permet cela
- À partir d'un AST Python ou de code brut

- Un nœud `ast.FunctionDef` est alors nécessaire pour construire une fonction
- Celui-ci définit le nom, les paramètres et le corps de la fonction

Construction de fonctions

```
>>> import ast
>>> func_body = ast.parse("print('hello')").body
>>> func_body
```

```
>>> fdef = ast.FunctionDef(
...     name='f',
...     args=ast.arguments(posonlyargs=[], args=[], kwonlya
...     body=func_body,
...     lineno=0,
...     col_offset=0,
...     decorator_list=[],
... )
```

```
>>> code = compile(ast.Module(body=[fdef], type_ignores=[])
>>> func_code = code.co_consts[0]
>>> func_code
```

```
>>> f = types.FunctionType(func_code, {})
>>> f()
```

Construction de fonctions

```
    def create_function(name, body, arg_names):
        function_body = ast.parse(body).body
        args = [ast.arg(arg=arg_name, lineno=0, col_offset=0) f

        function_def = ast.FunctionDef(
            name=name,
            args=ast.arguments(
                posonlyargs=[],
                args=args,
                kwonlyargs=[],
                defaults=[],
                kw_defaults=[]),
            body = function_body,
            decorator_list=[],
            lineno=0,
            col_offset=0,
        )
        module = compile(ast.Module(body=[function_def], type_i
```

Construction de fonctions

```
>>> addition = create_function('addition', 'return a + b',
>>> addition

>>> addition(3, 5)
```

Conclusion

Conclusion

```
conclusion()
```

- https://github.com/entwanne/presentation_callables

Questions ?