In [1]:

```
# merci PEP 526 !
_: print((list(map(_.append, map(chr, [104,101,108,108,111]))),
          "".join(_))[1]) = []
```

```
hello
```

L’interpréteur Python, quel sale type

PyConFR 2017, Toulouse

par Serge « sans paille » Guelton

avec la bénédiction de QuarksLab

Round 0

Quel type pour...

```
In [2]:  id  # id(obj: Any) -> int

Out[2]:  <function id>
```

```
In [3]:  int  # int(obj: SupportsInt) -> int

Out[3]:  int
```

```
In [4]:  list.append  # list.append(self: List[T], obj: T) -> None

Out[4]:  <method 'append' of 'list' objects>
```

Round 1

In [5]:

```
from typing import TypeVar  # PEP 484
T = TypeVar('T')

def add(self: T, other: T) -> T:  # PEP 3107
    return self + other
```

In [6]:

```
from typing import List, Tuple
t0: int = add(1, 2)
t1: List[int] = add([1], [2])
t2: Tuple = add((1,), (2,))
```

Round 2

In [7]:

```
from typing import List

l: List[int] = [1,2,3]

def indexer(i):
    return l[i]
```

In [8]:

```python
from typing import overload, Iterable
@overload
def indexer(i: int) -> int:
    pass
@overload
def indexer(i: slice) -> Iterable[int]:
    pass
def indexer(i):
    return l[i]
```

Round 3

Quel type pour

In [9]:

```
def bar(x):
    return str(x)

def foo(x):
    return bar if int(x) else None
```

In [10]:

```python
from typing import Optional, SupportsInt, Any, Callable

def bar(x: Any) -> str:
    return str(x)

def foo(x: SupportsInt) -> Optional[Callable[[Any], str]]:
    return bar if int(x) else None
```

In [11]:

```
from typing import Iterable, Tuple, List

x: Iterable[int] = range(3)
y: Iterable[int] = reversed(range(3))
l0: Iterable[Tuple[int, int]] = zip(x, y)
l1: Tuple[Iterable[int], Iterable[int]] = zip(*l0)
```

In [12]:

```
from random import randint
n = randint(1, 4)
t0: List[Tuple[int]] = [(1, )] * n
l2: Tuple[(int,) * n] = zip(*t0)
```

K.O.

```
In [13]: l: Any = eval("1 + 2")
```

```
In [14]: %%file pyconfr2017/ko.py
         a: int = 1
```

```
Overwriting pyconfr2017/ko.py
```

```
In [15]: ko = __import__("pyconfr2017.ko")
```

Pour avoir l’air savant

typage nominal

Utilise le nom du type des objets pour faire les calculs de type

typage structurel

Utilise les membres des objets pour faire des calculs de type

Python

Python utilise le *duck typing*

Typage **structurel** à l'exécution

Suis-je un fan de parkour ?

In [16]:

```
def isiterable0(x):  # nominal
    return isinstance(x, (set, tuple, list, dict, str))

def isiterable1(x):  # structurel
    return hasattr(x, '__iter__')
```

In [17]:

```
def isiterable2(x):  # duck type v0
    try:
        x.__iter__()
        return True
    except:
        return False
```

In [18]:

```
def isiterable3(x):  # duck type v1
    try:
        iter(x)
        return True
    except:
        return False
```

Quizz

- Quelle différence entre `isiterable1` et `isiterable2` ?
- Quelle différence entre `isiterable2` et `isiterable3` ?

Approche de haut niveau

In [19]:

```
from collections.abc import Iterable

def isiterable(x):
    return isinstance(x, Iterable)
```

In [20]:

```
def check_iterable(l):
    return isiterable0(l), isiterable1(l), isiterable2(l), isiterable3(l), isiterab
le(l)

l = [1, 2, 3, 5]
check_iterable(l)
```

Out[20]:

```
(True, True, True, True, True)
```

Peut-on se `mock`er ?

In [21]:

```
class EmptySequence(object):

    def __iter__(self): yield
    def __len__(self): return 0

es = EmptySequence()

check_iterable(es)
```

Out[21]:

```
(False, True, True, True, True)
```

Test de robustesse n°0

In [22]:

```
class Infinity(object):
    def __getitem__(self, _):
        return 0

infnty = Infinity()
check_iterable(infnty)
```

Out[22]:

```
(False, False, False, True, False)
```

Test de robustesse n°1

```
In [23]:
class Hole(object):
    def __iter__(self, _):
        pass

h = Hole()
check_iterable(h)
```

```
Out[23]: (False, True, False, False, True)
```

__subclasshook__

In [24]:

```python
import abc  # PEP 3119

class Appendable(abc.ABC):

    @classmethod
    def __subclasshook__(cls, C):
        return any('append' in B.__dict__ for B in C.mro())


class DevNull(object):

    def append(self, value):
        pass

    def __len__(self):
        return 0
```

```
In [25]: issubclass(DevNull, Appendable)

Out[25]: True
```

In [26]:

```
class TraitsFactory(object):
    def __getitem__(self, method_names):
        if not isinstance(method_names, tuple):
            method_names = method_names,

        class SlotCheck(abc.ABC):

            @classmethod
            def __subclasshook__(cls, C):
                return all(any(method_name in B.__dict__ for B in C.mro())
                           for method_name in method_names)
        return SlotCheck
Members = TraitsFactory()
```

```
In [27]: issubclass(DevNull, Members['append', '__len__'])

Out[27]: True
```

```
In [28]: issubclass(DevNull, Members['append', 'clear'])

Out[28]: False
```

NB ça ne teste que les *noms* de membre, pas leur type...

Contrat implicite

In [29]:

```
class Twice(list):
    def append(self, value, times):
        for _ in range(times):
            super(Twice, self).append(value)

tw = Twice()
tw.append("ore", 2)
len(tw)
```

Out[29]:

```
2
```

In [30]:

```
isinstance(tw, Members['append', '__len__'])
```

Out[30]:

```
True
```

In [31]:

```
tw.append("aison")  # gentlemna contract again
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-c3ee65ef9c33> in <module>()
----> 1 tw.append("aison")  # gentlemna contract again

TypeError: append() missing 1 required positional argument: 'times'
```

Le gadget de l’`inspect`eur

In [32]:

```
import inspect
inspect.signature(Twice.append).parameters
```

Out[32]:

```
mappingproxy({'self': <Parameter "self">,
              'times': <Parameter "times">,
              'value': <Parameter "value">})
```

Un bon moyen pour tester l’arité ?

```
In [33]: inspect.signature(list.append)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-a2a0ff704447> in <module>()
----> 1 inspect.signature(list.append)

/usr/lib/python3.6/inspect.py in signature(obj, follow_wrapped)
   3031 def signature(obj, *, follow_wrapped=True):
   3032     """Get a signature object for the passed callable."""
-> 3033     return Signature.from_callable(obj, follow_wrapped=follow_wrapped)
   3034 
   3035 

/usr/lib/python3.6/inspect.py in from_callable(cls, obj, follow_wrapped)
   2781         """Constructs Signature for the given callable object."""
   2782         return _signature_from_callable(obj, sigcls=cls,
-> 2783                                         follow_wrapper_chains=follow_wra
pped)
   2784 
   2785     @property

/usr/lib/python3.6/inspect.py in _signature_from_callable(obj, follow_wrapper_ch
ains, skip_bound_arg, sigcls)
   2260     if _signature_is_builtin(obj):
   2261         return _signature_from_builtin(sigcls, obj,
-> 2262                                        skip_bound_arg=skip_bound_arg)
   2263 
   2264     if isinstance(obj, functools.partial):

/usr/lib/python3.6/inspect.py in _signature_from_builtin(cls, func, skip_bound_a
rg)
   2085     s = getattr(func, "__text_signature__", None)
   2086     if not s:
-> 2087         raise ValueError("no signature found for builtin {!r}".format(fu
nc))
   2088 
   2089     return _signature_fromstr(cls, func, s, skip_bound_arg)
```

In [34]:

```python
from typing import Callable
isinstance(list.append, Callable)
```

Out[34]:

```
True
```

In [35]:

```
from typing import List, TypeVar; T = TypeVar('T')
isinstance(list.append, Callable[[List[T], T], None])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-8b27e89fb017> in <module>()
      1 from typing import List, TypeVar; T = TypeVar('T')
----> 2 isinstance(list.append, Callable[[List[T], T], None])

/usr/lib/python3.6/typing.py in __instancecheck__(self, instance)
   1181         # we just skip the cache check -- instance checks for generic
   1182         # classes are supposed to be rare anyways.
-> 1183         return issubclass(instance.__class__, self)
   1184 
   1185     def __copy__(self):

/usr/lib/python3.6/typing.py in __subclasscheck__(self, cls)
   1167         if self.__origin__ is not None:
   1168             if sys._getframe(1).f_globals['__name__'] not in ['abc', 'fu
nctools']:
-> 1169                 raise TypeError("Parameterized generics cannot be used w
ith class "
   1170                                 "or instance checks")
   1171             return False

TypeError: Parameterized generics cannot be used with class or instance checks
```

import typeguard

In [36]:

```
from typeguard import typechecked
```

In [37]:

```
def aff(x: int) -> int:
    return x * 2 + 1
```

In [38]:

```
aff(2)
```

Out[38]:

```
5
```

In [39]:

```
aff("2")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-d561ff0ebfbb> in <module>()
----> 1 aff("2")

<ipython-input-37-b3ed04983ac1> in aff(x)
      1 def aff(x: int) -> int:
----> 2     return x * 2 + 1

TypeError: must be str, not int
```

In [40]:

```
taff = typechecked(aff)
```

In [41]:
```
taff(2)
```

Out[41]:
```
5
```

In [42]:
```
taff("2")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-42-860c97b361c3> in <module>()
----> 1 taff("2")

~/.venvs/jupyter/lib/python3.6/site-packages/typeguard.py in wrapper(*args, **kw
args)
    456     def wrapper(*args, **kwargs):
    457         memo = _CallMemo(func, args=args, kwargs=kwargs)
--> 458         check_argument_types(memo)
    459         retval = func(*args, **kwargs)
    460         check_return_type(retval, memo)

~/.venvs/jupyter/lib/python3.6/site-packages/typeguard.py in check_argument_type
s(memo)
    425             value = memo.arguments[argname]
    426             description = 'argument "{}"'.format(argname, memo.func_name
)
--> 427             check_type(description, value, expected_type, memo)
    428 
    429     return True

~/.venvs/jupyter/lib/python3.6/site-packages/typeguard.py in check_type(argname,
 value, expected_type, memo)
    388                 raise TypeError(
    389                     'type of {} must be {}; got {} instead'.
--> 390                     format(argname, qualified_name(expected_type), quali
fied_name(value)))
    391     elif isinstance(expected_type, TypeVar):
    392         # Only happens on < 3.6

TypeError: type of argument "x" must be int; got str instead
```

```
In [43]: taff(2.)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-43-076db6d52d1d> in <module>()
----> 1 taff(2.)

~/.venvs/jupyter/lib/python3.6/site-packages/typeguard.py in wrapper(*args, **kw
args)
    456     def wrapper(*args, **kwargs):
    457         memo = _CallMemo(func, args=args, kwargs=kwargs)
--> 458         check_argument_types(memo)
    459         retval = func(*args, **kwargs)
    460         check_return_type(retval, memo)

~/.venvs/jupyter/lib/python3.6/site-packages/typeguard.py in check_argument_type
s(memo)
    425             value = memo.arguments[argname]
    426             description = 'argument "{}"'.format(argname, memo.func_name
)
--> 427             check_type(description, value, expected_type, memo)
    428 
    429     return True

~/.venvs/jupyter/lib/python3.6/site-packages/typeguard.py in check_type(argname,
 value, expected_type, memo)
    388                 raise TypeError(
    389                     'type of {} must be {}; got {} instead'.
--> 390                     format(argname, qualified_name(expected_type), quali
fied_name(value)))
    391     elif isinstance(expected_type, TypeVar):
    392         # Only happens on < 3.6

TypeError: type of argument "x" must be int; got float instead
```

In [44]:

```
from numbers import Number  # PEP 3141
@typechecked
def taff(x: Number) -> Number:
    return x * 2 + 1
```

```
In [45]: taff(1), taff(1.), taff(1j)
```

```
Out[45]: (3, 3.0, (1+2j))
```

```
In [46]: print("** without type checking **")
         %timeit aff(2)
         print("** with type checking **")
         %timeit taff(2)
```

```
** without type checking **
92.5 ns ± 2.53 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
** with type checking **
31.5 µs ± 103 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Un peu plus loin

In [47]:

```
@typechecked
def index(x: Members["__getitem__"]) -> None:
    x[1]
```

In [48]:

```
index([1,2])
```

In [ ]:

```
@typechecked
def pouce(x: Members['__getitem__']) -> None:
    return x[0]
pouce([0])
```

In [ ]:

```
from typing import List, TypeVar; T = TypeVar('T')

@typechecked
def majeur(x: List[T]) -> T:
    return x[2]
```

```
In [ ]: majeur([1,2,3])
```

```
In [ ]: majeur([1, "1", 1])
```

In [ ]:

```
@typechecked
def step(x : List[T]) -> List[T]:
    return x + [x[-2] + x[-1]]
```

In [ ]:

```
step([1, 2])
```

In [ ]:

```
from functools import reduce
reduce(lambda x, _: step(x), range(10), [0, 1])
```

In [ ]:

```
from typing import Tuple
@typechecked
def step(x : Tuple) -> Tuple:
    return x + (x[-2] + x[-1],)
step((1,2))
```

In [ ]:

```
@typechecked
def step(x : Tuple) -> Tuple:
    return x + (x[-2] + x[-1],)
step((1,2))
```

MyPy

In [ ]:

```
from mypy.api import run as mypy_runner
def mypy(*args):
    print( mypy_runner(args)[0])
```

In [ ]:

```
%%file pyconfr2017/mypy0.py
from typing import Tuple

def step(x : Tuple) -> Tuple:
    return x + (x[-2] + x[-1],)
step([1, 2])
```

In [ ]:

```
mypy("pyconfr2017/mypy0.py")
```

In [ ]:

```
%%file pyconfr2017/mypy1.py
from typing import Tuple

def step(x : Tuple) -> Tuple:
    return x + (x[-2] + x[-1],)
step((1, 2))
```

In [ ]:

```
mypy("pyconfr2017/mypy1.py")
```

In [ ]:

```
%%file pyconfr2017/mypy2.py
from typing import Tuple, Any

def step(x : Tuple[int, ...]) -> Tuple[int, ...]:
    return x + (x[-2] + x[-1],)
step((1, 2))
```

In [ ]:

```
mypy("pyconfr2017/mypy2.py")
```

Python n’est il pas Dynamique?

In [ ]:

```
float_mode = True
scalar = float if float_mode else int

@typechecked
def div(n: scalar):
    return 2 / n

div(scalar(3))
```

In [ ]:

```
%%file pyconfr2017/mypy3.py
float_mode = True
scalar_type = float if float_mode else int

def div(n: scalar_type):
    return 2 / n

div(scalar_type(3))
```

In [ ]:

```
mypy("pyconfr2017/mypy3.py")
```

Le coup fatal

In [ ]:

```
import numpy
help(numpy.sum)
```

Bref, l’interpreteur Python, quel branquignol ?

- Python a été conçu comme un langage de glue
- Dynamisme au cœur du langage
- Utilisons le pour faire ce pour quoi il a été conçu !

Bonus

- Pour toi, ML n'est **pas** une catégorie de langage ?
- Tu aimes la recherche et l'ingénierie ?
- Ça recrute à l'École de Managment de Lyon sur un poste de *research engineer* ½ R&D ½ Prof