Python

OOP

Giovanni Squillero
giovanni.squillero@polito.it

Python OOP

https://github.com/squillero/python-lectures/

version 1.0

but
first

Python — OOP

User Modules

User modules

- A Python file is a "module" and can be imported

file_a.py > ...
```
1 def foo(x):
2     print(f"FileA's foo({x})!")
```

file_b.py
```
1 import file_a
2
3 file_a.foo(23)
```

- When a file is imported, it is evaluated by the interpreter
  - All statements are executed
  - The **`__name__`** is set to the actual name of the file and not "__main__"

User modules

- A directory is a "module" and can be imported
- If the directory contains the file **`__init__.py`**, it is automatically read and evaluated by the interpreter
  - Other files may be imported using **`from pkg import foo`**

my_module > file_a.py > ...
```
1 def foo(x):
2     print(f"my_module's foo({x})!")
```

file_b.py
```
1 from my_module import file_a
2
3 file_a.foo(23)
```

  - The files may also be imported writing appropriate **`import`** instructions in **`__init__.py`**

User modules

- Several alternatives, with obscure, yet important differences

```
import foo                  # foo.py is a file
import mymod                # mymod is a directory containing __init__.py
from mymod import bar       # mymod/bar.py is a file
from mymod.bar import quz   # quz is a function in mymod/bar.py
```

✓ 0.4s Python

- … and even more alternatives, but **never** import *!

```
from math import *
```

✓ 0.3s Python

… also rember that names starting with **_underscore** are not imported

Docstrings in user modules

- Docstrings can be specified as the first statement in files (e.g., **`__init__.py`**)

file_b.py

```
1  import my_module
2
3
```

"my_module" is not accessed Pylance

(module) my_module

Quite a nice module!

Quick Fix... (Ctrl+.)

my_module > __init__.py

```
1  """
2  Quite a nice module!
3  """
```

file_b.py

```
1  from my_module import file_a
2
3
```

"file_a" is not accessed Pylance

(module) file_a

File A's functions are here

Quick Fix... (Ctrl+.)

my_module > file_a.py > ...

```
1  """File A's functions are here"""
```

Python

OOP

Giovanni Squillero
giovanni.squillero@polito.it

Python — OOP

What is OOP?

What is OOP?

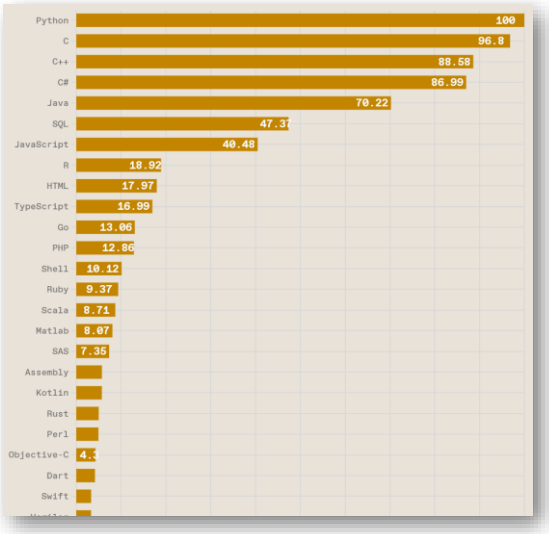
- Object-oriented programming (OOP) is a way of writing computer programs using the idea of "objects" to represent data and methods
- OOP provide a programming solution through interacting entities, usually (but not necessarily) based on the system's "*natural*" structure
- *Classes* are the blueprints (types) of *objects*
- Does OOP require *inheritance*? *type hierarchy*? *subclassing*?

- Encapsulate state and isolate objects
- Decoupling objects from each other
- Late binding

OOP Origin

- **Simula 1** and **Simula 67**
  - Developed in 1960s the by Ole-Johan Dahl and Kristen Nygaard "A superset of ALGOL 60"
- **C++**
  - Developed in late 1970s by Bjarne Stroustrup "Simula with classes and C syntax"
- **Java**
  - Developed in early 1990s by James Gosling

Top Programming Languages 2022



IEEE SPECTRUM

"Modern" Languages

Go

"Modern" Languages

1995

1972

1985

1991

2000

python

Go

2009

"Modern" Languages

1995

1985

1991

2000

09

OOP Today

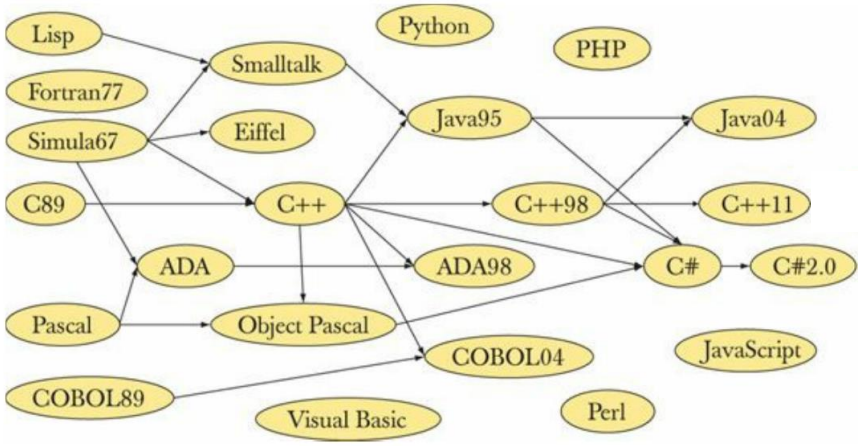


Image from: ***Programming — Principles and Practice Using C++*** **(Second Edition)**

OOP Today

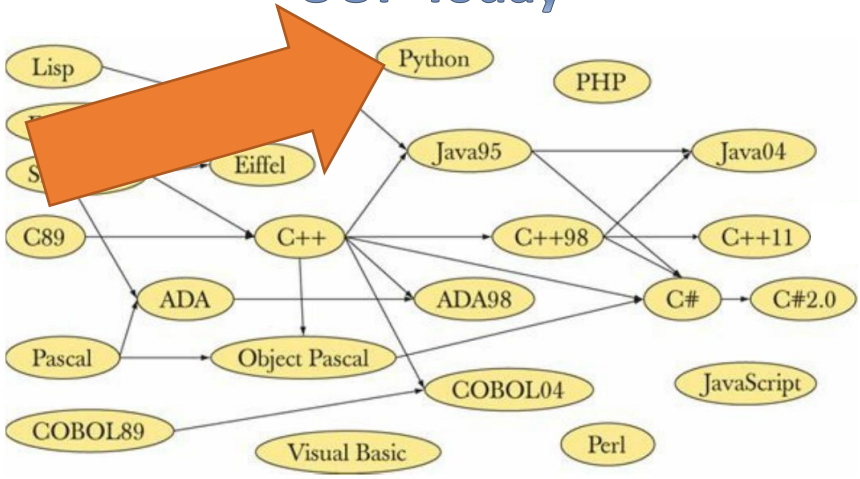
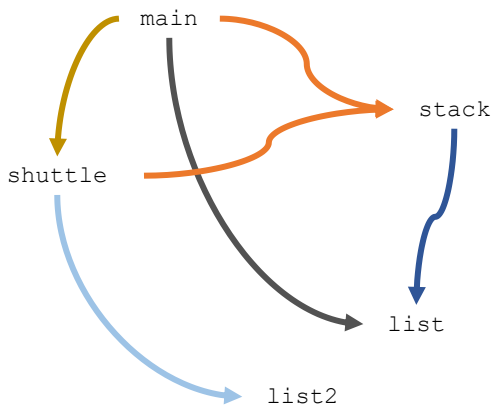


Image from: ***Programming — Principles and Practice Using C++*** **(Second Edition)**

Spaceship Project



Common Needs (since 1960s)

- Maintainability
- Portability (across different machines)
- Compatibility (across versions)
- Understability
- Scalability
- Efficiency
- Predictability

Encapsulation

- Associate code and data into units
- Example (from any basic programming class)
  - Split project on multiple files (modules)
  - push(42)
    the_answer = pop()

Information hiding

- Keep data safe from external interference and misuse
- Example (from C)
  - Multiple files and a sharp use of the "static" keyword
- Caveat
  - Information hiding is not possible in python

Data Abstraction

- Simplified view that only includes interesting features
- ADT
- Example (from basic C)
  - "Gray" pointers as handlers: `FILE *foo`

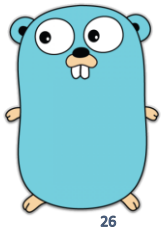
Syntactic Sugar

- `fprintf(foo, "C style\n");`

  vs.

- `foo.printf("Go style\n")`

- Putting the first argument before the function name does not require calling a *method* of an *object*, let alone a language supporting OOP



Genericity

- Different interpretations depending on the data type of parameters
- Allows declaration without specifying their exact data type
- Example:

```
printf("%d %s\n", foo, bar);
```

vs.

```
print(foo, bar);
```

Inheritance

- Hierarchical relationships between data types
- The *child* inherits from one or more *parents*, and may add/change details (but not *remove* anything)
- In OOP: **Class** and **Sub-class**
- Example:
  - **Penguin** inherits from **Bird** (it has wings, etc.)
    it adds **swim()**
    it changes **fly()**

Inheritance

- **Inheritance** vs. **Composition**
  - A data type contains (parts of) other data types
  - The difference can be much subtler than it appears, e.g., “anonymous” struct members is Go

Polymorphism

- "An organism or species can have many different forms or stages"

  — Oracle's "Java Tutorial"

- This principle can also be applied to OOP languages (like Java and C++)

Polymorphism through Hierarchy

- Subclasses can define their own unique behaviors and yet share some of the same functionality of the parent class
- E.g.

```
worm.move()
bird.move()
fish.move()

tectonic_plate.move()
```

Polymorphism through Duck Typing

- An object must implement the methods requested in an *interface*
- E.g. (Go)

```
type Movable interface{
  move() float64
}

bird.move()
fish.move()
tectonic_plate.move()
```

OOP Languages in 2022

- **C++**
  - The fastest OOP language in the west
  - Everything under programmer's control, not always in an easy way
  - Zero-overhead: you don't pay for what you don't use
  - Idioms that fake a garbage collector (e.g., RAII)
  - Significant compile time
  - Huge developing effort

OOP Languages in 2022

- **C++**
- **Java**
  - Garbage-collected, reasonably fast, not predictable
  - Runtime binding
  - Focus on Software Engineering
  - Perfect for large projects under a big boss and with unreliable co-workers

OOP Languages in 2022

- **C++**
- **Java**
- **Go**
  - Not really a full-fledged OOP language
  - Fast compile time
  - Great for parallel applications

OOP Languages in 2022

- **C++**
- **Java**
- **Go**
- **Python**
  - Full-fledged OOP language
  - Fast compile time
  - Terrible for parallel applications, great for high-level programming

OOP Languages in 2022

- **C++**
- **Java**
- **Go**
- **Python**
- **Java Script**
  - ?

Python — OOP

OOP in Python

OOP in Python

Python » English 3.10.7 3.10.7 Documentation » The Python Language Reference » 3. Data model

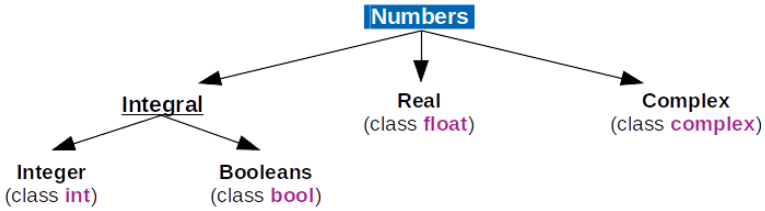
3. Data model

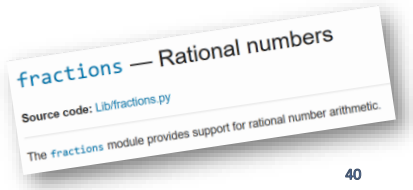
3.1. Objects, values and types

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.)

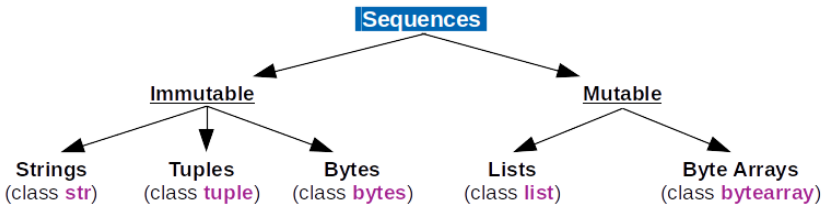
Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. The '`is`' operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

OOP in Python

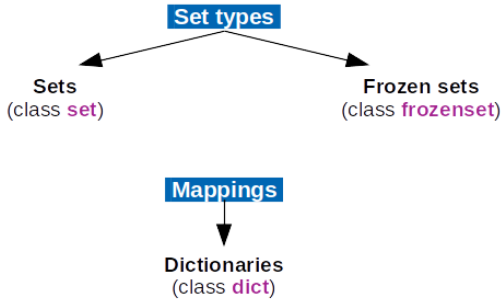




OOP in Python



OOP in Python



OOP in Python

None
(class NoneType)

Callable
< Functions, Methods, Classes >

Modules

Numbers

```
import numbers
import fractions
import math, cmath

def foo(n):
    print(f"{n}:", end=' ')
    if isinstance(n, numbers.Number): print('Number', end=' ')
    if isinstance(n, numbers.Integral): print('Integral', end=' ')
    if isinstance(n, numbers.Rational): print('Rational', end=' ')
    if isinstance(n, numbers.Real): print('Real', end=' ')
    if isinstance(n, numbers.Complex): print('Complex', end=' ')
    print()

foo(5/2)
foo(fractions.Fraction(10, 4))
foo(math.sqrt(2))
foo(cmath.sqrt(-9))
foo("Bob Rock")
```

✓ 0.5s

```
2.5: Number Real Complex
5/2: Number Rational Real Complex
1.4142135623730951: Number Real Complex
3j: Number Complex
Bob Rock:
```


Python — OOP

Custom Class

Beers

```
class Beer:
    pass

class Ale(Beer):
    pass

class Lager(Beer):
    pass

guinness = Ale()

repr(guinness)
```

0.5s Python

```
'<__main__.Ale object at 0x0000024EEDBF9CC0>'
```

```
isinstance(guinness, Beer), isinstance(guinness, Ale), isinstance(guinness, Lager)
```

0.4s

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

Names (TL;DR)

- `module_name`, `package_name`, `ClassName`, `method_name`, `ExceptionName`, `function_name`, `GLOBAL_CONSTANT_NAME`, `global_var_name`, `instance_var_name`, `function_parameter_name`, `local_var_name`.

Encapsulation

```
pilsner_urquell = Lager()
pilsner_urquell.country = 'Czech Republic'

heater_allen_pils = Lager()
heater_allen_pils.country = 'United States'
heater_allen_pils.state = 'Oregon'
```

0.3s

`dir(pilsner_urquell)` 0.4s

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'country']
```

`dir(heater_allen_pils)` 0.3s

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'country',
 'state']
```

__dunder__

```
pilsner_urquell.__dict__
{'country': 'Czech Republic'}

heater_allen_pils.__dict__
{'country': 'United States', 'state': 'Oregon'}
```

```
dir(heater_allen_pils)
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'country',
 'state']
```


Reasonably well-written Class

"Special" names (such as special methods) are __dunder__

"private" name starts with underscore (a mere convention)

Real Python programmers don't use setters nor getters

```
class Beer:
    """3rd most popular drink in the world after water and tea"""

    def __init__(self, yeast):
        self._yeast = yeast

    @property
    def yeast(self):
        return self._yeast
```

Class comment

Self (kind of "this") is always explicit

```
tuborg = Beer('Saccharomyces carlsbergensis')
tuborg.yeast
'Saccharomyces carlsbergensis'

tuborg.yeast = 'Saccharomyces pastorianus'
AttributeError                            Traceback (most recent call last)
Cell In [64], line 1
----> 1 tuborg.yeast = 'Saccharomyces pastorianus'

AttributeError: can't set attribute 'yeast'
```

Read-only, for a "setter" use:

```
@yeast.setter
def yeast(self, new_yeast):
    self._yeast = new_yeast
```

Slots

- Reserve space for a fixed set of names (may be much faster)

```
class Point:
    """(x, y) Cartesian point"""

    __slots__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y

p1 = Point(2, 3)
p1.foo = 42
```

0.4s Python

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [108], line 11
      8         self.y = y
     10 p1 = Point(2, 3)
---> 11 p1.foo = 42

AttributeError: 'Point' object has no attribute 'foo'
```


staticmethods

- Methods that do not require an *object*, but only make sense inside the *class*…
- *Class* methods
- Similar to *static* methods in C++ or Java
- No *self* as first parameter

```
@staticmethod
def get_args(all_args, tag):
    tag_args = dict()
    for a in all_args:
```

Style

- Start names private or protected within a class with a single underscore: `_foo`
- Avoid name mangling if possible: `__foo`
- Use nouns for properties, full sentences for methods: `foo.factors = bar.factorize(baz.formula)`
- Do not implement getters and setters, use properties instead
- Whether a function does not need parameters consider using a property: `foo.first_bar` instead of `foo.calculate_first_bar()`
- Do not hide complexity: `number.get_prime_factors()`

Name Mangling

- A.k.a., name decoration

```
class Beer:
    def __init__(self):
        self._foo = 23
        self.__bar = 10

x = Beer()
x._foo = 0
x.__bar = 0

x.__dict__
```

0.3s Python

```
{'_foo': 0, '_Beer__bar': 10, '__bar': 0}
```

- Not a real protection, but the closest thing to “private”
- Can be useful when sub-classing

Python Data Model

`x.user(y)` → `TypeX.user(x, y)`

`special(y)` → `TypeY.__special__(y)`

Printing Custom Classes

- `print(x)` → `print(str(x))` → `print(ClassX.__str__(x))`
- `print(x)` vs. `repr(x)`

```
def __repr__(self):
    return f"Beer @ {hex(id(self))}"
```

```
class Beer:
    """3rd most popular drink in the world after water and tea"""

    def __init__(self, type, yeast=None):
        if yeast is None:
            yeast = 'Unknown'
        self._type = type
        self._yeast = yeast

    @property
    def type(self):
        return self._type

    @property
    def yeast(self):
        return self._yeast

    def __str__(self):
        return f"Beer (type={self._type}, yeast={self._yeast})"

funky_buddha = Beer(type='Porter')
print(funky_buddha)
```

```
Beer (type=Porter, yeast=Unknown)
```

Basic Customizations

`x < y → x.__lt__(y) → ClassX.__lt__(x, y)`

- Etc…

Basic Customizations

- `__init__(self[, ...])`
  - Called after the instance has been created
- `__del__(self)`
  - Called when the instance is about to be destroyed

Basic Customizations

- `__repr__(self)`
  - Called by repr() — Official representation of the object (you should not change it)
- `__str__(self)`
  - Called by `str()`, `print()` and `format()` — Computer the informal, nicely printable representation of the object
- `__bytes__(self)`
  - Called by `bytes()` — Compute a byte-string representation of an object
- `__format__(self, format_spec)`
  - Called by `format` — Compute a formatted representation of the object as specified by an f-string (e.g., `f"{obj:x}"`)

Basic Customizations

- `__lt__(self, other)`
  - Called by `x < y`
- `__le__(self, other)`
  - Called by `x <= y`
- `__eq__(self, other)`
  - Called by `x == y`
- `__ne__(self, other)`
  - Called by `x != y`
- `__gt__(self, other)`
  - Called by `x > y`
- `__ge__(self, other)`
  - Called by `x >= y`

Python is smart!

```python
class Point:
    """(x, y) Cartesian point"""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x},{self.y})"

    def __eq__(self, other):
        logging.debug(f"{self}=={other}?")
        return math.isclose(self.x**2 + self.y**2, other.x**2 + other.y**2)

    def __lt__(self, other):
        logging.debug(f"{self}<{other}?")
        return self.x**2 + self.y**2 < other.x**2 + other.y**2

    def __le__(self, other):
        logging.debug(f"{self}<={other}?")
        return self < other or self == other
```

Python

```
p1 = Point(1, 1)
p2 = Point(2, 1)

p1 >= p2
```

✓ 0.4s

```
DEBUG:root:(2,1)<=(1,1)?
DEBUG:root:(2,1)<(1,1)?
DEBUG:root:(2,1)==(1,1)?
False
```


Basic Customizations

- **`__hash__(self)`**
  - Called by **`hash()`** — Compute the hash, i.e., an integer
- **`__bool__(self)`**
  - Called by **`bool()`** — Compute a Boolean value when the object is evaluated in a test. If __bool__ is not defined, **`__len__() != 0`** is used. If neither **`__bool__()`** nor **`__len__()`** are defined, the object is True
- Pippo

Emulating Callable Objects

- `__call__(self, …)`

```
class Functor:
    def __init__(self, offset):
        self._offset = offset
    def __call__(self, x):
        return self._offset + x

foo42 = Functor(42)
foo100 = Functor(100)

foo42(1), foo100(1)
```

✓ 0.4s Python

```
(43, 101)
```


Emulating Container Types

- `__len__(self)` // `__length_hint__(self)`

- `__getitem__(self, key)`
- `__setitem__(self, key, value)`
- `__delitem__(self, key)`

- `__missing__(self, key)`
- `__iter__(self)`
- `__reversed__(self)`
- `__contains__(self, item)`

Emulating Numeric Types

- `__add__(self, other)`
- `__sub__(self, other)`
- `__mul__(self, other)`
- `__matmul__(self, other)`
- `__truediv__(self, other)`
  - `__floordiv__(self, other)`
- `__mod__(self, other)`
- `__divmod__(self, other)`
- `__pow__(self, other[, modulo])`

Emulating Numeric Types

- `__lshift__(self, other)`
- `__rshift__(self, other)`
- `__and__(self, other)`
- `__xor__(self, other)`
- `__or__(self, other)`

Emulating Numeric Types

- Reflected (swapped) operands. Called only if the left operand does not support the corresponding operation and the operands are of different types
  - `__radd__(self, other)`
  - `__rsub__(self, other)`
  - `__rmul__(self, other)`
  - `__rmatmul__(self, other)`
  - `__rtruediv__(self, other)`
  - `__rfloordiv__(self, other)`
  - `__rmod__(self, other)`
  - `__rdivmod__(self, other)`
  - `__rpow__(self, other[, modulo])`
  - `__rlshift__(self, other)`
  - `__rrshift__(self, other)`
  - `__rand__(self, other)`
  - `__rxor__(self, other)`
  - `__ror__(self, other)`

Emulating Numeric Types

- Augmented arithmetic assignments
  - `__iadd__(self, other)`
  - `__isub__(self, other)`
  - `__imul__(self, other)`
  - `__imatmul__(self, other)`
  - `__itruediv__(self, other)`
  - `__ifloordiv__(self, other)¶`
  - `__imod__(self, other)`
  - `__ipow__(self, other[, modulo])`
  - `__ilshift__(self, other)`
  - `__irshift__(self, other)`
  - `__iand__(self, other)`
  - `__ixor__(self, other)`
  - `__ior__(self, other)`

Emulating Numeric Types

- Called to implement the unary arithmetic operations
  - `__neg__(self)`
  - `__pos__(self)`
  - `__abs__(self)`
  - `__invert__(self)`

Conversions

- `__complex__(self)`
- `__int__(self)`
- `__float__(self)`

- `__round__(self[, ndigits])`
- `__trunc__(self)`
- `__floor__(self)`
- `__ceil__(self)¶`

Context Managers

- I.e., use with the *with* statement
  - `__enter__(self)`
  - `__exit__(self, exc_type, exc_value, traceback)`

Iterators

- (Container) `__iter__()`
  - Return an iterator object.
- (Iterator) `__iter__()`
  - Return the iterator object itself.
- (Iterator) `__next__()`
  - Return the next item from the iterator.
  - If there are no further items, raise the StopIteration exception.

Generators

- A convenient way to implement the iterator protocol!

```
def my_generator(n):
    for p in range(n):
        yield 2**p

[x for x in my_generator(4)]
```

✓ 0.2s

```
[1, 2, 4, 8]
```


Quite Easy Approach

- Even easier…

```
class Test:
    def __iter__(self):
        return (x for x in range(10))
```

✓ 0.3s

Class `slice`

- A set of indices
- The *start* and *step* arguments default to None.
- Slice objects have read-only data attributes start, stop, and step which merely return the argument values (or their default).
- An alternate version that returns an iterator:
  - **`itertools.islice(iterable, stop)`**
  - **`itertools.islice(iterable, start, stop[, step])`**

Decorators

```
def check_performance(func):
    def wrapper_func(*args, **kwargs):
        start_time = perf_counter()
        func(*args, **kwargs)
        finish_time = perf_counter()
        ta = [str(_) for _ in args]
        for k, v in kwargs.items():
            ta.append(f"{k}={v}")
        print(f"{func.__name__}({', '.join(ta)}): {finish_time - start_time:.6f}")

    return wrapper_func
```

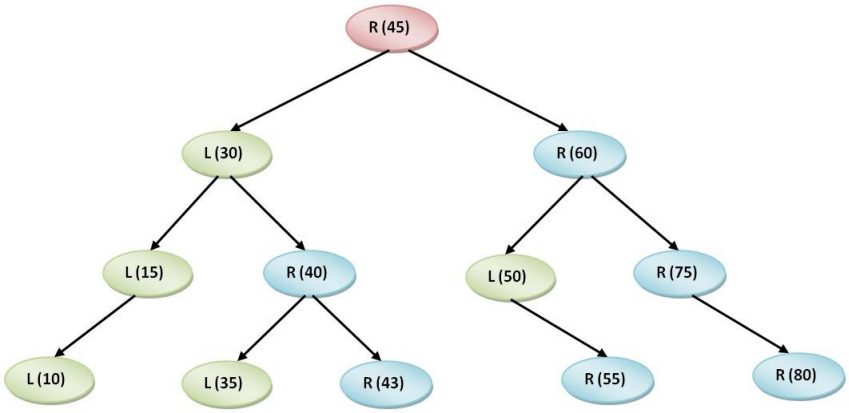
```
@check_performance
def my_func(my_arg):
    print("Hoy!")

my_func(my_arg=42)
```

✓ 0.3s

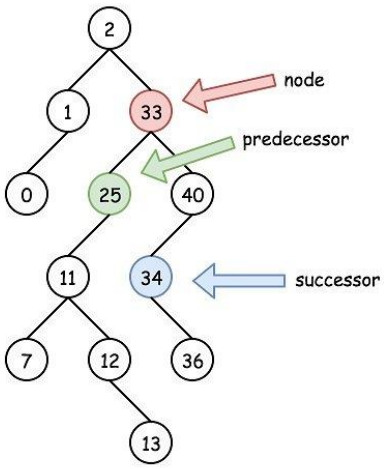
```
Hoy!
my_func(my_arg=42): 0.000022
```

Wrap-up Excercise

- Create a BST class that supports slicing



Predecessor/Successor



predecessor =
one step left and then right till you can

successor =
one step right and then left till you can


Google
squillero
squillero
squillero polito
squillero giovanni
squillero github
Google 搜索
手气不错
giovanni.squillero@polito.it