Freie Universität Berlin

Computational Sciences Projektseminar

Object-oriented programming, design patterns, and anti patterns

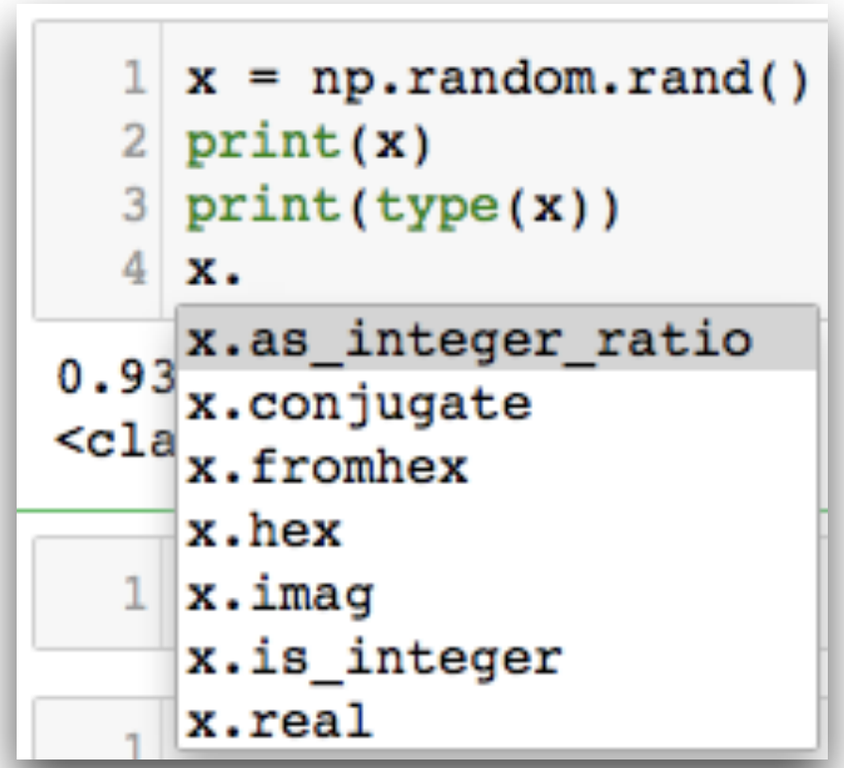
Object-oriented programming (OOP)

```
x = np.random.rand()
print(x)              # 0.9389428619918472
print(type(x))        # <class 'float'>


print(x.real)         # 0.9389428619918472
print(x.imag)         # 0.0


print(x.is_integer)
# <built-in method is_integer of float object at 0x1117916a8>


print(x.is_integer())
# False
```



Attribute

Method

Object-oriented programming (OOP)

An object

- is an instance of a class
- contains data (attributes) and functionality (methods)

A class

- is a blueprint for objects
- can inherit from one or more parent class(es)

Example: a class for an x/y-point

```
class Point(object):
    def __init__(self, x=0.0, y=0.0):
        self.x = x
        self.y = y
    def norm(self):
        return np.sqrt(self.x**2 + self.y**2)
```

```
p = Point(3, 4)
print(p.x, p.y)   # 3 4
print(p.norm())   # 5.0
print(p)          # <__main__.Point object at 0x111999a90>
```

Example: a class for an x/y-point

```
class Point(object):
    def __init__(self, x=0.0, y=0.0):
        self.x = x
        self.y = y
    def norm(self):
        return np.sqrt(self.x**2 + self.y**2)
    def __repr__(self):
        return 'Point(%s, %s)' % (self.x, self.y)

p = Point(3, 4)
print(p.x, p.y)  # 3 4
print(p.norm())  # 5.0
print(p)         # Point(3, 4)
```

Example: a class for an x/y-point

```
class Point(object):
    def __init__(self, x=0.0, y=0.0):
        self.x = x
        self.y = y
    def norm(self):
        return np.sqrt(self.x**2 + self.y**2)
    def __repr__(self):
        return 'Point(%s, %s)' % (self.x, self.y)

a, b = Point(), Point(3, 4)
print(a, b)   # Point(0.0, 0.0) Point(3, 4)
print(a + b)

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Example: a class for an x/y-point

```
class Point(object):
    def __init__(self, x=0.0, y=0.0):
        self.x = x
        self.y = y
    def norm(self):
        return np.sqrt(self.x**2 + self.y**2)
    def __repr__(self):
        return 'Point(%s, %s)' % (self.x, self.y)
    def __add__(self, p):
        return Point(self.x + p.x, self.y + p.y)

a, b = Point(), Point(3, 4)
print(a, b)   # Point(0.0, 0.0) Point(3, 4)
print(a + b)  # Point(3, 4)
```

Example: subclassing

```
class PointCharge(Point):
    def __init__(self, x=0, y=0, q=0):
        super(PointCharge, self).__init__(x, y)
        self.q = q
    def __repr__(self):
        string = 'Charge q=%s at ' % self.q
        string += super(PointCharge, self).__repr__()
        return string

print(PointCharge(q=1))  # Charge q=1 at Point(0, 0)
print(PointCharge(q=1) + PointCharge(q=-1))
# Point(0, 0)
```

Part II

Design patterns and anti patterns

Design patterns: lazy initialisation

delay expensive calculations until they are needed

```
class DataIntense(object):
    def __init__(self):
        self._data = None
    @property
    def data(self):
        if self._data is None:
            self._data = expensive_function()
        return self._data

data_intense = DataIntense() # not expensive
print(data_intense.data)     # but this is
print(data_intense.data)     # and this is not
```

Design patterns: templates

create a template with the basic skeleton and subclass

```
class Solver(object):
    def __init__(self, **kwargs):
        # whatever
    def check_input(self, **kwargs):
        # more code
```

```
class JacobiSolver(Solver):
    def __init__(self, **kwargs):
        super(JacobiSolver, self).__init__(**kwargs)
    def solve(self, data):
        self.check_input(data)
        # solve via Jacobi
```

Design patterns: decorators

encapsulate a class/function to change its behaviour

```
from time import time, sleep

def benchmark(func):
    def wrapper(*args, **kwargs):
        start = time()
        res = func(*args, **kwargs)
        print('run time: %.2f s' % (time() - start))
        return res
    return wrapper

@benchmark
def some_func(delay=1):
    sleep(delay)

some_func(1.23) # 'run time: 1.23 s'
```

Examples of anti patterns (social)

Analysis paralysis

overanalysing/-thinking the problem at hand and, thus, never taking any action or reaching a decision

Bicycle shed

spending disproportionate effort on trivial issues

Ninty-ninty rule

"The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time."

— Tom Cargill, Bell Labs

Examples of anti patterns (software design and OOP)

Big ball of mud / spaghetti code

a code without perceivable structure/architecture;
unfortunately very common among untrained programmers

Object orgy

no proper encapsulating; direct access to internals

Sequential coupling

the need to call a class' methods in a particular order

Examples of anti patterns (programming)

Cargo cult programming

the use of patterns/methods without a proper understanding

Magic numbers

use of unexplained numbers in algorithms

Repeating yourself

writing the same code again and again and again…

Examples of anti patterns (methodological)

Programming by permutation

changing code by trial and error until it works
(but nobody knows why)

Reinventing the square wheel

why adopting an existing solution if we can craft one ourselves
which is inferior…

Tester driven development

let the testers find the conditions under which your code can run