2023 12December 12

December 21, 2023

1 Agenda

1. Objects
   - Object system in Python
   - Attributes (ICPO)
   - Magic methods
   - Inheritance
   - `__del__`
   - Metaclasses
2. Iterator protocol
   - Protocol
   - Adding iteration to our classes
   - Generator functions
   - Generator comprehensions
3. Function annotations + Mypy
4. Concurrency
   - Threads
   - Processes
   - `asyncio`
5. Profiling
6. Data science / analytics
   - NumPy
   - Pandas
   - Matplotlib
7. Pytest

2 What is Pythonic?

```
[ ]: import this
```

3 Everything is an object!

That means, everything in Python has three things:

- ID
- type
- Attributes

[ ]:

```python
x = 'abcd'
id(x)
```

[ ]:

```python
y = 'efgh'
id(y)
```

[ ]:

```python
# I can check if x and y refer to the same object

id(x) == id(y)
```

[ ]:

```python
x = [10, 20, 30]
y = [10, 20, 30]

id(x) == id(y)
```

[ ]:

```python
# do they have the same values?
x == y
```

[ ]:

```python
# if I want to check whether two objects are the same, we can use "is"

x = [10, 20, 30]
y = [10, 20, 30]

x is y    # id(x) == id(y)
```

[ ]:

```python
x = 10
y = 10

x == y
```

[ ]:

```python
x is y
```

[ ]:

```python
x = 1000
y = 1000

x == y
```

[ ]:

```python
x is y
```

Python defines all ints from -5 until 256 when the language starts up. If we use one of these ints, then we get the pre-defined object.

[ ]:

```python
def myfunc():
    x = 1000
    y = 1000
    print(x == y)
    print(x is y)
```

[ ]:

```
myfunc()
```

[ ]:

```
myfunc.__code__.co_code
```

[ ]:

```
import dis    # disassembler
dis.dis(myfunc)
```

[ ]:

```
myfunc.__code__.co_consts
```

[ ]:

```
x = 1000; y = 1000

x is y
```

[ ]:

```
def myfunc():
    x = 1000
    y = x
    print(x == y)
    print(x is y)
```

[ ]:

```
dis.dis(myfunc)
```

[ ]:

```
x = 100
y = x

x = 200
y
```

[ ]:

```
x = 'abcd'
y = 'abcd'

x == y
```

[ ]:

```
x is y
```

[ ]:

```
x = 'a.b'
y = 'a.b'

x == y
```

[ ]:

```
x is y
```

[ ]:

```
x = 'abcd' * 100_000
y = 'abcd' * 100_000

x == y
```

[ ]:

```
x is y
```

[ ]:
```
z = 'xyz'    # z is turned into a string, and used as a dict key!
```

[ ]:
```
globals()
```

[ ]:
```
globals()['z']
```

[ ]:
```
globals()['z'] = 1234
z
```

[ ]:
```
def myfunc():
    x = 1234
```

[ ]:
```
myfunc.__code__.co_varnames
```

[ ]:
```
x = 100     # global x

def myfunc():
    x += 1    # x = x + 1
    print(x)

myfunc()
```

[ ]:
```
def myfunc():
    x = 100
    y = [10, 20, 30]
    print(locals())

myfunc()
```

[ ]:
```
d = {'a':10, 'b':20, 'c':30}

def myfunc(a_dict):
    a_dict['a'] = 12345

print(d)
myfunc(d)
print(d)
```

Visualized in Python Tutor at:

https://pythontutor.com/render.html#code=d%20%3D%20%7B’a’%3A10,%20’b’%3A20,%20’c’%3A30%7D%0A%
frontend.js&py=311&rawInputLstJSON=%5B%5D&textReferences=false

[ ]:
```
# every object has a type

x = 100
type(x)
```

[ ]:

```
s = 'abcd'
type(s)
```

[ ]:

```
mylist = [10, 20, 30]
type(mylist)
```

[ ]:

```
type(int)
```

[ ]:

```
type(str)
```

[ ]:

```
type(list)
```

[ ]:

```
# every class in Python is an instance of type!

type(type)
```

[ ]:

```
x = 10
y = '20'

x + y
```

[ ]:

```
# to check type in code, use isinstance
```

[ ]:

```
# attributes
# every object has attributes, a private dict that we access via "."
# we can add just about any attribute to just about any object we want

# dir(x) returns a list of strings, the attributes on x

s = 'abcd'
dir(s)
```

4 Let's define a class!

[ ]:

```
class MyClass:
    def __init__(self):    # "dunder-init" -- self is the instance we've␣
 ↪created!
        self.x = 100       # add the attribute x, value 100
        self.y = 'abcd'    # add this attribute y, value 'abcd'

m = MyClass()    # here, we call (indirectly) to `__new__`
print(m.x)
print(m.y)
```

5 What's really happening?

- The constructor method in Python `__new__`. This method gets three attributes: The class, `*args`, `**kwargs`. It then creates a new instance of our class. Let's say that it sticks the new instance in a variable called `o`.
- `__new__` then calls `__init__`, passing it `o` (which in `__init__` is `self`), and adds `*args` and `**kwargs`. The job of `__init__` is to add attributes to the new object.
- When `__init__` returns, the object has changed. `__new__` returns that (changed) object to the caller.

[ ]:

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

p = Person('Reuven')
print(vars(p))    # vars(p) returns a dict of all attributes on the instance
```

[ ]:

```
# what if I want to get the name, or set it?

class Person:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_name(self, new_name):
        self.name = new_name


p = Person('Reuven')
print(p.get_name())
p.set_name('my new name')
print(p.get_name())
```

[ ]:

```
# In Python, we normally don't write getters and setters
# we normally retrieve and set attributes *directly*

# there's no need, everything is public
# you could use a property, which looks like data but acts like a function/
 ↪method
```

[ ]:

```
# here's my non-getter/setter code:

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

```python
p = Person('Reuven')
print(p.name)
p.name = 'my new name'
print(p.name)
```

[ ]:

```python
class Person:
    def __init__(self, name):
        self.name = name

p = Person('Reuven')
print(vars(Person))
```

[ ]:

```python
a = 1
vars(a)
```

[ ]:

```python
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    def fullname(self):
        return f'{self.first} {self.last}'

p = Person('Reuven', 'Lerner')
print(p.fullname())
```

[ ]:

```python
s = 'abcd'
s.upper()    # --> str.upper(s)
```

[ ]:

```python
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    def fullname(self):
        return f'{self.first} {self.last}'

    @staticmethod
    def hello(name):
        return f'Hello, {name}!'

p = Person('Reuven', 'Lerner')
print(p.fullname())
```

[ ]:

```python
Person.hello('whoever')
```

[ ]:

```
p.hello('world')
```

6 Exercise: Ice cream

1. Define a `Scoop` class, whose instances have a single `flavor` attribute.
2. Define three instances of `Scoop`, each with a different flavor. Print the flavor from each one.
3. Define a `Bowl` class, whose instances have a single `scoops` attribute – a list that will contain instances of `Scoop`. We'll also want two other methods:
   - `add_scoops` – a method that takes any number of `Scoop` objects, and adds them to our `scoops` attribute
   - `flavors` – method that takes no arguments, but that returns a list of strings with the flavors in that bowl.

Example:

```
s1 = Scoop('chocolate')
s2 = Scoop('vanilla')
s3 = Scoop('coffee')

b = Bowl()
b.add_scoops(s1, s2)
b.add_scoops(s3)
print(b.flavors())  ['chocolate', 'vanilla', 'coffee']
```

[ ]:

```
class Scoop:
    def __init__(self, flavor):
        self.flavor = flavor

s1 = Scoop('chocolate')
s2 = Scoop('vanilla')
s3 = Scoop('coffee')

for one_scoop in [s1, s2, s3]:
    print(one_scoop.flavor)

class Bowl:
    def __init__(self):
        self.scoops = []

    def add_scoops(self, *new_scoops):
        self.scoops += new_scoop

        for one_scoop in new_scoops:
            self.scoops.append(one_scoop)

# b = Bowl()
# b.add_scoops(s1, s2)
# b.add_scoops(s3)
# print(b.flavors())  ['chocolate', 'vanilla', 'coffee']
```

[ ]:
```
mylist = [10, 20, 30]

mylist += 'abcd'

mylist
```

[ ]:
```
# append takes one argument, and adds it to the end of the list
mylist = [10, 20, 30]
mylist.append('abcd')
mylist.append([100, 200, 300])
mylist
```

[ ]:
```
# extend takes one argument, and runs a for loop on it and adds
mylist = [10, 20, 30]
mylist.extend('abcd')
mylist.extend([100, 200, [300, 400, 500]])
mylist
```

[ ]:
```
self.scoops[:] = [*self.scoops, *new_scoops]
```

[ ]:
```
mylist1 = [10, 20, 30]
mylist2 = [100, 200, 300]

[mylist1, mylist2]
```

[ ]:
```
[*mylist1, *mylist2]
```

[ ]:
```
mylist = [10, 20, 30, 40, 50]
x = mylist

mylist = [100, 200, 300]    # mylist, the variable, refers to a new list
x
```

[ ]:
```
mylist = [10, 20, 30, 40, 50]
x = mylist

mylist[:] = [100, 200, 300]  # replace the contents of mylist (and x) with␣
 ↪other stuff
x
```

[ ]:
```
%%timeit

mylist = []
mylist.append(100)
mylist.append(200)
mylist.append(300)
```

[ ]:

```
%%timeit

mylist = []
mylist.extend([100, 200, 300])
```

[ ]:

```
%%timeit

mylist = []
mylist[:] = [100, 200, 300]
```

[ ]:

```
class Scoop:
    def __init__(self, flavor):
        self.flavor = flavor

s1 = Scoop('chocolate')
s2 = Scoop('vanilla')
s3 = Scoop('coffee')

for one_scoop in [s1, s2, s3]:
    print(one_scoop.flavor)

class Bowl:
    def __init__(self):
        self.scoops = []

    def add_scoops(self, *new_scoops):
        for one_scoop in new_scoops:
            self.scoops.append(one_scoop)

    def flavors(self):

# I have: a list of scoops
# I want: a list of strings (flavors)
# I can map from the first to the second with the .flavor attribute

        return [one_scoop.flavor
               for one_scoop in self.scoops]

#        output = []
#         for one_scoop in self.scoops:
#             output.append(one_scoop.flavor)

#         return output

b = Bowl()
```

```
b.add_scoops(s1, s2)
b.add_scoops(s3)
print(b.flavors()) #  ['chocolate', 'vanilla', 'coffee']
```

[ ]:

```
mylist = [10, 20, 30]
t = (100, 200, 300)

mylist.extend(t)
```

[ ]:

```
mylist
```

7 Next up

- Attributes - instance + class + ICPO

Resume at :45

[ ]:

```
population = 0

class Person:
    def __init__(self, name):
        self.name = name
        population += 1

    def greet(self):
        return f'Hello, {self.name}!'

print(f'Before, population = {population}')
p1 = Person('name1')
p2 = Person('name2')
print(f'After, population = {population}')

print(p1.greet())
print(p2.greet())
```

[ ]:

```
# let's declare population to be *only* global

population = 0

class Person:
    def __init__(self, name):
        global population    # meaning: don't record population as local during␣
 ↪compilation
        self.name = name
        population += 1

    def greet(self):
        return f'Hello, {self.name}!'

print(f'Before, population = {population}')
p1 = Person('name1')
p2 = Person('name2')
print(f'After, population = {population}')

print(p1.greet())
print(p2.greet())
```

[ ]:

```
# instead of a global variable, let's make population an attribute on the class

class Person:
    def __init__(self, name):
        self.name = name
        Person.population += 1

    def greet(self):
        return f'Hello, {self.name}!'

Person.population = 0

print(f'Before, population = {Person.population}')
p1 = Person('name1')
p2 = Person('name2')
print(f'After, population = {Person.population}')

print(p1.greet())
print(p2.greet())
```

[ ]:

```
print('A')
class MyClass:
    print('B')
    def __init__(self):
        print('C')
        self.x = 100
    print('D')
print('E')

m1 = MyClass()
m2 = MyClass()
```

When I use `def`, Python does two things: - Creates a function object - Assigns that function to a variable

Moreover, the body of the function isn't run at definition time.

However, when we define a class, the body of the class is executed. Any variable we define inside of the class is not really a variable, but rather an attribute on the class.

[ ]:

```
MyClass.__init__
```

[ ]:

```
# Let's use a cleaner syntax to add the population attribute

class Person:
    population = 0   # this defines Person.population = 0

    def __init__(self, name):
        self.name = name
        Person.population += 1

    def greet(self):
        return f'Hello, {self.name}!'

print(f'Before, population = {Person.population}')
p1 = Person('name1')
p2 = Person('name2')
print(f'After, Person.population = {Person.population}') # Person has population? yes, 2
print(f'After, p1.population = {p1.population}') # p1 has population? No. Person? Yes, 2
print(f'After, p2.population = {p2.population}') # p2 (same thing)

print(p1.greet()) # p1 has greet? No. Person has greet? Yes
print(p2.greet()) # p2 has greet? No. person has greet? Yes.
```

8 ICPO – attribute resolution order

When we ask Python for an attribute, it tries to find the attribute on several objects:

- `I` - instance that we specified.
- `C` - If it cannot find the attribute on the instance, it looks on the class
- `P` - If it cannot find on the class, it looks on the parent
- `O` - If it cannot find on the parent, it looks on `object`

[ ]:

```
# who does Person inherit from?

Person.__bases__
```

[ ]:

```
# what if we change Person.population += 1 to self.population += 1?

class Person:
    population = 0   # this defines Person.population = 0

    def __init__(self, name):
        self.name = name
        Person.population += 1
    def greet(self):
        return f'Hello, {self.name}!'

print(f'Before, population = {Person.population}')
p1 = Person('name1')
p2 = Person('name2')
print(f'After, Person.population = {Person.population}')
print(f'After, p1.population = {p1.population}')
print(f'After, p2.population = {p2.population}')

print(p1.greet())
print(p2.greet())
```

9 Why class attributes?

- Methods. All methods are class attributes.
- Constants. We can have (sorta kinda) constants by assigning them on the class, assuming that we won't change them. If we have a value that we'll use everywhere in the class, we can put it here.
- Shared resources. Something that's shared among all instances can go in the class as an attribute.

[ ]:

```
# one-time only initialization of class attribute from an instance

class Person:
    population = None   # initial value

    def __init__(self, name):
        self.name = name

        if Person.population is None:   # initialize Person.population once
            Person.population = 0

        Person.population += 1          # always run this

    def greet(self):
        return f'Hello, {self.name}!'

print(f'Before, population = {Person.population}')
p1 = Person('name1')
p2 = Person('name2')
print(f'After, Person.population = {Person.population}')
print(f'After, p1.population = {p1.population}')
print(f'After, p2.population = {p2.population}')

print(p1.greet())
print(p2.greet())
```

[ ]:

```
# generally, if/else is done like this:

x = 10

if x % 2:
    result = 'odd'
else:
    result = 'even'

print(result)
```

[ ]:

```
# there is another way...

x = 10

result = 'odd' if x % 2 else 'even'

print(result)
```

10 Exercise: Limited bowls

1. Let's limit the number of scoops you can put in a bowl to 3.
2. Modify `add_scoops` to do this; any scoop beyond the third is ignored.

[ ]:

```
x = 10

x++
```

[ ]:

```
(+(+x))    # unary plus
```

[ ]:

```
x
```

[ ]:

```
class Scoop:
    def __init__(self, flavor):
        self.flavor = flavor

class Bowl:
    MAX_SCOOPS = 3

    def __init__(self):
        self.scoops = []

    def add_scoops(self, *new_scoops):
        for one_scoop in new_scoops:
            if len(self.scoops) < Bowl.MAX_SCOOPS:
                self.scoops.append(one_scoop)

    def flavors(self):

        return [one_scoop.flavor
               for one_scoop in self.scoops]

s1 = Scoop('chocolate')
s2 = Scoop('vanilla')
s3 = Scoop('coffee')
s4 = Scoop('flavor 4')
s5 = Scoop('flavor 5')

b = Bowl()
b.add_scoops(s1, s2)
b.add_scoops(s3)
b.add_scoops(s4, s5)
print(b.flavors())
```

[ ]:

```
# Inheritance

class Person:

    def __init__(self, name):
        self.name = name

    def greet(self):
        return f'Hello, {self.name}!'

p1 = Person('name1')
p2 = Person('name2')

print(p1.greet())
print(p2.greet())

class Employee(Person):    # Employee is-a Person, i.e., inherits from Person

    def __init__(self, name, id_number):
        super().__init__(name)        # do what my parent does in __init__...
        self.id_number = id_number    # add my own things

e1 = Employee('emp1', 1)# e1 has __init__? no. Empoyee has __init__? Yes
e2 = Employee('emp2', 2)

print(e1.greet()) # e1 has greet? No. Employee has greet? No. Person has greet?␣
 ↪yes
print(e2.greet())
```

[ ]:

```
Employee.__bases__
```

[ ]:

```
Person.__bases__
```

11 Next time

1. Inheritance
2. Magic methods (`__del__`)
3. Object system
4. Metaclasses
5. Iterators etc.