Info1 Python Tutorial - Part IV

Carol V. Alexandru-Funakoshi

October 17, 2023

1 Quick recap from last week

1.0.1 Ternary conditional operator

[1]:
```
x = -5
x_abs = x if x >= 0 else -x
print(x_abs)
# FYI, Python ships with an abs() function:
print(abs(x))
```

```
5
5
```

[2]:
```
grades = {"Alice": 5.5, "Bob": 3.5, "Linda": 6.0}
{name: grade + 0.25 if 4 <= grade <= 5.75 else grade for name, grade in grades.
 ↪items()}
```

[2]:
```
{'Alice': 5.75, 'Bob': 3.5, 'Linda': 6.0}
```

1.0.2 `None` and `NoneType`

[3]:
```
print(type(123))
print(type(True))
print(type(None))
```

```
<class 'int'>
<class 'bool'>
<class 'NoneType'>
```

Functions in Python implicitly return `None` unless they explicitely return something else

[4]:
```
def printing(x):
    print(x + 1)        # this prints 11; the function eventually returns None␣
 ↪after that
printing(10)            # this evaluates to: None, so Jupyter doesn't print␣
 ↪anything
```

```
11
```

The `print` function returns `None`

[5]:
```
def x():
    return print("Hello")
result = x()
print(result)
print(type(result))
```

```
Hello
None
<class 'NoneType'>
```

1.0.3 Early return

[6]:
```
my_cities = {
    "New York":    (40.6943,  -73.9249),
    "Mumbai":      (18.9667,   72.8333),
}
def find_city(cities, city):
    if city not in cities:
        return None            # function ends execution here if city was not␣
 ↪in cities
    return cities[city]        # this line is only reached if city IS in cities
print(find_city(my_cities, "Mumbai"))
print(find_city(my_cities, "Zurich"))
```

```
(18.9667, 72.8333)
None
```

1.0.4 for loops

[7]:
```
for i in range(3):
    print(i)
```

```
0
1
2
```

[8]:
```
my_cities = {
    "New York":    (40.6943,  -73.9249),
    "Mumbai":      (18.9667,   72.8333),
}
# note that city is a string and coordinates is a tuple with 2 elements,␣
 ↪because items() returns key/value pairs.
for city, coordinates in my_cities.items():
    print(f"{city} is located at {coordinates[0]}, {coordinates[1]}")
```

```
New York is located at 40.6943, -73.9249
Mumbai is located at 18.9667, 72.8333
```

1.0.5 `while` loops

Use a `while` loop **only** if the number of effective iterations is unknown. Use `for` loops to iterate over collections!

[9]:
```
from random import randrange

player_is_alive = True
while player_is_alive:
    print("Player is stomping goombas and collecting coins")
    # player has a 1 in 5 random chance of dying on each iteration
    player_is_alive = not randrange(5) == 0
```

```
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
Player is stomping goombas and collecting coins
```

1.0.6 References

[10]:
```
l = [1, 2, 3]
```

[11]:
```
l = [1, 2, 3]
x = l
```

[12]:
```
x[0] = 7
print(l)
print(x)
```

```
[7, 2, 3]
[7, 2, 3]
```

1.0.7 Checking for Equality

- `==` checks if the two sides are *equivalent*
- `is` checks if the two sides are the exact same thing (i.e., the same object in memory).

[13]:
```
l = [7, 2, 3]
x = l
y = [7, 2, 3]
```

```
print(l == x)   # True
print(y == x)   # True as well, because the lists referred to by l and y both␣
 ↪contain the exact same elements
print(l is x)   # True, because l and x refer to the exact same object in memory
print(y is x)   # False, because y and x are two different objects in memory
```

```
True
True
True
False
```

Use `is` when comparing with `None`

[14]:
```
contacts = {"Bob": "+41991234567"}
def get_number(phonebook, person):
    if person not in phonebook:
        return None
    return phonebook[person]

if get_number(contacts, "Anna") is None:
    print("We don't know her number")
```

```
We don't know her number
```

Sidenote: be careful if you're dealing with values which could be *Falsy* or `None`. When used in a conditional expression, both will result in `None`:

[15]:
```
None == False
```

[15]:
```
False
```

[16]:
```
bool(None) == bool(False)
```

[16]:
```
True
```

[17]:
```
if not None:
    print("None is falsy")
if not False:
    print("False is falsy")
```

```
None is falsy
False is falsy
```

[18]:
```
def is_negative(x):
    if x < 0:
        return True
    if x > 0:
        return False
if is_negative(0):
    print("0 is negative")
```

```
else:
    print("0 is positive") # but by most definitions, zero is neither positive␣
 ↪nor negative
```

```
0 is positive
```

2 Pass-by-reference

Python function calls use ``pass-by-reference''. This means that if a function is called with a reference (i.e., a variable), then only that reference is passed into the function. The value referenced is *not* copied (the latter would be called ``pass-by-value'').

In the following example we * create a list containing three elements * create a variable `x` referencing that list * declare a function `foo` which takes one parameter (``a list'' `l`) and appends a value to it * call the function `foo` with `x` as the parameter. `x` is a reference to the list we created. The function receives this reference and appends an element to the referenced list. The function does not return anything. It just edited `l`. Because `l` references the same list as `x`, `x` appears to have been modified as well.

We also print whatever `x` refers to before and after calling `foo`.

[19]:
```
x = [1, 2, 3]
def foo(l):
    l.append(4)
print(x)
foo(x)
print(x)
```

```
[1, 2, 3]
[1, 2, 3, 4]
```

Things to keep in mind:

- A variable is a **reference** to something in memory.
- A variable is not the exact same thing as its value
- Multiple variables can reference the same thing in memory.
- When passing a variable to a function, it's only the reference that is place into the function. Values are not copied.

So semantically, the following would result in the exact same outcome:

[20]:
```
x = [1, 2, 3]
def foo():
    x.append(4)
print(x)
foo()
print(x)
```

```
[1, 2, 3]
[1, 2, 3, 4]
```

In this example, `foo` does not actually take any arguments. Instead, we reference `x` directly, rather than taking the detour of using `l`.

This brings us to the next topic…

3 Scope

Scope determines where in a program a given variable or function can be accessed and modified.

For example, `x` is *in scope* both inside and outside of the `foo` function:

[21]:
```
x = [1, 2, 3]
def foo():
    x.append(4)
foo()
print(x)
```

```
[1, 2, 3, 4]
```

Let's look at another example. Here's our familiar `power2` function, which takes a single parameter, `y`. When calling this function, for example `power2(4)`, then the variable `y` inside the function is assigned the value `4`:

[22]:
```
def power2(y):
    return y ** 2
power2(4)
```

[22]:
```
16
```

However, outside the function, `y` is undefined:

[23]:
```
def power2(y):
    return y ** 2
power2(4)
#print(y)           # impossible, because y is not defined in the module scope!
```

[23]:
```
16
```

This is because the *scope* of `y` is the function `power2`. In other words, `y` is only defined within the function body.

On the other hand, if we assign a variable on the module level (i.e., ``without any indentation''), then its scope is the entire module. Such a variable will be accessible anywhere (below its definition) within the module:

[24]:
```
z = 10
def power2(y):
    print(z)
    return y ** 2
print(z)
power2(4)
```

```
10
10
```

[24]:
```
16
```

As you can see, `z` is *in scope* both within the `power2` function and outside of it, because it was defined inside the *module* scope.

3.0.1 Shadowing

This can lead to a tricky situation called **shadowing**:

[25]:
```
y = 10
def power2(y):
    return y ** 2
power2(4)
```

[25]:
```
16
```

In this example, the declaration of `y` as a parameter inside `power2` **shadows** the definition of `y` in the module scope. The `y` inside the function is **not** the same as the `y` outside the function. This can be illustrated easily by printing the variable in multiple places:

[26]:
```
y = 10
def power2(y):
    print(y)        # second print output: value of y which is passed to the␣
 ↪function, has nothing to do with the y on line 1
    return y ** 2
print(y)            # first print output: value of y defined on line 1
power2(4)
```

```
10
4
```

[26]:
```
16
```

Even reassigning `y` within the function has no effect on the `y` outside the function!

[27]:
```
y = 10
def power2(y):
    y = 6           # reassigning y just overwrites the y passed to the␣
 ↪function on line 2!
    print(y)        # value of y which was assigned on line 3
    return y ** 2
print(y)            # still the value of y defined on line 1, reassignment␣
 ↪inside power2 had no impact on this!
print(power2(4))
print(y)
print(power2(4))
```

```
10
6
36
10
6
36
```

This is true even if the variable being assigned was not passed as a parameter!

Important: any **assignment** you make inside the function will create a new variable within the function scope, even if *a* variable with the same name exists before the assignment!

[28]:
```
y = 10
def power2():
    y = 6           # this does NOT reassign the y from line 1, but creates a␣
 ↪new y inside the function scope!
    print(y)        # value of y which was assigned on line 3
    return y ** 2
print(y)            # still the value of y defined on line 1, reassignment␣
 ↪inside power2 had no impact on this!
print(power2())
print(y)            # y is STILL 10, despite y = 6 inside the function!
print(power2())
```

```
10
6
36
10
6
36
```

It's best to avoid this kind of thing from happening altogether: if you're already using a variable name in a higher-level scope (for example the module scope), then use different variable names for parameters or variables in general in lower-level scopes:

[29]:
```
y = 10
def power2(num):        # avoid shadowing by giving this variable a name that␣
 ↪does not shadow anything from the higher-level scope
    return num ** 2
print(y)
print(power2(4))
```

```
10
16
```

A new scope is created for any * Module (i.e. a new file/script) * Function (i.e. anything declared using `def` or `lambda`) * Class (i.e. anything declared using `class`) * comprehension

However, the following do *not* create a new scope: * `if`/`elif`/`else` blocks * `for` loops

[30]:
```
if True:
    x = 10
print(x)       # x is in scope, because if statements do not create a new scope
def foo():
    x = 20     # this x defined here does NOT overwrite the x from the module␣
 ↪scope! It's a different x.
foo()
print(x)
```

```
10
10
```

So far, in (almost) all examples of functions, we assumed that the function receives **everything** it needs as parameters. For example, we pass to `find_city` both the dictionary to be searched as the parameter `cities` and the name of the city as parameter `city`:

```
def find_city(cities, city):
```

And then we call the function passing values for both parameters:

```
find_city(my_cities, "Mumbai")
```

such that inside the function `cities` *refers to* the dictionary assigned to `my_cities`

[31]:
```
my_cities = {
    "Tokyo":       (35.6839,  139.7744),
    "New York":    (40.6943,  -73.9249),
    "Mexico City": (19.4333,  -99.1333),
    "Mumbai":      (18.9667,   72.8333),
    "Sao Paulo":   (-23.5504, -46.6339),
}
def find_city(cities, city):
    if city not in cities:
        return None
    return cities[city]
print(find_city(my_cities, "Mumbai"))
print(find_city(my_cities, "Zurich"))
```

```
(18.9667, 72.8333)
None
```

But if we wanted to, we could rewrite this function to simply reference the module-scope variable `my_cities`:

[32]:
```
my_cities = {
    "Tokyo":       (35.6839,  139.7744),
    "New York":    (40.6943,  -73.9249),
    "Mexico City": (19.4333,  -99.1333),
    "Mumbai":      (18.9667,   72.8333),
    "Sao Paulo":   (-23.5504, -46.6339),
}
```

```
def find_city(city):
    if city not in my_cities:
        return None
    return my_cities[city]
print(find_city("Mumbai"))
print(find_city("Zurich"))
```

```
(18.9667, 72.8333)
None
```

This is ultimately a design choice. In general, referencing variables from outside the local scope can lead to more errors.

3.0.2 `global`

Remember that any *assignment* inside a function will create a new local variable, even if a variable with the same name exists in a higher-level scope:

[33]:
```
y = 10
def normal():
    y = 6               # this does NOT reassign the y from line 1, but creates␣
 ↪a new y inside the function scope!
print(y)
normal()
print(y)
```

```
10
10
```

There is a way of making a function-scope variable be the same as a variable from a higher scope:

[34]:
```
y = 10
def dont_do_this():
    global y            # this pulls the higher-level 'y' into the function scope
    y = 6               # now, (re)assigning y actually changes y in the module␣
 ↪scope
print(y)
dont_do_this()
print(y)
```

```
10
6
```

However: **don't do this**. There are some fringe reasons where it makes sense to do this, but it's practically never necessary and is generally considered a code smell.

4 Classes and objects

You can imagine that lists, tuples, sets and dictionaries can be used to represent a wide variety of real-world data. Let's say, for example, that we're writing a vector drawing program (like Adobe

Illustrator or Inkscape).

4.0.1 A vector graphics drawing program

Our program should be able to render various shapes on the screen. For this purpose, it will certainly need to have some internal representation for these shapes. Of course, some shapes can be defined in more than one way, but let's say we settle on these defining features for three of the most basic shapes:

- Square: side length
- Rectangle: width and height
- Circle: radius

We can use dictionaries and functions to store and transform these shapes. Say we store a dictionary with appropriate properties to describe any instance of each shape:

[35]:
```
s1 = {"type": "square", "side": 15}
s2 = {"type": "square", "side": 10}
r1 = {"type": "rectangle", "height": 5, "width": 100}
c1 = {"type": "circle", "radius": 2}
shapes = [s1, s2, r1, c1]
print(shapes)
```

```
[{'type': 'square', 'side': 15}, {'type': 'square', 'side': 10}, {'type':
'rectangle', 'height': 5, 'width': 100}, {'type': 'circle', 'radius': 2}]
```

And we could write functions to calculate the `area` of any shape:

[36]:
```
import math
def area(it):
    if it["type"] == "square":
        return it["side"]**2
    elif it["type"] == "rectangle":
        return it["height"] * it["width"]
    if it["type"] == "circle":
        return math.pi * it["radius"] ** 2
print(shapes)
[area(shape) for shape in shapes]
```

```
[{'type': 'square', 'side': 15}, {'type': 'square', 'side': 10}, {'type':
'rectangle', 'height': 5, 'width': 100}, {'type': 'circle', 'radius': 2}]
```

[36]:
```
[225, 100, 500, 12.566370614359172]
```

This *works*, but there are a couple of potential issues with this approach: * Creating new objects is a bit tedious: we have to spell out dictionary keys and values (…`"type": "rectangle"`…) a lot. Lot's of room for errors! * Working on these objects is also tedious: again we have to spell out the dictionary keys: (…`it["type"] == "rectangle"`…) * The `area` function is split apart by `if` conditions.

It all seems rather brittle. Because this kind of scenario is extremely common in real-world programming, most programming languages have mechanisms for structuring and abstracting it. Python

(like many other languages) supports *classes*.

Let's have another look just at the circle:

[37]:
```
c1  = {"type": "circle", "radius": 25}       # a specific circle, crudely␣
↪represented using a dictionary
def area(it):                                # a function that takes a␣
↪dictionary (which hopefully has the right keys to be a circle)
    if it["type"] == "circle":               # if the dictionary appears to be␣
↪a circle
        return math.pi * it["radius"] ** 2   # calculate and return its area
```

4.0.2 Introducing `class`

Instead of using `"type"` and `"radius"` keys in some arbitrary dictionary, we can define a `class` that describes what circles are like and what can be done with them.

A class defines the **attributes** and **behavior** of a thing. For our circle:

[38]:
```
class Circle:                                # NOT a specific circle! Just a␣
↪description of what a circle is and does
    def __init__(self, radius):              # __init__ is called when a new␣
↪Circle is created
        self.radius = radius                 # we store radius as an attribute of␣
↪a newly created Circle
    def area(self):                          # now instead of taking a dictionary␣
↪"it", the function just takes a Circle object "self"
        return math.pi * self.radius ** 2    # we compute and return the area␣
↪just like before, but reading self.radius instead of it["radius"]
```

Now we have a class `Circle` (note that class names are written using `CamelCase` by convention), with two functions: * `__init__` determines what happens when we want to create a new circle * `Circle`'s `area` method does the scaling just like the old function previously did

When functions are part of a class, we usually call them *methods* instead of *functions*.

Now instead of

[39]:
```
c1  = {"type": "circle", "radius": 2}
```

we will do

[40]:
```
c1 = Circle(2)
```

The number `25` we pass along is received by the special `__init__` function.

It's obvious that the type is a `Circle` because that's the point of having classes. In fact, Python will happily tell us its type, while before, the type would have been `dict`:

[41]:
```
type(c1)
```

[41]:
```
__main__.Circle
```

And instead of scaling a circle like this:

[42]:
```
c1  = {"type": "circle", "radius": 2}
area(c1)
```

[42]:
```
12.566370614359172
```

we can do this:

[43]:
```
c1 = Circle(2)
c1.area()
```

[43]:
```
12.566370614359172
```

Notice how the old function…

[44]:
```
def area(it):
    if it["type"] == "circle":
        return math.pi * it["radius"] ** 2
```

…took an object `it` (which at that point was just a simple dictionary representing a circle) and changed the value for the `radius` key.

The new method…

[45]:
```
    def area(self):
        return math.pi * self.radius ** 2
```

does exactly the same. Just like `it` previously, `self` inside a class always refers to ``the object we're talking about here''. And because `area` is implemented inside the `Circle` class, Python will automatically fill in `self` with the object where the function call is taken:

[46]:
```
c1.area()           # no need to specify any parameters: 'self' is obviously c1
```

[46]:
```
12.566370614359172
```

Seriously! What Python does internally, you can do yourself, if you really want to for some reason. if you call the method on the *class* rather than an object, you will have to supply `self` (in this case, `c1`) by hand:

[47]:
```
c1 = Circle(2)
Circle.area(c1)     # The upper-case C at the beginning is important! This calls␣
 ↪the method on the Circle CLASS, not on the c1 OBJECT
```

[47]:
```
12.566370614359172
```

Inside the `area` method, instead of specifying a dictionary key `["radius"]`, we refer to the objects radius via the `self.radius` property. Remember, we set that property when the object was created using `__init__`. Here's the whole class again:

[48]:
```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * self.radius ** 2
c1 = Circle(2)              # c1.radius is set by __init__ (where self refers␣
 ↪to the same object as c1)
c1.area()                   # area calculated based on c1.radius in area (where␣
 ↪self refers to the same object as c1)
```

[48]:
```
12.566370614359172
```

[49]:
```
c1 = Circle(2)
c1.area()
```

[49]:
```
12.566370614359172
```

4.0.3 printing objects

One thing you'll notice rather quickly is that printing your proud creation will not be particularily insightful:

[50]:
```
print(c1)
```

```
<__main__.Circle object at 0x7f7d0122e390>
```

You have to gift your classes a nice string representation yourself. You do this by implementing the `__str__` and `__repr__` methods: * ``__str__` is for customers''. It should return a pretty representation of the object * ``__repr__` is for developers''. It should return an unambiguous representation (ideally one that could be used to recreate the object)

For our simple `Circle`, this should suffice:

[51]:
```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * self.radius ** 2
    def __str__(self):
        return f"A circle with radius {self.radius}"
    def __repr__(self):
        return f"Circle({self.radius})"
c1 = Circle(25)
print(c1)
```

```
A circle with radius 25
```

You'll notice that when you `print` an object, the `__str__` method will be used. However, when you print some other data structure that just refers to your object, then the `__repr__` method will be used. This is easily illustrated if we put some `Circles` in a list and then print the **list**:

[52]:
```
my_circles = [c1, Circle(70), Circle(1)]
print(my_circles[1])    # __str__ is called to print the single instance at␣
↪index 1
print(my_circles)       # __repr__ is called for each instance in the list
```

```
A circle with radius 70
[Circle(25), Circle(70), Circle(1)]
```

The existing method `area` reads properties of a given shape and returns a value. Naturally, doing this doesn't change the object. But just like we can modify collections like lists and dictionaries, we can also modify our custom objects.

Let's implement another method for our `Circle` class which will be able to change the size of the circle. But before that, let's remember how we would have done it in the past, using just dictionaries and plain functions:

[53]:
```
c1  = {"type": "circle", "radius": 25}       # a specific circle, crudely␣
↪represented using a dictionary
def scale(it, factor):                       # a function that takes a␣
↪dictionary (which hopefully has the right keys to be a circle) and a factor
    if it["type"] == "circle":               # if the dictionary appears to be␣
↪a circle
        it["radius"] *= factor               # modify the dictionary
```

In contrast to the `area` function, which does not take any additional parameters beyond `it`, the `scale` function takes one additional parameter `factor`. Likewise, our `Circle` method will take an additional parameter beyond `self`:

[54]:
```
class Circle:                           # NOT a specific circle! Just a description␣
↪of what a circle is and does
    def __init__(self, radius):     # __init__ is called when a new Circle is␣
↪created
        self.radius = radius        # we store radius as an attribute of a␣
↪newly created Circle
    def scale(self, factor):        # now the function just takes "self"␣
↪instead of "it", and it's a Circle object, rather than just a dictionary
        self.radius *= factor       # we modify the Circle objects radius, just␣
↪like before

    # the other methods
    def area(self):
        return math.pi * self.radius ** 2
    def __str__(self):
        return f"A circle with radius {self.radius}"
    def __repr__(self):
        return f"Circle({self.radius})"
```

Now instead of scaling a circle like this:

[55]:
```
c1  = {"type": "circle", "radius": 25}
print(c1["radius"])
scale(c1, 3)
print(c1["radius"])
```

```
25
75
```

we can do this:

[56]:
```
c1 = Circle(25)
print(c1.radius)
c1.scale(3)
print(c1.radius)
```

```
25
75
```

Exercise

Fully implement `Square` and `Rect` classes, both with appropriate `area` and `scale` functions.

Note that the style guide recommends leaving a blank line between methods (makes it much easier to read):

[ ]:
```
class Square:
    pass


class Rect:
    pass


class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def scale(self, factor):
        self.radius *= factor

    def __str__(self):
        return f"A circle with radius {self.radius}"

    def __repr__(self):
        return f"Circle({self.radius})"
```

```
# when you've implemented all three, this should work:
shapes = [Square(150), Square(99), Rect(50, 150), Circle(25)]
print([shape.area() for shape in shapes])
for shape in shapes:
    shape.scale(3)
print(shapes)            # uses __repr__
for shape in shapes:
    print(shape)         # uses __str__
```