Discussion 8

DSC 20, Spring 2023

classes, copy types, exceptions

Agenda

- **Classes** - self, instance vs class, inheritance
- **Copy Types** - deep vs shallow
- **Exceptions** - types and cases, raise vs assert

Classes

Classes aggregate code together into a functional body. A lot of effective python code is written as classes (every time you install a new package, it's basically written as a lot of different classes). For example, recall the pandas dataframe. If you look at the source code for pandas, you'll see that it's a very complicated class definition.

Classes all generally have (at the minimum) a constructor (__init__) function and other object related methods that are used.

Working with self

When you invoke a class, you've instantiated an instance of the class (i.e. you've created an object of the same framework as you defined in the class). Think of classes as a blueprint and everytime you create an instance of one, you've produced a "physical" manifestation. In order to set values for each unique manifestation, we need to use the 'self' keyword. Try to think of 'self' as referencing a **specific, singular** instance of the class. Every method that works with a specific instance's values must be passed in self as an argument.

In [1]:
```
class rectangle:
    is_rectangle = True
    rec_counter = 0
    def __init__(self, length, width):
        self.length = length
        self.width = width
        rectangle.rec_counter+=1

    def get_area(self):
        return self.length * self.width

    def get_perimeter(self):
        return 2*self.length + 2*self.width
```

In [2]:
```
rec_1 = rectangle(4,4)
rec_2 = rectangle(2,6)
print("rectangle 1's area is: %s units" %rec_1.get_area())
print("rectangle 2's area is: %s units" %rec_2.get_area())
```

```
rectangle 1's area is: 16 units
rectangle 2's area is: 12 units
```

The 2 rectangles are different **instances** of class rectangle. Both of them are objects created from the framework of their original class. The use of the self keyword allows for the 2 of them to exist separately despite being generated from the same class.

Instance vs Class variables

Instance variables are attached to instances of a class by keyword self (usually done in the constructor). Class variables are variables attached to the class itself.

In [3]:
```
rectangle.rec_counter
```

Out[3]:
```
2
```

The previously defined class contained a variable named rec_counter which was incremented everytime the constructor was called. This acts as a "tracker" for how many rectangles have been created so far and is an example of a class variable. If I made another rectangle, it should continue tracking and increase to 3.

In [4]:
```
rec_3 = rectangle(1,1)
rectangle.rec_counter
```

Out[4]:
```
3
```

Checkpoint

What is the output of the following code?

In [5]:
```
class tutor:
    ID = 1
    def __init__(self, name, prof, course):
        self.name = name
        self.prof = prof
        self.course = course
        self.id = tutor.ID
        tutor.ID += 1
    def change_course(self, prof, course):
        if self.prof != prof and self.course != course:
            self.prof = prof
            self.course = course
            self.changed = True
        return False

t1 = tutor('cyh', 'suraj', 'DSC10')
t2 = tutor('bhc', 'marina', 'DSC20')
t3 = tutor('nwz', 'marina', 'DSC20')
```

In [6]:
```
t3.id # output 1
tutor.change_course(t1, 'marina', 'dsc20') # output 2
t2.change_course('marina','dsc30') #output 3
t1.changed # output 4
```

Checkpoint Solution

In [7]:
```
t3.id
```

Out[7]:
```
3
```

In [8]:
```
tutor.change_course(t1, 'marina', 'dsc20')
```

Out[8]:
```
False
```

In [9]:
```
t2.change_course('marina','dsc30')
```

Out[9]:
```
False
```

In [10]:
```
t1.changed
```

Out[10]:
```
True
```

Inheritance

Classes can inherit functionality from a parent class, allowing for additional methods to be added and old ones to be modified. Classes can also inherit from multiple parent classes. reading on the topic can be found here.

In [11]:
```
class lecture:
    def __init__(self, date, topic, length):
        self.date = date
        self.topic = topic
        self.length = length
    def get_length(self):
        return self.length

class discussion(lecture):
    def __init__(self, date, topic, length, pq=4):
        super().__init__(date, topic, length)
        self.pq = pq
    def get_length(self):
        return self.length + self.pq
```

In [12]:
```
lec = lecture('05-17-23', 'classes', 10)
disc = discussion('05-17-23', 'classes', 10)
print("lec's length is: %s" %lec.get_length())
print("disc's length is: %s" %disc.get_length())
```

```
lec's length is: 10
```

```
disc's length is: 14
```

discussion's constructor invokes its parent's constructor and adds its own parameter. It also overwrites the get_length function from its parent.

Checkpoint

What is the output of the following code?

In [13]:
```
class post_strike(tutor):
    def __init__(self, name, prof, course, exp=0):
        super().__init__(name, prof, course)
        self.exp = 0
    def teach(self):
        self.exp += 1
    def change_course(self, prof, course):
        if self.course != course and self.exp > 1:
            self.prof = prof
            self.course = course
            return True
        else:
            return False
    def go_strike(self):
        self.strike = True
t1 = post_strike('cyh', 'suraj', 'DSC10')
t2 = post_strike('bhc', 'marina', 'DSC20')
t3 = tutor('nwz', 'marina', 'DSC20')
```

In [ ]:
```
t1.teach(); t1.teach(); t1.change_course('marina','dsc20') # output 1
t2.go_strike(); t2.strike # output 2
t3.strike # output 3
t3.change_course('marina', 'dsc30') # output 4
```

Checkpoint Solution

In [15]:
```
t1.teach(); t1.teach(); t1.change_course('marina','dsc20')
```

Out[15]:
```
False
```

In [16]:
```
t2.go_strike(); t2.strike
```

Out[16]:
```
True
```

In [17]:
```
t3.strike
```

```
---------------------------------------------------------------
-----------
AttributeError                            Traceback (most recent
call last)
Cell In[17], line 1
----> 1 t3.strike

AttributeError: 'tutor' object has no attribute 'strike'
```

In [18]:
```
t3.change_course('marina', 'dsc30')
```

Out[18]:
```
False
```

Deep vs Shallow Copies

In short, shallow copies copy the **reference** while deep copies copy the **object**. Changes made to a shallow copy are reflected in the original object because the reference is maintained. Changes made to a deep copy are **not reflected** in the original object because it's a completely different object with a difference reference.

In [19]:
```
even_nums = [2,4,6,8,10]
odd_nums = [1,3,5,7,9]

shallow = even_nums
deep = list(odd_nums)

shallow.append(0)
deep.append(0)

print("even_nums: %s" %even_nums)
print("shallow: %s" %shallow)
print("odd_nums: %s" %odd_nums)
print("deep %s" %deep)
```

```
even_nums: [2, 4, 6, 8, 10, 0]
shallow: [2, 4, 6, 8, 10, 0]
odd_nums: [1, 3, 5, 7, 9]
deep [1, 3, 5, 7, 9, 0]
```

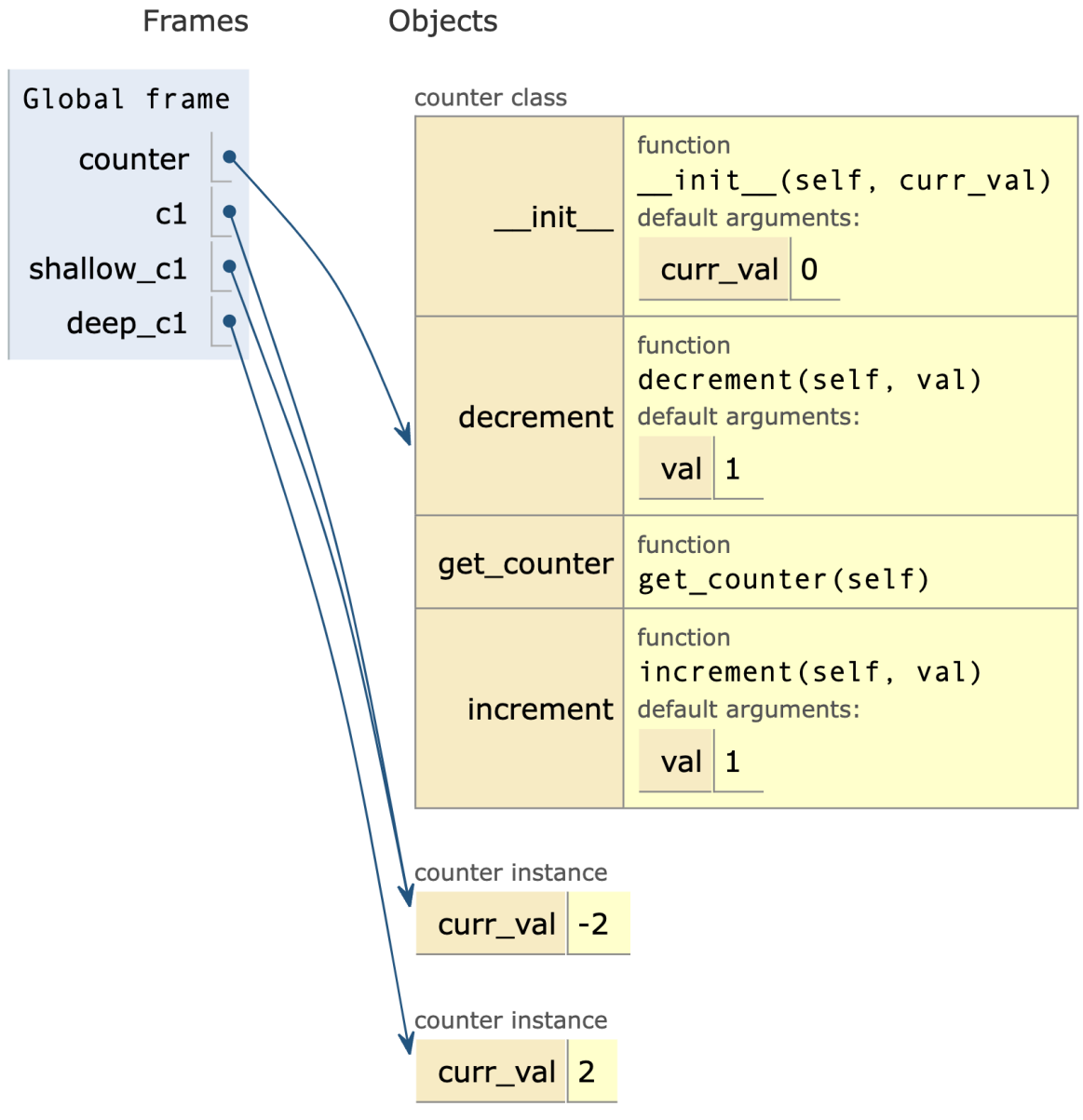
The same applies for classes you define. Deep copies are made by calling the constructor of the class on the same parameters (in the previous example, the constructor was the list() caster). Consider the class:

In [20]:
```
class counter:
    def __init__(self, curr_val = 0):
        self.curr_val = curr_val
    def get_counter(self):
        return self.curr_val
    def increment(self, val=1):
        self.curr_val += val
    def decrement(self, val=1):
        self.curr_val -= val
```

In [21]:
```
c1 = counter()
shallow_c1 = c1
deep_c1 = counter(c1.get_counter())
shallow_c1.decrement(2)
deep_c1.increment(2)
print("c1's value: %s" %c1.get_counter())
print("shallow_c1's value: %s" %shallow_c1.get_counter())
print("deep_c1's value: %s" %deep_c1.get_counter())
```

```
c1's value: -2
shallow_c1's value: -2
deep_c1's value: 2
```

how it actually works in the background


Frames
Objects
Global frame
counter
c1
shallow_c1
deep_c1
counter class
__init__
function
__init__(self, curr_val)
default arguments:
curr_val 0
decrement
function
decrement(self, val)
default arguments:
val 1
get_counter
function
get_counter(self)
increment
function
increment(self, val)
default arguments:
val 1
counter instance
curr_val -2
counter instance
curr_val 2

Exceptions

Errors! Errors are a special class in Python. Errors are triggered when improper code is attempted. You've seen them a millions of times from your own code, now you can create them using the raise keyword.

Exception Types and Cases

- **KeyError** - related to dictionaries; attempted access using a key not present in the object
- **IndexError** - related to lists/strings; attempted acccess of an index that's out of range
- **TypeError** - attempt to unify non - matching data types (ex. str + int) or attempt to access unknown attribute of datatype
- **FileNotFoundError** - related to files; attempted to open a file name that can't be found

raise

python keyword to throw an error.

syntax: raise [error type](error message // optional)

In [22]:
```
def foo(x):
    if x < 21:
        raise ValueError("number too small")
foo(0)
```

```
---------------------------------------------------------------
-----------
ValueError                                Traceback (most recent
call last)
Cell In[22], line 4
      2     if x < 21:
      3         raise ValueError("number too small")
----> 4 foo(0)

Cell In[22], line 3, in foo(x)
      1 def foo(x):
      2     if x < 21:
----> 3         raise ValueError("number too small")

ValueError: number too small
```

try except else

Exceptions can be handled in try-except blocks. This prevents code from terminating the moment an error happens.

In [23]:
```
def foo(z):
    try:
        z.append('y')
    except AttributeError as e:
        print(e)
    else:
        print(z)
        print('done!')

foo(4)
print('------------------------------------')
foo([])
```

```
'int' object has no attribute 'append'
------------------------------------
['y']
done!
```

try-except vs assert

Assert statements are used to validate inputs and prevent logical errors. Try-except is used to catch error generating inputs / code.

In [24]:
```
def process_file(filepath):
    # What happens if I pass in an incorrect filepath?
    assert isinstance(filepath, str) # asserts can't catch such errors
    with open(filepath, 'r') as f:
        return f.readlines()
process_file('nope')
```

```
---------------------------------------------------------------------
-----------
FileNotFoundError                         Traceback (most recent
call last)
Cell In[24], line 6
      4     with open(filepath, 'r') as f:
      5         return f.readlines()
----> 6 process_file('nope')

Cell In[24], line 4, in process_file(filepath)
      1 def process_file(filepath):
      2     # What happens if I pass in an incorrect filepath?
      3     assert isinstance(filepath, str) # asserts can't cat
ch such errors
----> 4     with open(filepath, 'r') as f:
      5         return f.readlines()
```

```
File /opt/anaconda3/envs/tutor/lib/python3.10/site-packages/IPyt
hon/core/interactiveshell.py:284, in _modified_open(file, *args,
**kwargs)
    277 if file in {0, 1, 2}:
    278     raise ValueError(
    279         f"IPython won't let you open fd={file} by defaul
t "
    280         "as it is likely to crash IPython. If you know w
hat you are doing, "
    281         "you can use builtins' open."
    282     )
--> 284 return io_open(file, *args, **kwargs)

FileNotFoundError: [Errno 2] No such file or directory: 'nope'
```

In [25]:
```
def process_file(filepath):
    # What happens if I pass in an incorrect filepath?
    try:
        f = open(filepath,'r')
    except FileNotFoundError as e:
        print('filepath is not valid')
        print(e)
    else:
        return f.readlines()
process_file('nope')
print('-------------------------------------')
process_file('files/sample.txt')
```

```
filepath is not valid
[Errno 2] No such file or directory: 'nope'
-------------------------------------
```

Out[25]:
```
["It's week 8!"]
```

Checkpoint

What's the output? If there is an error, how do you change the function to fix it?

In [26]:
```
def open_something(filepath):
    assert isinstance(filepath, str)
    try:
        with open(filepath,'r') as f:
            data = f.readlines()
    except FileNotFoundError as e:
        print('filepath is not valid')
    else:
        try:
            output = sum(data)
        except TypeError as e:
            print('not numbers')
        else:
            return output
```

In [27]:
```
with open('files/math.txt', 'r') as f:
    print(f.read())
```

```
1
2
3
4
5
6
```

```
7
8
9
10
```

In [28]:
```
open_something('files/math.txt')
```

```
not numbers
```

Checkpoint Solution

In [29]:
```
open_something('files/math.txt')
```

```
not numbers
```

In [30]:
```
def fixed(filepath):
    assert isinstance(filepath, str)
    try:
        with open(filepath,'r') as f:
            data = f.readlines()
    except FileNotFoundError as e:
        print('filepath is not valid')
    else:
        try:
            data = [int(x) for x in data]
            output = sum(data)
        except (TypeError, ValueError) as e:
            print('not numbers')
        else:
            return output
```

In [31]:
```
fixed('files/math.txt')
```

Out[31]:
```
55
```

Thanks for coming!

There's a discussion quiz on canvas!