Math 260: Python programming in math

Fall 2020

Loose ends, more on objects

Sub-classes

An object is expressed as a sub-type of another with a **sub-class**.

- A sub-class inherits the properties from the larger class.
  - called the 'super-class' or 'base class'
- Represents 'is-a' relationships (cat is an animal, special matrices...)

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

    def annoy(self):
        print(self.noise*100)
```

```
c = Cat("mittens")
d = Dog("rover")
c.annoy() #defined!
c.fetch() #not defined for cats!
```

```
class Cat(Pet):
    def __init__(self, name, age):
        super().__init__(name, "meow")
        self.lives = 9

    def knock_over(obj):
        obj.destroy()

class Dog(Pet):
    def __init__(self, name, age):
        super().__init__(name, "woof")

    def fetch(obj):
        return obj
```

- `super()` refers to the base class
- In init, we (usually) call the constructor of this class first.
- Then, initialize anything specific to the sub-class.

Inheritance rules

Certain rules govern how objects inherit. AA key property is **Overloading:**

- Python first searches in the class, then in the base class of it, and so on
- If a sub-class has no *__init__* defined, it inherits the base class constructor

```
class Fish(Pet):
    #...inherits the Pet constructor...
    def swim():
        return
f = Fish("Nemo",1,"bloop")
```

- Two sub-classes can overload the same base function with unique defs!

```
class Cat(Pet):
    def __init__(self, name, age):
        super().__init__(name, age,"meow")
    def annoy(self):
        print("Meow Meow Meow!")
        swat_owner()
```

```
class Dog(Pet):
    def __init__(self, name, age):
        super().__init__(name, age,"woof")
    def annoy(self):
        print("Woof Woof!")
        bark()
```

- `animal.annoy()` will call `Cat.annoy` for a cat and `Dog.annoy` for a dog.

Polymorphism

```
s Cat(Pet):
def __init__(self, name, age):
    super().__init__(name, age,"meow")
def annoy(self):
    print("Meow Meow Meow!")
    swat_owner()
```

```
print(f" Time taken: elapsed:.1e")
```

```
class Dog(Pet):
    def __init__(self, name, age
        super().__init__(name, a
    def annoy(self):
        print("Woof Woof!")
        bark()
```

- `animal.annoy()` will call `Cat.annoy` for a cat and `Dog.annoy` for a dog.
- This property is called **polymorphism**: the ability of an object to be one of several types.
- This way of representing is-a relationships is an example of object oriented programming (OOP)

You've seen polymorphism in action before!

- For loops require 'iterable' objects. Each type defines what that means.
- Errors inherit from the base exception class `Exception`. You can create your own by sub-classing it!

Key point: This sort of OOP is only worth doing if the structure is truly important! (your functions must care about the relationships)

Modularity

Another useful principle is **modularity**.

- The idea is to separate the program into **modular parts** that operate more or less independently
- Each part does not care about the internal workings of the other
- Example: your Matrix class has an internal representation of its data, methods for adding etc.
- Example: `solve(A,b)` from HW 3 hides the LU factorization steps
- Algorithms are 'black boxes' that take inputs in and magically produce output

Modularity

Why write modular code?

- Code can be fit into / combined easily with other algorithms
- Parts can be independently tested

Why not?

- The modular property can be harder to maintain than a less modular code
- Code that tries to be too general can become a mess (if dependencies make the code better, don't try to modularize!)
- You can require the user/programmer to know what they are doing (and deal with any requirements for the functions)

Miscellaneous python

Iterators: map

Earlier we saw that `zip` creates an **iterator** of sets of lists:

```
for k, v in zip(keys, vals):
    print(k,v)
```

- (iterators have a starting point and a defined 'next item')
- An iterator allows python to avoid creating memory
- (in the same way that `range` iterates over integers)

The **map** function applies a function to a set of lists or other iterables:

```
def combine(x,y):
    return 2*x + y

m = map(combine, [1,2,3,4], [5,6,7,8])
a = list(m) #list version: [7,10,13,16]
for v in m:
    print(v)
```

- Returns a 'map' object that is iterable (not a list!)
- Unlike a list comprehension, does not actually build the list

Iterators: generators

This, however, is not really the python way.

- We'd like to use a list comprehension! But this is wasteful...

```
def f(x):
    return 2*x

# creates a new list
for val in [f(x) for x in range(10)]
    print (val)
```

```
def f(x):
    return 2*x

# does not create a list
for x in range(10):
    val = f(x)
    print(val)
```

- Instead, python offers **generator expressions**, which are iterable objects that describe applying a function to a set of things:

```
def f(x):
    return 2*x

for val in (f(x) for x in range(10)):
    print(val)
```

- Syntax difference: () instead of []
- Python uses **lazy evaluation**: the effective 'list' of values is not created; instead, elements are computed as needed.

Packaging your code: wrappers

Solution: Use a **wrapper function...**

```
def internal_sort(j, k, arr, work):
    #... some mergesort like function ...
    # [arr is sorted at the end]

def sort(arr, overwrite=True):
    """ Documentation goes here """
    if not overwrite:
        sorted = make_a_copy_of(arr)
    else:
        sorted = arr
    work = new_array(len(arr))
    internal_sort(0, len(arr)-1, sorted, work)
    return sorted

#... some application...
a =-sort(mydata, overwrite=False)
```

- A **wrapper** 'wraps' around your actual function, hiding what's inside
- It separates the **usage** of the algorithm from its **implementation**
- Changes to internal sort (e.g. order of its inputs) can be made *without* the 'application' code changing!

Style: tuples vs. multiple arguments

Suppose you have a function that takes a pair of data:

```
def integ1(f, ival):   # call: integ(f, [0, 1])
# OR...
def integ2(f, a, b):   # call: integ(f, 0, 1)
```

- You have to choose the 'shape' and type of inputs
- Python has a trick for 'distributing' tuples in arguments:

```
ival = (0, 1)
integ1(f, ival)  # works
integ2(f, ival[0], ival[1])  # inelegant
integ2(f, *ival)  # equivalent to above
```

- The star prefix says: 'put the elements of the tuple into the arguments'.
- You can use this to clean up code when you need both forms of input
- Example: multiple returns are always a tuple...

```
def get_ival(x):
    return x, 2*x

integ2(f, *get_ival(3))
```

Decorators

Suppose you want to write some timing code for a function:

```
from time import perf_counter

start = perf_counter()
myfunc()
elapsed = perf_counter - start
print("Time taken: {elapsed:.1e}")
```

The 'shell' has to be written out each time. You could do this:

```
def timer(func):
    start = perf_counter()
    myfunc()
    elapsed = perf_counter() - start
    print(f"Time taken: {elapsed:.1e}")

timer(myfunc)
```

But (more generally)¡ we often want to 'add a property' to a function/class.

- `myfunc` has a 'timing' functionality added to it
- The `timer` function wraps the function, sort-of...

Decorators

Let's write a function that sticks a new property onto a function:

```
def timerify(func):
    def func_with_timer(*args, **kwargs):
        start = perf_counter()
        output = func(*args, **kwargs)
        elapsed = perf_counter() - start
        print(f"Time: {elapsed:.1e}")
        return output
    return func_with_timer
```

Usage:

```
def myfunc(x):
    return 2*x
myfunc_timed = timerify(myfunc)  # **build the new function

y = myfunc_timed(2) # now also prints elapsed time
```

This new function, called a **decorator**, adds timing code and returns a new 'decorated' function.

Decorators

```
def myfunc(x):
    return 2*x
myfunc_timed = timerify(myfunc)  # **build the new function

y = myfunc_timed(2) # now also prints elapsed time
```

The syntax here is clumsy. Python has better syntax to simplify:

```
@timerify
def myfunc(x):
    return x
```

- The @ says 'apply the decorator to this function'.
- Shorthand for the (**) line above
- Sophisticated decorators can be written (with some effort) to do more...
- See `functools` package for many examples

Key point:

Decorators are used to add properties by 'decorating' other functions/classes. Decorators (the @ code) can be written for useful actions like timing, bounds/type checking, deducing functions from others (like $<$ from $\geq$)...

Decorators (a glimpse at more)

- Decorators often need to know about their input function/class
- Python gives you ways for functions/classes to know its attributes, e.g.:

```
func.__name__  # (name of a function)
obj.__dict__ # (instance variables in class / values)
```

Suppose we want our timer to also print the function name...

```
def timing(func):

    @functools.wraps(func)
    def func_with_timer(*args, **kwargs):
        start = perf_counter()
        output = func(*args, **kwargs)
        elapsed = perf_counter() - start
        print("{}, time: {:1e}".format(func.__name__, elapsed))
        return output

    return func_with_timer
```

The 'wraps' decorator forces the decorated function to keep its original name (`func.__name__`), rather than the wrapped name (`func_with_timer`).