STATS 507
Data Analysis in Python

Week4-2: More class method, Iterators, and Generators, Exceptions

Dr. Xian Zhang

Adapted from slides by Professor Jeffrey Regier

Recap: Class as programmer-defined types

Objects are instances of a class and are a data abstraction that captures:

- An **internal representation**
    - Through data attributes

- An **interface** for interacting with objects
    - Though methods (aka procedures/functions)
    - Defines behaviors but hides implementations

Creating the class (a parallel to function)

- Define the class **name**
- Define class <u>data attributes</u>
- Define <u>procedural attributes</u>

Using the class:

- Create new **instances** of the class
- Doing operations on the instances

Recap: creating our own class

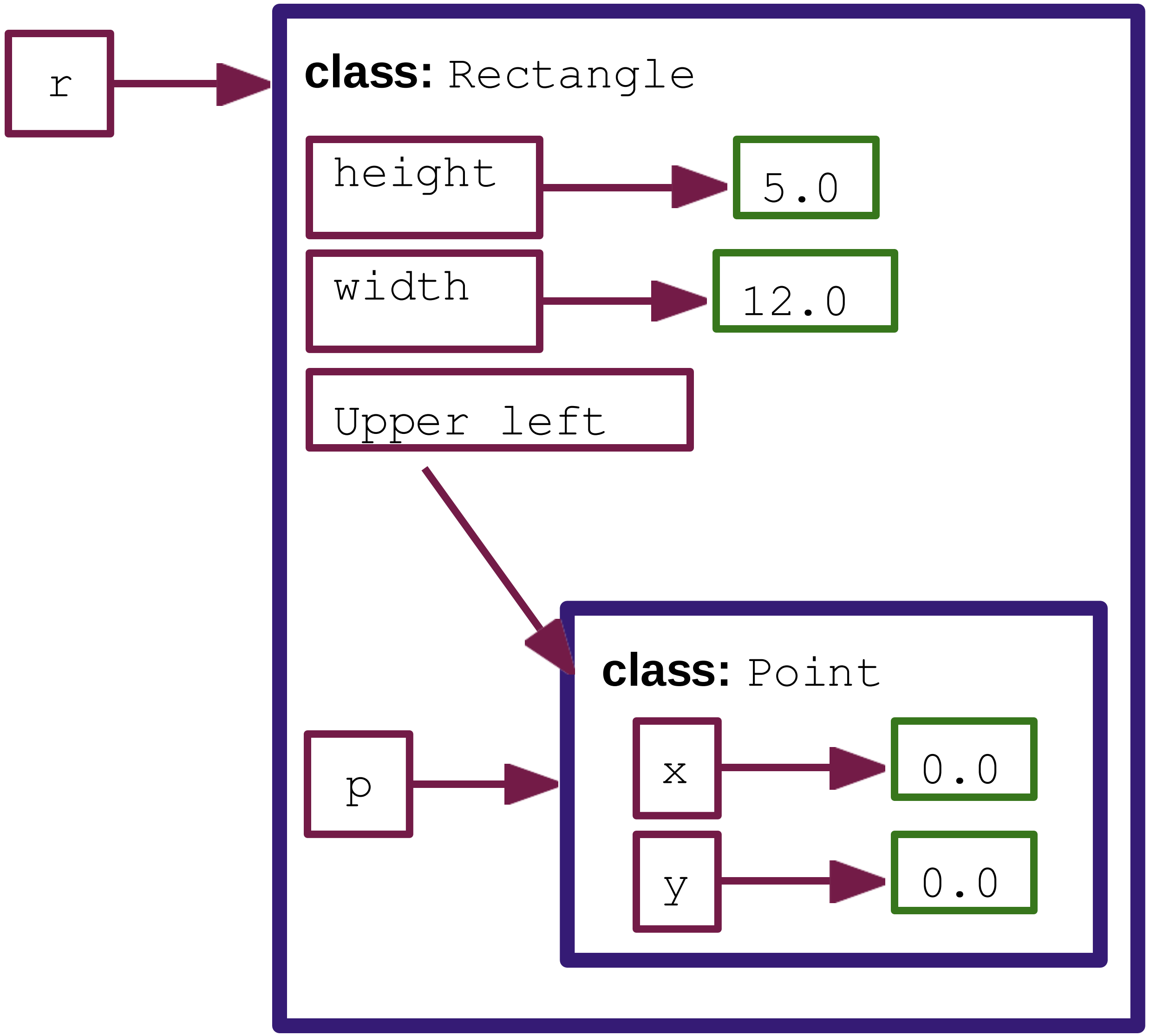
```python
# YOUR CODE HERE
class Rectangle:
    def __init__(self, upper_left, height, width):
        if not isinstance(upper_left, Point):
            raise TypeError("upper_left must be a Point object")
        self.upper_left = upper_left
        self.height = height
        self.width = width

    def __str__(self):
        return f"Rectangle(upper_left={self.upper_left}, " \
               f"height={self.height}, width={self.width})"

    def area(self):
        return self.height * self.width

upper_left_corner = Point(0,0)
my_rectangle = Rectangle(upper_left_corner, 2, 4)
print(my_rectangle)
print(my_rectangle.area())
```

```
Rectangle(upper_left=Point(0, 0), height=2, width=4)
8
```



Recap: Inheritance

Inheritance is perhaps the most useful feature of object-oriented programming

Inheritance allows us to <u>create new classes from old ones</u>

Parent class

(base class/superclass)

Child class

(derived class/ subclass)

- **Inherit** all data and behaviors of the parent class
- **Add** more **<u>info(data)</u>**
- **Add** more **<u>behavior</u>**
- **Override** behavior

Class definition Class name

Parent class

```
class Cat(Animal):
    def speak(self):
        print("meow")
```

```
my_cat = Cat(7)
my_cat.set_name("Fay")
print(my_cat.get_name())
print(my_cat.speak())
```

```
Fay
meow
None
```

One more thing for OOP on `__init__()`

`__init__()` for parent class

A special method to **initialize** some data attributes or perform initialize operations.

`self` represents an instance of a class, It is a parameter to refer to an instance of the class without creating one yet. **Always** going to be the first parameter of any method.

```
class Animal():
    def __init__(self, age):
        self.age = age
        self.name = None
    def __str__(self):
        return "animals: " + str(self.name) + ":" + str(self.age)
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, new_age):
        self.age = new_age
    def set_name(self, new_name = ''):
        self.name = new_name
```

`__init__()` for child class

```
class Cat(Animal):
    def __init__(self, age, breed="Unknown"):
        super().__init__(age)
        self.breed = breed
    def speak(self):
        print("meow")
    def get_breed(self):
        return self.name
    def set_breed(self, breed):
        self.breed = breed
```

`super().__init__()` to ensure the attributes in parent class are properly initialized.

additional initialization steps specific to the subclass

```
my_cat = Cat(7)
my_cat.set_name("Fay")
my_cat.set_breed("British short hair")
print(my_cat.get_name())
print(my_cat.get_breed())
print(my_cat.speak())
```

```
Fay
Fay
meow
None
```

1. Iterator and Generators

2. Exceptions and Assertions

Recall: iteration and iterable

Iterable: an object capable of returning its members one at a time:

- Lists are iterable.
- So are all sequence types (string, tuple…)
- Some non-sequence types like dict, file objects are also iterable.

We can loop over objects that are iterable.

```python
nums = [1,2,3]
for n in nums:
    print(n)
```

```
1
2
3
```

```python
s = 'abc'
for c in s:
    print(c)
```

```
a
b
c
```

https://docs.python.org/3/glossary.html#term-iterable

How can we tell if an object is iterable?

The `__iter__()` method (another ”dunder” method):

- An object can be looped over if it has the `__iter__()` method
- We say it is iterable.

```
nums = [1,2,3]
print(dir(nums))
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc_
_', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__se
titem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'i
nsert', 'pop', 'remove', 'reverse', 'sort']
```

What is an iterator?

An iterator is an object that represents a “data stream”

Supports method `__next__()`:

returns next element of the stream/sequence

raises `StopIteration` error when there are no more elements left

Is list an iterator?

```
nums = [1,2,3]
print(next(nums))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[13], line 2
      1 nums = [1,2,3]
----> 2 print(next(nums))

TypeError: 'list' object is not an iterator
```

Getting an iterator

Lists are **not** iterators, so we first have to turn the list `t` into an iterator using the function `iter()`.

```
t = [1,2]
titer = iter(t)
next(titer)
```

```
1
```

Now, each time we call `next()`, we get the next element in the list. **Reminder:** `next(iter)` and `iter.__next__()` are equivalent.

```
next(titer)
```

```
2
```

Once we run out of elements, we get an error.

```
next(titer)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-20-105e88283d1e> in <module>()
----> 1 next(titer)

StopIteration:
```

Iterator in for loop

You are already familiar with iterators from previous lectures. When you ask Python to traverse an object `obj` with a for-loop, Python calls `iter(obj)` to obtain an iterator over the elements of `obj`.

```
t = [1,2,3]
for x in t:
    print(x)
print()
for x in iter(t):
    print(x)
```

```
1
2
3

1
2
3
```

These two for-loops are equivalent. The first one hides the call to `iter()` from you, whereas in the second, we are doing the work that Python would otherwise do for us by casting `t` to an iterator.

Creating an iterator – a dummy class

```python
class dummy():
    '''Class that is not iterable,
    because it has neither __next__()
    nor __iter__().'''

d = dummy()
for x in d:
    print(x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-fc084e213893> in <module>()
      5
      6 d = dummy()
----> 7 for x in d:
      8     print(x)

TypeError: 'dummy' object is not iterable
```

If we try to iterate over an object that is not iterable, we're going to get an error.

Objects of class `dummy` have neither `__iter__()` (i.e., doesn't support `iter()`) nor `__next__()`, so iteration is hopeless. When we try to iterate, Python is going to raise a `TypeError`.

Creating our own iterator

`__next__()`: create our own `__next__()` method

```python
class Squares():
    '''Iterator over the squares.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(k*k)
s = Squares()
[next(s) for _ in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`__next__()` is the important point, here. It returns a value, the next square.

`next(iter)` is equivalent to calling `__next__()`. Variable _ in the list comprehension is a placeholder, tells Python to ignore the value.

Can we loop over the `Squares`? Why?

Iterable v.s iterator

```python
class Squares():
    '''Iterator over the squares.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(k*k)
s = Squares()
for x in s:
    print(x)
```

Merely being an iterator isn't enough, either!
`for X in Y` requires that object `Y` be iterable.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-f0187d07bc4c> in <module>()
      7         return(k*k)
      8 s = Squares()
----> 9 for x in s:
     10     print(x)

TypeError: 'Squares' object is not iterable
```

Creating an iteratorable

```python
class Squares():
    '''Iterator over the squares.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(k*k)
    def __iter__(self):
        return(self)
s = Squares()
for x in s:
    print(x)
```

```
0
1
4
9
16
25
```

Iterable means that an object has the `__iter__`() method, which returns an iterator. So `__iter__`() returns a new object that supports `__next__`().

Now `Squares` supports `__iter__()` (it just returns itself!), so Python allows us to iterate over it.

This is an infinite loop.

Generator expressions

Recall that a list comprehension creates a list from an iterable

```
def square(k):
    return(k*k)
[square(x) for x in range(17) if x%2==0]
```

```
[0, 4, 16, 36, 64, 100, 144, 196, 256]
```

List comprehension computes and returns the whole list. What if the iterable were infinite? Then this list comprehension would never return!

```
s = Squares()
[x**2 for x in s]
```

This list comprehension is going to be infinite! But I really ought to be able to **get an iterator** over the squares of the elements.

```
sqgen = (x**2 for x in s)
sqgen
```

```
<generator object <genexpr> at 0x106d02780>
```

This is the motivation for **generator expressions**. Generator expressions are like list comprehensions, but they create an iterator rather than a list.

Generator expressions are written like list comprehensions, but with parentheses instead of square brackets.

Generators

Related to generator expressions are **generators**

Provide a simple way to write iterators (avoids having to create a new class)

```
def harmonic(n):
    return(sum([1/k for k in range(1,n+1)]))
harmonic(10)
```

```
2.9289682539682538
```

Each time we call this function, a local namespace is created, we do a bunch of work there, and then all that work disappears when the namespace is destroyed.

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
[next(h) for _ in range(3)]
```

```
[1.0, 1.5, 1.8333333333333333]
```

Alternatively, we can write `harmonic` as a **generator**. Generators work like functions, but they maintain internal state, and they `yield` instead of `return`. Each time a generator gets called, it runs until it encounters a `yield` statement or reaches the end of the `def` block.

https://en.wikipedia.org/wiki/Harmonic_number

Writing iterators using generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

Writing iterators using generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

Create a new `harmonic` generator. Inside this object, Python keeps track of where in the `def` code we are. So far, no code has been run.

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

Writing iterators using generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

Python sees the `yield` keyword and determines that this should be a generator definition rather than a function definition.

```
next(h)
```

```
1.0
```

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

If/when we run out of `yield` statements (i.e., because we reach the end of the definition block), the generator returns a `StopIteration` error, as required of an iterator (not shown here).

In-class practice

Lambda expressions

Lambda expressions let you define functions without using a def statement

- Called an **in-line function** or **anonymous function**
- Name is a reference to lambda calculus, a concept from symbolic logic

```
def my_square(x):
    return x**2
list(map(my_square, range(1,10)))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Define a function, then pass it to `map`.

```
list(map(lambda x: x**2, range(1,10)))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Alternatively, define an equivalent function **in-line**, using a **lambda statement**.

A lambda expression returns a function, so `my_square` and `lambda x: x**2` are, in a certain sense, equivalent.

Lambda expressions

```
lambda x : x**2 + 1
```

```
<function __main__.<lambda>>
```

Arguments of the function are listed before the colon. So this function takes a single argument...

```
lambda x,y,z,n : x**n + y**n == z**n
```

```
<function __main__.<lambda>>
```

...while this one takes four.

```
(lambda x,y,z,n : x**n + y**n == z**n)(3,4,5,2)
```

```
True
```

```
(lambda x,y,z,n : x**n + y**n == z**n)(13,17,19,42)
```

```
False
```

```
my_square
```

```
<function __main__.my_square>
```

Lambda expressions

```
lambda x : x**2 + 1
```

```
<function __main__.<lambda>>
```

Return value of the function is listed on the right of the colon. So this function returns the square of its input plus 1....

```
lambda x,y,z,n : x**n + y**n == z**n
```

```
<function __main__.<lambda>>
```

...and this one returns a Boolean stating whether or not the four numbers satisfy Fermat’s last theorem.

```
(lambda x,y,z,n : x**n + y**n == z**n)(3,4,5,2)
```

```
True
```

https://en.wikipedia.org/wiki/Fermat's_Last_Theorem

```
(lambda x,y,z,n : x**n + y**n == z**n)(13,17,19,42)
```

```
False
```

```
my_square
```

```
<function __main__.my_square>
```

Lambda function type and name

```
1 lambda x : x**2 + 1
```

```
<function __main__.<lambda>>
```

```
1 lambda x,y,z,n : x**n + y**n == z**n
```

```
<function __main__.<lambda>>
```

Lambda expressions return actual functions, which we can apply to inputs.

```
1 (lambda x,y,z,n : x**n + y**n == z**n)(3,4,5,2)
```

```
True
```

```
1 (lambda x,y,z,n : x**n + y**n == z**n)(13,17,19,42)
```

```
False
```

```
1 my_square
```

```
<function __main__.my_square>
```

Function names are stored in an attribute `__name__`. Since lambda expressions yield anonymous functions, they all have the **generic** name `'<lambda>'`.

Lambda expressions

```python
f = lambda x : x+'goat'
f('cat')
```

```
'catgoat'
```

```python
(lambda x : 2*x)(21)
```

```
42
```

```python
list(map(lambda x: x**2, range(1,10)))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Lambda expressions can be used anywhere you would use a function. Note that the term **anonymous function** makes sense: the lambda expression defines a function, but it never gets a variable name (unless we assign it to something, like in the `'goat'` example to the left).

Assigning function to a variable

```python
f = lambda x : x+'goat'
f('cat')
```

```
'catgoat'
```

```python
def my_square(x):
    return(x**2)
my_square
```

```
<function __main__.my_square>
```

The fact that we can have variables whose values are functions is actually quite special. We say that Python has **first-class functions**. That is, functions are perfectly reasonable values for a variable to have.

You’ve seen these ideas before if you’ve used R’s `tapply` (or similar), MATLAB’s function handles, C/C++ function pointers, etc.

Quantifiers over iterables: `any()` & `all()`

```
any([False,True,False])
```

```
True
```

`any` takes an iterable as its input and returns `True` if and only if one or more elements is `True`.

```
any((0,'',0.0))
```

```
False
```

Reminder: 0, 0.0, empty string, empty list, etc all evaluate to `False`. Just about everything else evaluates to `True`.

```
all([(1,0),1,'cat'])
```

```
True
```

`all` takes an iterable as its input and returns `True` if and only if all elements are `True`.

```
all(map(is_even,fibo))
```

```
False
```

Quantifiers over iterables: `any()` & `all()`

Complicated functions become elegant one-liners!

```
def is_prime(n):
    return not any((n%x==0 for x in range(2,n)))
is_prime(8675309)
```

```
True
```

```
is_prime(8675310)
```

```
False
```

Of course, sometimes that elegance comes at the cost of efficiency. In this example, we're failing to use a speedup that would be gained from using, e.g., the sieve of Eratosthenes and stopping checking above `sqrt(n)`.

https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

1. Iterator and Generators

2. Exceptions and Assertions

Unexpected conditions

What happens when procedure execution hits an unexpected condition?

- Got an **exception**… to what was expected.

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

```
---------------------------------------------------------------------------
IndexError
Cell In[43], line 2
      1 l = [1,2,3]
----> 2 l[4]

IndexError: list index out of range
```

```
int('exception')
```

```
---------------------------------------------------------------------------
ValueError
Cell In[45], line 1
----> 1 int('exception')

ValueError: invalid literal for int()
```

```
a
```

```
---------------------------------------------------------------------------
NameError
Cell In[47], line 1
----> 1 a

NameError: name 'a' is not defined
```

```
"str"/2
```

```
---------------------------------------------------------------------------
TypeError
Cell In[49], line 1
----> 1 "str"/2

TypeError: unsupported operand type(s)
```

Handling exceptions

Instead of crash, we can **handle** exceptions using:

```
try:
    # do some potentially
    # problematic code
except:
    # do something to
    # handle the problem
```

```
if <all potentially problematic code succeeds>:
    # great, all that code
    # just ran fine!
else:
    # do something to
    # handle the problem
```

Exception handler in Python

- If code in `try` block all succeed, `except` block will not be executed
- Exceptions raised by any statement in body of `try` are handled by the `except` statement

Slides credit to MIT open course MIT6,100L

Example: without exception handling

Python will crash immediately

```python
def divide_numbers(a, b):
    result = a / b
    return result
print(divide_numbers(5,0))
print("Done")
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[73], line 4
      2     result = a / b
      3     return result
----> 4 print(divide_numbers(5,0))
      5 print("Done")

Cell In[73], line 2, in divide_numbers(a, b)
      1 def divide_numbers(a, b):
----> 2     result = a / b
      3     return result

ZeroDivisionError: division by zero
```

Example: with exception handling

Python will be able to handle exception even though code in try block is problematic.

```
def divide_numbers(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
print(divide_numbers(5,0))
print("Done")
```

We do not have to specify the Error.

```
Error: Division by zero is not allowed.
Done
```

Example: can catch different errors

Can have separate except clauses to deal with a particular type of exception.

```
def divide_numbers(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
    except TypeError:
        return "Error: Please pass numeric values only."
    except:
        return "Something else went wrong..."

divide_numbers(3, 'a string')
```

Only execute if this kind of error is raised in `try` block

Can associate other blocks with `try` block

Besides `except` blocks

- `else`
  - Body will always be executed **when** `try` block competes with no exceptions
- `finally`
  - Body will always be executed, regardless of whether an exception was raised or not
  - Useful for cleanup actions
    - Ex: close a file…

```python
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."
    else:
        return f"The result is: {result}"
    finally:
        print("Execution complete.")
```

```
Execution complete.
'The result is: 2.0'
```

Assertions: a defensive programming tool

It relates to an assumption on the state of computation are as expected.

Use as assert statement to raise an `AssertionError` exception if assumptions are not met

```
assert <statement that should evaluate to be true>, “statement not true”
```

Assertion usage

To use as good **defensive programming**

- Check **inputs** to functions, but can be used anywhere
- Check **outputs** to functions to avoid propagating bad values
- Can make it easier to **debug**.

```python
def divide(a, b):
    assert b != 0, "Division by zero is not allowed"
    return a / b

# Test the function
print(divide(10, 2))  # This will work fine
print(divide(10, 0))  # This will raise an AssertionError
```

```python
class Person:
    def __init__(self, age):
        assert age > 0, "Age must be positive"
        self.age = age

# Create a Person object
p = Person(25)  # Works fine
p = Person(-5)  # Raises AssertionError
```

Other things

HW3 due this Friday.

HW4 is out today.

Coming next: