STATS 507
Data Analysis in Python

Week 4: Object-Oriented Programming, Iterators, and Generators

Adapted from slides by Keith Levin and Charles Severance

Objects are everywhere in Python

5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
    Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*L*)
    Extend the list by appending all the items in the given list. Equivalent to `a[len(a):] = L`.

`list.`**`insert`**(*i*, *x*)
    Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)
    Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`pop`**([*i*])
    Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

Image credit:
Charles
Severence

https://docs.python.org/3/tutorial/datastructures.html

Data scientists frequently use objects (even if they don't write new classes)

12.6. sqlite3 — DB-API 2.0 interface for SQLite databases

Source code: Lib/sqlite3/

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**.

To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the `example.db` file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')
```

Image credit: Charles Severence

Classes: programmer-defined types

Sometimes we use a collection of variables to represent a specific object

- **Example:** we used a tuple of tuples to represent a matrix
- **Example:** representing state of a board game
  - List of players, piece positions, etc.
- **Example:** representing a statistical model
  - Want to support methods for estimation, data generation, etc.

Important point: these data structures quickly become very complicated, and we want a way to encapsulate them. This is a core motivation (but hardly the only one) for **object-oriented programming**.

Classes encapsulate data types

Example: I want to represent a point in 2-dimensional space $\mathbb{R}^2$

Option 1: just represent a point by a 2-tuple

Option 2: make a point **class**, so that we have a whole new data type

Additional good reasons for this will become apparent shortly!

```
class Point:
    '''Represents a 2-d point.'''
```

Class header declares a new class, called `Point`.

Docstring provides explanation of what the class represents, and a bit about what it does. This is an ideal place to document your class.

```
print(Point)
```

```
<class '__main__.Point'>
```

Credit: Running example adapted from A. B. Downey, *Think Python*

Classes encapsulate data types

Note: By convention, class names are written in **CamelCase**.

Example: I want to represent a point in 2-dimensional space $\mathbb{R}^2$

Option 1: just represent a point by a 2-tuple

Option 2: make a point **class**, so that we have a whole new data type

Additional good reasons for this will become apparent shortly!

```
class Point:
    '''Represents a 2-d point.'''
```

```
print(Point)
```

Class definition creates a **class object**, `Point`.

```
<class '__main__.Point'>
```

Credit: Running example adapted from A. B. Downey, *Think Python*

Creating an object: Instantiation

```python
class Point:
    '''Represents a 2-d point.'''

p = Point()
p
```

```
<__main__.Point at 0x10669b940>
```

This defines a class `Point`, and from here on we can create new variables of type `Point`.

Creating an object: Instantiation

```
class Point:
    '''Represents a 2-d point.'''

p = Point()
p
```

```
<__main__.Point at 0x10669b940>
```

Creating a new object is called **instantiation**. Here we are creating an **instance** `p` of the class `Point`.

Indeed, `p` is of type `Point`.

Note: An **instance** is an individual object from a given class.
In general, the terms **object** and **instance** are interchangeable: an object is an instantiation of a class.

Assigning Attributes

This dot notation should look familiar. Here, we are assigning values to **attributes** `x` and `y` of the object `p`. This both creates the attributes, and assigns their values.

Once the attributes are created, we can access them, again with dot notation.

```
p = Point()
p.x = 3.0
p.y = 4.0
(p.x,p.y)
```

```
(3.0, 4.0)
```

Attempting to access an attribute that an object doesn't have is an error.

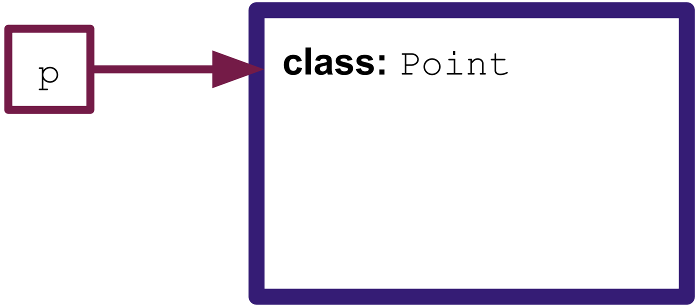
```
p.goat
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-f74ee22f01ba> in <module>()
----> 1 p.goat

AttributeError: 'Point' object has no attribute 'goat'
```

Thinking about Attributes: Object Diagrams

```
class Point:
    '''Represents a 2-d point.'''

p = Point()
p.x = 3.0
p.y = 4.0
```

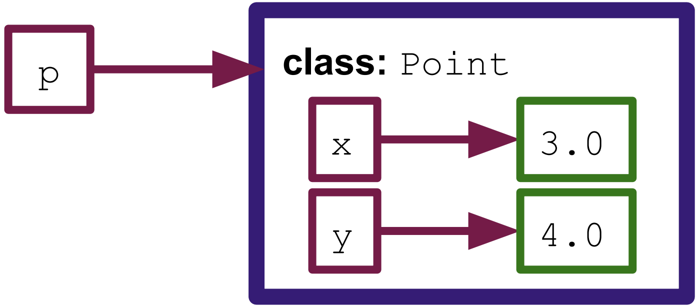
At this point, `p` is just an object with no attributes.



Thinking about Attributes: Object Diagrams

```python
class Point:
    '''Represents a 2-d point.'''

p = Point()
p.x = 3.0
p.y = 4.0
```

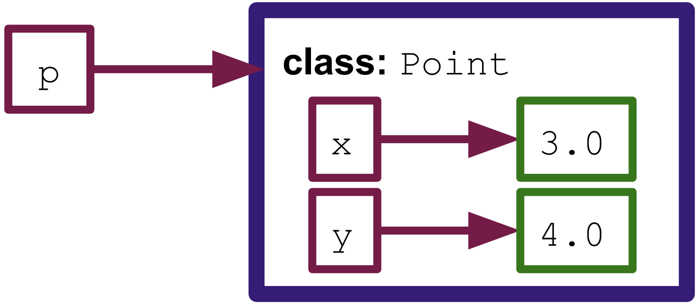
After these two lines, `p` has attributes `x` and `y`.



Thinking about Attributes: Object Diagrams

```python
class Point:
    '''Represents a 2-d point.'''

p = Point()
p.x = 3.0
p.y = 4.0
```

After these two lines, `p` has attributes `x` and `y`.

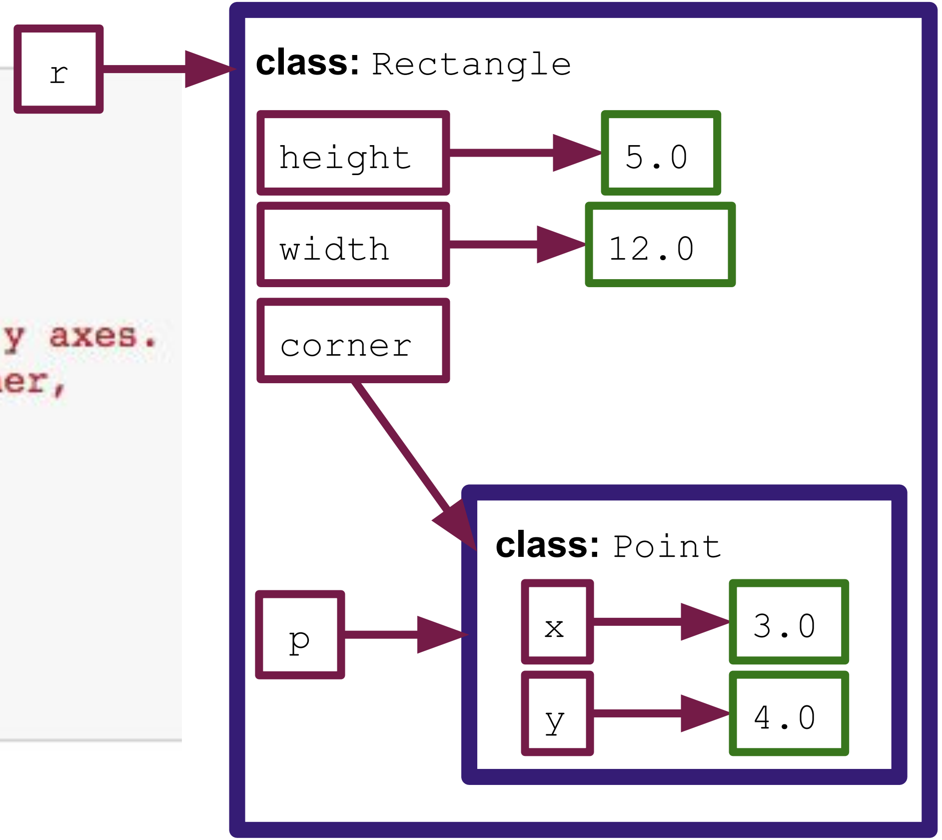


So dot notation `p.x`, essentially says, look inside the object `p` and find the attribute `x`.

Nesting Objects

Objects can have other objects as their attributes.
We often call the attribute object **embedded**.

```python
class Point:
    '''Represents a 2-d point.'''

class Rectangle:
    '''Represents a rectangle whose
    sides are parallel to the x and y axes.
    Specified by its upper-left corner,
    height, and width.'''

p = Point(); p.x = 3.0; p.y = 4.0
r = Rectangle()
r.corner = p
r.height = 5.0
r.width = 12.0
```



Nesting Objects

```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0
r1.width = 12.0

r2 = Rectangle()
r2.corner = Point()
r2.corner.x = 3.0
r2.corner.y = 4.0
r2.height = 5.0
r2.width = 12.0
```

Both of these blocks of code create equivalent `Rectangle` objects.

Note here that instead of creating a point and then embedding it, we embed a `Point` object and *then* populate its attributes.

Objects are mutable

```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0; r1.width = 12.0
r1.height = 2*r1.height

def shift_rectangle(rec, dx, dy):
    rec.corner.x = rec.corner.x + dx
    rec.corner.y = rec.corner.y + dy

shift_rectangle(r1, 2, 3)
(r1.corner.x, r1.corner.y)
```

```
(5.0, 7.0)
```

If my `Rectangle` object were immutable, this line would be an error, because I'm making an assignment.

Since objects are mutable, I can change attributes of an object inside a function and those changes remain in the object in the `__main__` namespace.

Returning Objects

```
def double_sides(r):
    rdouble = Rectangle()
    rdouble.corner = r.corner
    rdouble.height = 2*r.height
    rdouble.width = 2*r.width
    return(rdouble)

p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0
r1.width = 12.0

r2 = double_sides(r1)
r2.height, r2.width
```

```
(10.0, 24.0)
```

Functions can return objects. Note that this function is implicitly assuming that `rdouble` has the attributes `corner`, `height` and `width`. We will see how to do this soon.

The function creates a *new* Rectangle and returns it. Note that it doesn't change the attributes of its argument.

Copying and Aliasing

Recall that aliasing is when two or more variables have the same referent

i.e., when two variables are identical

Aliasing can often cause unexpected problems

Solution: make **copy** of object; variables equivalent, but not identical

```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
import copy
p2 = copy.copy(p1)
p1 is p2
```

```
False
```

The `copy` module provides functions for copying objects. `p2` is a copy of `p1`, so they should **not** be identical...

```
p1 == p2
```

```
False
```

...but they should **be** equivalent.

Copying and Aliasing

Documentation for the `copy` module:
https://docs.python.org/3/library/copy.html

Recall that aliasing is when two or more variables have the same referent

i.e., when two variables are identical

Aliasing can often cause unexpected problems

Solution: make **copy** of object; variables equivalent, but not identical

```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
import copy
p2 = copy.copy(p1)
p1 is p2
```

```
False
```

The
cop
they

```
p1 == p2
```

```
False
```

...bu

Hey, those were supposed to be equivalent! What's up with that? **Answer:** by default, for programmer-defined types, `==` and `is` are the same. It's up to you, the programmer, to tell Python how to tell if two objects are equivalent, by defining a method `object.__eq__`. We'll come back to this.

Copying and Aliasing

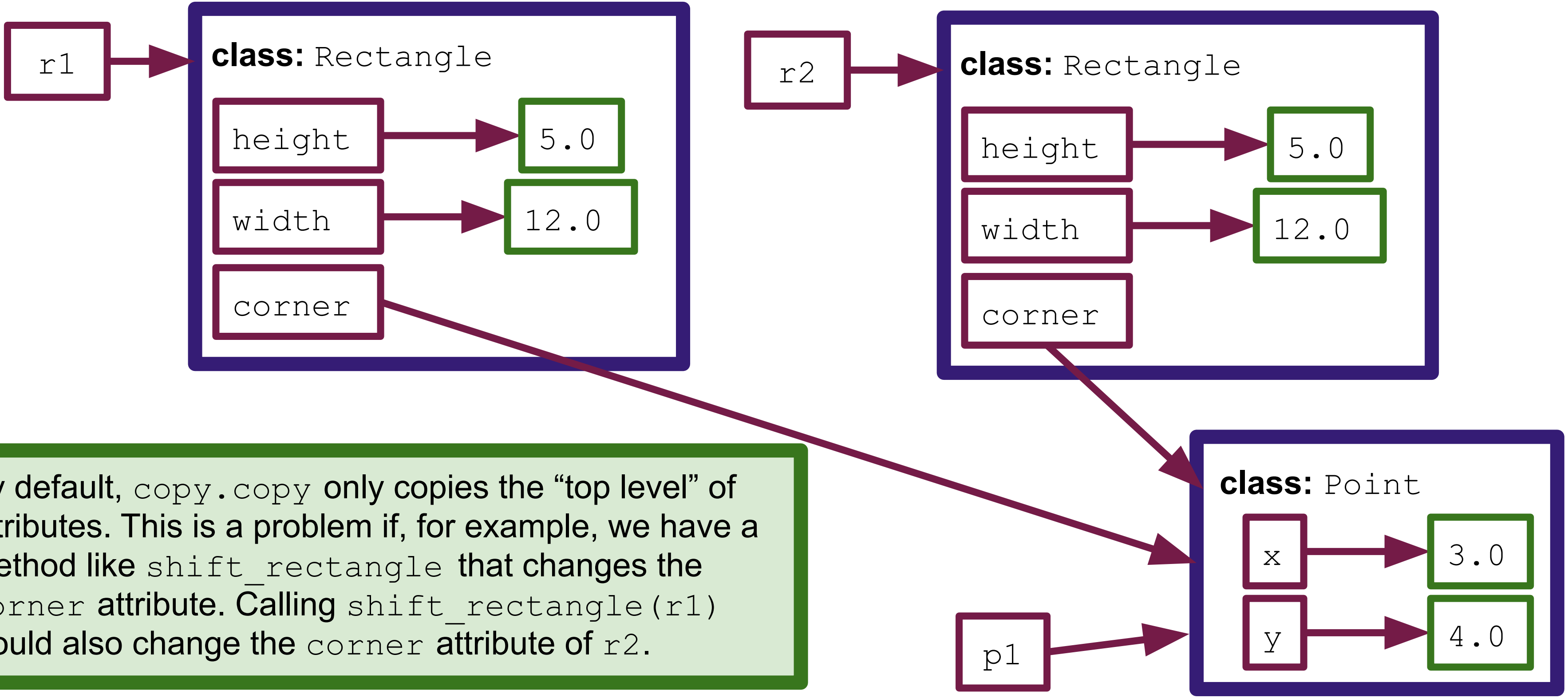
```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0; r1.width = 12.0
r2 = copy.copy(r1)

r1.corner is r2.corner
```

```
True
```

Here we construct a Rectangle, and then copy it. Expected behavior is that mutable attributes should **not** be identical, and yet...

...evidently our copied objects still have attributes that are identical.

Copying and Aliasing



By default, `copy.copy` only copies the “top level” of attributes. This is a problem if, for example, we have a method like `shift_rectangle` that changes the `corner` attribute. Calling `shift_rectangle(r1)` would also change the `corner` attribute of `r2`.

Copying and Aliasing

```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0; r1.width = 12.0
r2 = copy.deepcopy(r1)

r1.corner is r2.corner
```

```
False
```

`copy.deepcopy` is a recursive version of `copy.copy`. So it recursively makes copies of all attributes, and their attributes and so on.

We often refer to `copy.copy` as a **shallow copy** in contrast to `copy.deepcopy`.

Now when we test for identity we get the expected behavior. Python has created a copy of `r1.corner`.

`copy.deepcopy` documentation explains how the copying operation is carried out: https://docs.python.org/3/library/copy.html#copy.deepcopy

Pure functions vs modifiers

A **pure function** is a function that returns an object

...and **does not** modify any of its arguments

A **modifier** is a function that changes attributes of one or more of its arguments

```
def double_sides(r):
    rdouble = Rectangle()
    rdouble.corner = r.corner
    rdouble.height = 2*r.height
    rdouble.width = 2*r.width
    return(rdouble)

def shift_rectangle(rec, dx, dy):
    rec.corner.x = rec.corner.x + dx
    rec.corner.y = rec.corner.y + dy
```

`double_sides` is a **pure function**. It creates a new object and returns it, without changing the attributes of its argument `r`.

`shift_rectangle` changes the attributes of its argument `rec`, so it is a **modifier**. We say that the function has **side effects**, in that it causes changes outside its scope.

https://en.wikipedia.org/wiki/Side_effect_(computer_science)

Pure functions vs modifiers

Why should one prefer one over the other?

Pure functions

Are often easier to debug and verify (i.e., check correctness)

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

Common in **functional programming**

Modifiers

Often faster and more efficient

Common in **object-oriented programming**

Modifiers vs Methods

A modifier is a **function** that changes attributes of its arguments

A **method** is *like* a function, but it is provided by an object.

Define a class representing a 24-hour time.

```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

t = Time()
t.hours=12; t.mins=34; t.secs=56
t.print_time()
```

```
12:34:56
```

Class supports a **method** called `print_time`, which prints a string representation of the time.

Every method must include `self` as its first argument. The idea is that the object is, in some sense, the object on which the method is being called.

Credit: Running example adapted from A. B. Downey, *Think Python*

More on Methods

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

    def time_to_int(self):
        return(self.secs + 60*self.mins + 3600*self.hours)

def int_to_time(seconds):
    '''Convert a number of seconds to a Time object.'''
    t = Time()
    (minutes, t.secs) = divmod(seconds,60)
    (hrs, t.mins) = divmod(minutes,60)
    t.hours = hrs % 24 #military time!
    return t

t = int_to_time(1337)
t.time_to_int()
```

```
1337
```

`int_to_time` is a pure function that creates and returns a new `Time` object.

`Time.time_to_int` is a method, but it is still a pure function in that it has no side effects.

More on Modifiers

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def increment_pure(self, seconds):
        '''Return new Time object representing this time
        incremented by the given number of seconds.'''
        t = Time()
        t = int_to_time(self.time_to_int() + seconds)
        return t

    def increment_modifier(self, seconds):
        '''Increment this time by the given
        number of seconds.'''
        (mins, self.secs) = divmod(self.secs+seconds, 60)
        (hours, self.mins) = divmod(self.mins+mins, 60)
        self.hours = (self.hours + hours)%24

t1 = int_to_time(1234)
t1.increment_modifier(1111)
t1.time_to_int()
```

```
2345
```

I cropped out `time_to_int` and `print_time` for space.

Two different versions of the same operation. One is a pure function (pure method?), that does not change attributes of the caller. The second method is a modifier.

The modifier method does indeed change the attributes of the caller.

More on Modifiers

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''
    def time_to_int(self):
        return(self.secs + 60*self.mins + 3600*self.hours)
    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

    def increment_pure(self, seconds):
        '''Return new Time object representing this time
        incremented by the given number of seconds.'''
        t = Time()
        t = int_to_time(self.time_to_int() + seconds)
        return t

t1.increment_pure(100, 200)
```

Here's an error you may encounter. How the heck did `increment_pure` get 3 arguments?!

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-1d8fb5e5c628> in <module>()
     14         return t
     15
---> 16 t1.increment_pure(100, 200)

TypeError: increment_pure() takes 2 positional arguments but 3 were given
```

Answer: the caller is considered an argument (because of `self`)!

Recap: Objects, so far

So far: creating classes, attributes, methods

Next steps:

- How to implement operators (+, *, string conversion, etc)
- More complicated methods
- Inheritance

We will not come anywhere near covering OOP in its entirety

- My goal is only to make sure you see the general concepts
- Take a software engineering course to learn the deeper principles of OOP

Creating objects: the `__init__` method

```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def __init__(self, hours=0, mins=0, secs=0):
        self.hours = hours
        self.mins = mins
        self.secs = secs

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

t = Time(); t.print_time()
```

```
00:00:00
```

```
t = Time(10); t.print_time()
```

```
10:00:00
```

```
t = Time(10,20); t.print_time()
```

```
10:20:00
```

`__init__` is a special method that gets called when we instantiate an object. This one takes four arguments.

If we supply fewer than three arguments to `__init__`, it defaults the extras, assigning from left to right until it runs out of arguments.

Note: arguments that are not keyword arguments are called **positional arguments**.

Creating objects: the `__init__` method

```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def __init__(self, hours=0, mins=0, secs=0):
        self.hours = hours
        self.mins = mins
        self.secs = secs

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

t = Time(); t.print_time()
```

```
00:00:00
```

```
t = Time(10); t.print_time()
```

```
10:00:00
```

```
t = Time(10,20); t.print_time()
```

```
10:20:00
```

Important point: notice how much cleaner this is than creating an object and then assigning attributes like we did earlier. Defining an `__init__` method also lets us ensure that there are certain attributes that are **always** populated in an object. This avoids the risk of an `AttributeError` sneaking up on us later. **Best practice** is to create all of the attributes that an object is going to have **at initialization**. Once again, Python allows you to do something, but it's best never to do it!

While we're on the subject...

Useful functions to know for debugging purposes: `vars` and `getattr`

```
for attr in vars(t1):
    print(attr, getattr(t1,attr))
```

```
hours 11
mins 15
secs 10
```

`vars` returns a dictionary keyed on attribute names, values are attribute values.

This can be a useful pattern for debugging.

Objects to strings: the __str__ method

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def __init__(self, hours=0, mins=0, secs=0):
        self.hours = hours
        self.mins = mins
        self.secs = secs

    def __str__(self):
        return("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

t = Time(10,20,30)
print(t)
```

```
10:20:30
```

`__str__` is a special method that returns a string representation of the object. Print will always try to call this method via `str()`.

From the documentation: `str(object)` returns object.__str__(), which is the “informal” or nicely printable string representation of *object*. For string objects, this is the string itself. If *object* does not have a __str__() method, then str() falls back to returning repr(object).
https://docs.python.org/3.5/library/stdtypes.html#str

Overloading operators

We can get other operators (+, *, /, comparisons, etc) by defining special functions

```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def time_to_int(self):
        return(self.secs + 60*self.mins + 3600*self.hours)

    def __add__(self, other):
        '''Add other to this time, return result.'''
        s = self.time_to_int() + other.time_to_int()
        return(int_to_time(s))

t1 = Time(11,15,10); t2 = Time(1,5,1)
print(t1+t2)
```

```
12:20:11
```

`__init__` and `__str__` cropped for space.

Defining the `__add__` operator lets us use + with `Time` objects. This is called **overloading** the + operator. All operators in Python have special names like this. More information: https://docs.python.org/3/reference/datamodel.html#specialnames

Type-based dispatch

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def __add__(self, other):
        '''Add other to this time, return result.'''
        if isinstance(other,Time):
            s = self.time_to_int() + other.time_to_int()
            return(int_to_time(s))
        elif isinstance(other,int):
            s = self.time_to_int() + other
            return(int_to_time(s))
        else:
            raise TypeError('Invalid type.')

t1 = Time(11,15,10)
print(t1 + 60)
```

```
11:16:10
```

Other methods cropped for space.

`isinstance` returns `True` iff its first argument is of the type given by its second argument.

Depending on the type of `other`, our method behaves differently. This is called **type-based dispatch**. This is in keeping with Python's general approach of always trying to do something sensible with inputs.

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def __add__(self, other):
        '''Add other to this time, return result.'''
        if isinstance(other,Time):
            s = self.time_to_int() + other.time_to_int()
            return(int_to_time(s))
        elif isinstance(other,int):
            s = self.time_to_int() + other
            return(int_to_time(s))
        else:
            raise TypeError('Invalid type.')

t1 = Time(11,15,10)
print(60 + t1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-18f9bcbbe091> in <module>()
     26
     27 t1 = Time(11,15,10)
---> 28 print(60 + t1)

TypeError: unsupported operand type(s) for +: 'int' and 'Time'
```

Our + operator isn’t commutative! This is because `int + Time` causes Python to call the `int.__add__` operator, which doesn’t know how to add a `Time` to an `int`. We have to define a `Time.__radd__` operator for this to work.

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def __add__(self, other):
        '''Add other to this time, return result.'''
        if isinstance(other,Time):
            s = self.time_to_int() + other.time_to_int()
            return(int_to_time(s))
        elif isinstance(other,int):
            s = self.time_to_int() + other
            return(int_to_time(s))
        else:
            raise TypeError('Invalid type.')

t1 = Time(11,15,10)
print(60 + t1)
```

Our + operator isn't commutative! This is because `int + Time` causes Python to call the `int.__add__` operator, which doesn't know how to add a `Time` to an `int`. We have to define a `Time.__radd__` operator for this to work.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (m
<ipython-input-10-18f9bcbbe091> in <module>()
     26
     27 t1 = Time(11,15,10)
---> 28 print(60 + t1)

TypeError: unsupported operand type(s) for +: 'int' and 'Time'
```

Simple solution:

```python
def __radd__(self, other):
    return self.__add__(other)
```

Polymorphism

Type-based dispatch is useful, but tedious

Better: write functions that work for many types

Examples:

String functions often work on tuples

int functions often work on floats or complex

Functions that work for many types are called **polymorphic.** Polymorphism is useful because it allows code reuse.

`hist` below is a good example of polymorphism. Works for all sequences!

```python
def hist(s):
    h = dict()
    for x in s:
        h[x] = h.get(x,0)+1
    return h

hist('apple')
```

```
{'a': 1, 'e': 1, 'l': 1, 'p': 2}
```

```python
hist((1,1,2,3,5,8))
```

```
{1: 2, 2: 1, 3: 1, 5: 1, 8: 1}
```

```python
hist(list('gattaca'))
```

```
{'a': 3, 'c': 1, 'g': 1, 't': 2}
```

Interface and Implementation

Key distinction in object-oriented programming

Interface is the set of methods supplied by a class

Implementation is how the methods are actually carried out

Important point: ability to change implementation **without** affecting interface

Example: our `Time` class was represented by hour, minutes and seconds

Could have equivalently represented as seconds since midnight

In either case, we can write all the same methods (addition, conversion, etc)

Certain implementations make certain operations easier than others.
Example: comparing two times in our hours, minutes, seconds representation is complicated, but if `Time` were represented as seconds since midnight, comparison becomes trivial. On the other hand, printing hh:mm:ss representation of a `Time` is complicated if our implementation is seconds since midnight.

Inheritance

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

Inheritance allows us to create new Classes from old ones

Running example:

Objects are playing cards, hands and decks

Assumes some knowledge of Poker https://en.wikipedia.org/wiki/Poker

52 cards in a deck

4 suits: Spades > Hearts > Diamonds > Clubs

13 ranks: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King

Creating our class

We will encode suits and ranks by numbers, rather than strings. This will make comparison easier.

A card is specified by its suit and rank, so those will be the attributes of the card class. The default card will be the two of clubs.

```
class Card:
    '''Represents a playing card'''
    def __init__(suit=0,rank=2):
        self.suit = suit
        self.rank = rank
```

This stage of choosing how you will represent objects (and what objects to represent) is often the most important part of the coding process. It's well worth your time to carefully plan and design your objects, how they will be represented and what methods they will support.

Suit encoding
0 : Clubs
1 : Diamonds
2 : Hearts
3 : Spades

Rank encoding
0 : None
1 : Ace
2 : 2
3 : 3
…
10 : 10
11 : Jack
12 : Queen
13 : King

Creating our class

```
class Card:
    '''Represents a playing card'''

    suit_names = ['Spades','Hearts','Diamonds','Clubs']
    rank_names = [None,'Ace','2','3','4','5','6','7',
                  '8','9','10','Jack','Queen','King']

    def __init__(self,suit=0,rank=2):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        rankstr = self.rank_names[self.rank]
        suitstr = self.suit_names[self.suit]
        return("%s of %s" % (rankstr, suitstr))

print(Card(0,1))
```

```
Ace of Spades
```

Variables defined in a class but outside any method are called **class attributes**. They are shared across all instances of the class.

Instance attributes are assigned to a specific object (e.g., `rank` and `suit`). Both class and instance attributes are accessed via dot notation.

Here we use instance attributes to index into class attributes.

More operators

```
class Card:
    '''Represents a playing card'''

    def __lt__(self, other):
        t1 = (self.rank, self.suit)
        t2 = (other.rank, other.suit)
        return t1 < t2

    def __gt__(self, other):
        return other < self

    def __eq__(self, other):
        return(self.rank==other.rank and self.suit==other.suit)
c1 = Card(2,11); c2 = Card(2,12)
c1 < c2
```

```
True
```

Cropped for space.

We've chosen to order cards based on rank and then suit, with aces low. So a jack is bigger than a ten, regardless of the suit of either one. Downey orders by suit first, then rank.

```
c1 == Card(2,11)
```

```
True
```

Now that we've defined the `__eq__` operator, we can check for equivalence correctly.

Objects with other objects

```
class Deck:
    '''Represents a deck of cards'''
    def __init__(self):
        self.cards = list()
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit,rank)
                self.cards.append(card)

    def __str__(self):
        res = list()
        for c in self.cards:
            res.append(str(c))
        return('\n'.join(res))

d = Deck()
print(d)
```

Define a new object representing a deck of cards. A standard deck of playing cards is 52 cards, four suits, 13 ranks per suit, etc.

Represent cards in the deck via a list. To populate the list, just use a nested for-loop to iterate over suits and ranks.

String representation of a deck will just be the cards in the deck, in order, one per line. Note that this produces a **single string**, but it includes newline characters.

```
Ace of Spades
2 of Spades
3 of Spades
4 of Spades
5 of Spades
```

There's another 45 or so more strings down there...

Providing additional methods

```
import random
class Deck:
    '''Represents a deck of cards'''
```

```
    def pop_card(self):
        return(self.cards.pop())
    def add_card(self,c):
        self.cards.append(c)
    def shuffle(self):
        random.shuffle(self.cards)
```

One method for dealing a card off the “top” of the deck, and one method for adding a card back to the “bottom” of the deck.

Note: methods like this that are really just wrappers around other existing methods are often called **veneer** or **thin methods**.

```
d = Deck()
d.shuffle()
print(d)
```

```
2 of Hearts
9 of Clubs
Ace of Spades
3 of Clubs
6 of Spades
```

After shuffling, the cards are not in the same order as they were on initialization.

Let’s take stock

We have:

a class that represents playing cards (and some basic methods)

a class that represents a deck of cards (and some basic methods)

Now, the next logical thing we want is a class for representing a hand of cards

So we can actually represent a game of poker, hearts, bridge, etc.

The naïve approach would be to create a new class Hand from scratch

But a more graceful solution is to use **inheritance**

Key observation: a hand is a lot like a deck (it’s a collection of cards)

...of course, a hand is also different from a deck in some ways...

Inheritance

```
class Hand(Deck):
    '''Represents a hand of cards'''

h = Hand()
h.shuffle()
print(h)
```

```
Ace of Clubs
Queen of Diamonds
9 of Hearts
King of Hearts
8 of Clubs
8 of Hearts
Queen of Clubs
3 of Diamonds
5 of Hearts
7 of Clubs
King of Diamonds
```

This syntax means that the class `Hand` **inherits** from the class `Deck`. Inheritance means that `Hand` has all the same methods and class attributes as `Deck` does.

We say that the **child** class `Hand` inherits from the **parent** class `Deck`.

So, for example, `Hand` has `__init__` and `shuffle` methods, and they are identical to those in `Deck`. Of course, we quickly see that the `__init__` inherited from `Deck` isn't quite what we want for `Hand`. A hand of cards isn't usually the entire deck...

So we already see the ways in which inheritance can be useful, but we also see immediately that there's no free lunch here. We will have to **override** the `__init__` function inherited from `Deck`.

Inheritance: methods and overriding

```
class Hand(Deck):
    '''Represents a hand of cards'''

    def __init__(self, label=''):
        self.cards = list()
        self.label=label

h = Hand('new hand')
d = Deck(); d.shuffle()
h.add_card(d.pop_card())
print(h)
```

```
6 of Spades
```

Redefining the `__init__` method overrides the one inherited from `Deck`.

Simple way to deal a single card from the deck to the hand.

Inheritance: methods and overriding

```python
import random
class Deck:
    '''Represents a deck of cards'''

    def move_cards(self, hand, ncards):
        for i in range(ncards):
            hand.add_card(self.pop_card())
```

Encapsulate this pattern in a method supplied by `Deck`, and we have a method that deals cards to a hand.

```python
d = Deck(); d.shuffle()
h = Hand()
d.move_cards(h,5)
print(h)
```

```
2 of Spades
King of Spades
9 of Diamonds
2 of Diamonds
7 of Clubs
```

Note that this method is supplied by `Deck` but it modifies both the caller and the `Hand` object in the first argument.

Note: `Hand` also inherits the `move_cards` method from `Deck`, so we have a way to move cards from one hand to another (e.g., as at the beginning of a round of hearts)

Inheritance: pros and cons

Pros:

Makes for simple, fast program development

Enables code reuse

Can reflect some natural structure of the problem

Cons:

Can make debugging challenging (e.g., where did this method come from?)

Code gets spread across multiple classes

Can accidentally override (or forget to override) a method

A Final Note on OOP

- Object-oriented programming is ubiquitous in software development
- Useful when designing large systems with many interacting parts
- As a statistician, most systems you build are less complex
  - (At least not in the sense of requiring lots of interacting subsystems)
- We’ve only scratched the surface of OOP
  - Not covered: factories, multiple inheritance, abstract classes…
  - Take a software engineering course to learn more about this

Intermission

Next up:

iterators, generators, and more!

Iterators

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

Iterators

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

```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.

Iterators

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

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

```
1
```

```
next(titer)
```

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

```
2
```

```
next(titer)
```

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

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

StopIteration:
```

Iterators

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

```
1
```

```
next(titer)
```

```
2
```

```
next(titer)
```

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

StopIteration:
```

Lists are **not** iterators, but we can turn a list **into** an iterator by calling `iter()` on it. Thus, lists are **iterable**, meaning that it is possible to obtain an iterator over their elements. https://docs.python.org/3/glossary.html#term-iterable

From the documentation: "When an iterable object is passed as an argument to the built-in function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself. **The for statement does that automatically for you,** creating a temporary unnamed variable to hold the iterator for the duration of the loop."

Iterators

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.

Iterators

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

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

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

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`.

```
---------------------------------------------------------------------------
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
```

Iterators

```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
```

Iterators

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

```
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)
```

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

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

This is an infinite loop.

Iterators

```
t1 = ['cat','dog','bird','goat']
t1_iter = iter(t1)
t2 = list(t1_iter)
t1 == t2
```

```
True
```

```
t1 is t2
```

```
False
```

We can turn an iterator *back* into a list, tuple, etc. **Caution:** if you have an iterator like our `Squares` example earlier, this list is infinite and you'll just run out of memory.

Many built-in functions work on iterators. e.g., `max`, `min`, `sum`, work on any iterator (provided elements support the operation); `in` operator will also work on any iterator

Warning: Once again, care must be taken if the iterator is infinite.

List Comprehensions and 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 of `Catalan` object `c`...

```
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

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.

Generators

```python
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.

```python
next(h)
```

```
1.0
```

```python
next(h)
```

```
1.5
```

```python
next(h)
```

```
1.8333333333333333
```

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
```

Generators

```python
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
next(h)
```

```
1.0
```

```python
next(h)
```

```
1.5
```

```python
next(h)
```

```
1.8333333333333333
```

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

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

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
```

```
next(h)
```

```
1.5
```

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

```
next(h)
```

```
1.8333333333333333
```

Generators

```python
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
next(h)
```

```
1.0
```

```python
next(h)
```

```
1.5
```

```python
next(h)
```

```
1.8333333333333333
```

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

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

Generators

```python
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
next(h)
```

```
1.0
```

```python
next(h)
```

```
1.5
```

```python
next(h)
```

```
1.8333333333333333
```

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

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).

Generators

Generators supply a few more bells and whistles

- Ability to pass values *into* the generator to modify behavior
- Can make generators both produce and consume information
  - **Coroutines** as opposed to **subroutines**

See generator documentation for more:

- https://docs.python.org/3/reference/expressions.html#generator-iterator-methods

`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
```

```
(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>>
```

```
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.

```
(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>
```

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

`lambda` expressions

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

```
'catgoat'
```

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

```
42
```

```
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).

First-class functions

```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()` and `all()`

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

```
True
```

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

```
False
```

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

```
True
```

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

```
False
```

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

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

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

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

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

```
True
```

```
is_prime(8675310)
```

```
False
```

Here's a nice example of why functional programming is useful. Complicated functions become elegant one-liners!

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

`zip`, revisited

```
h = harmonic()
c = Catalan()
z = zip(h,c)
z
```

```
<zip at 0x101c11a08>
```

```
[next(z) for x in range(10)]
```

```
[(1.0, 1.0),
 (1.5, 1.0),
 (1.8333333333333333, 2.0),
 (2.083333333333333, 5.0),
 (2.283333333333333, 14.0),
 (2.4499999999999997, 42.0),
 (2.5928571428571425, 132.0),
 (2.7178571428571425, 429.0),
 (2.8289682539682537, 1430.0),
 (2.9289682539682538, 4862.0)]
```

Recall that `zip` takes two or more iterables and returns an iterator over tuples

Here are two infinite iterators, and we `zip` them. So `z` should also be an infinite iterator. But this expression doesn't result in an infinite evaluation...

The trick is that `zip` uses **lazy evaluation**. Rather than trying to build all the tuples right when we call `zip`, Python is lazy. It only builds tuples as we ask for them! We'll see this plenty more in this course. https://en.wikipedia.org/wiki/Lazy_evaluation

Speaking of laziness

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

```
True
```

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

```
False
```

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

```
True
```

`any` and `all` are lazy. As soon as any finds a `True` element, it returns `True`. As soon as all finds a `False` element, it returns `False`. This is a simpler (i.e., less general) notion of laziness than lazy evaluation, but the underlying motivation is the same. Do as little work as is necessary to get your answer!

Next time,

`numpy`, `scipy` and `matplotlib`!