6.00.1.x - Week 1 - Python Basics

September 14, 2020

1 Video 1: Introduction

What do we want you to take away?

How to think computationally, algorithmically. How can I describe this problem in order to get the computer to do it?

- Data structures.
- Iteration and recursions as computational metaphors.
- Abstraction of procedures and data types.
- Organize and modularize systems using object classes and methods.
- Different classes of algorithms, searching and sorting.
- Complexity of algorithms.

What does a computer do?

1. Perform calculations.
    - Built in to the language.
    - Ones that you define as the programmer.
2. Remember results.

Are there limits?

- Some problems are too complex.
- Some problems are fundamentally impossible to compute.

2 Video 2: Knowledge

Types of knowledge

1. Declarative knowledge: statements of fact.

   Square root of a number x is y suh that y*y = x

2. Imperative knowledge: is a recipe or "how-to".

   Recipe for deducing square root of number x.

What is a recipe? (aka algorithm!)

1. Sequence of simple steps.
2. Flow of control process that specifies when each step is executed.
3. A means of determining when to stop.

3 Video 3: Machines

How to capture a recipe in a mechanical process?

1. Fixed program computer.

   Examples: Hand-held computer, Alan Turing's Bombe.

2. Stored program computer.

   Can load a program and execute its instructions.

Basic Machine Architecture

1. Memory – Data, or instructions (a program).
2. Input / outputs.
3. ALU: Arithmetic Logic Unit.

Works between the IO and the memory. Performs primitive operations

4. Control Unit.

Has a program counter. It reads each instruction and then adds one (to move to the 2nd instruction, and so on). Eventually a test might be reached, and depending on its result, the counter will be changed.

Stored Program Computer

1. Sequence of instructions stored inside computer.

   Built from predefined set of primitive instructions: logic, tests, moving data.

2. A special program (interpreter) executes each instruction in order.

   Uses tests to change flow of control.

   Stops when done.

Stored Program Computer

- Turing showed you can compute anything using 6 primitives.

  Example: right, left, scan, read, write, do nothing (Turing Machine).

- Modern programming languages provide more convenient set of primitives.
- Anything that's computable in one (Turing Complete) language is computable in any other (Turing Complete) programming language.

4 Video 4: Languages

How to create recipes?

- A programming languages provides primitive operations.
- **Expressions** are complex but legal combinations of primitives in a programming language.
- Expressions and computations have **values** and **meanings** in a programming language.

Primitive constructs

- English: words.
- Programming Language: numbers, strings, simple operators.

Syntax: Is this a legal sentence or not?

- "hi"5, isn't syntactically valid.
- 3.2*5 is syntactically valid.

Semantics: What does this expression evaluate to?

- Static semantics: "Which syntactically valid strings actually have a meaning?"
  - 3 + "hi", syntactically valid (primitive, operator, primitive), yet not semantically valid.
- In a programming language, a both syntactically and static semantically correct expression will have only **one** meaning.

Type of errors

- Syntactic errors: easily caught.
- Static semantic errors: can cause unpredictable behaviour, some languages check for them.
- No semantic errors but different meaning than what was originally inteded: these are the worst errors.

5 Video 5: Types

Program: Sequence of definitions and commands. Definitions are evaluated. Commands are executed.

- A program will either be directly typed in a **shell** or stored in a **file** that is read into the shell and evaluated.

Objects: programs manipulate data objects.

- Objects have a type that defines the kinds of the things programs can do to them.
- Objects are **scalar** (cannot be subdivided) or **non-scalar** (have internal structure that can be accessed).

Scalar Objects

- int - Represent integers.
- float - Represente real numbers.
- bool - Represent Boolean values True and False.
- NoneType - Special and has one value, None.
- Can use type() to see the type of an object.

Type Conversions (Cast)

- Can converb object of one type to another.
  - float(3) converts integer 3 to float 3.0
  - int(3.9) trumps float 3.9 to int 3

Interesting example

- 3+5 as an input, gives an out: 5.
- print(3+5) gives no out, since no value returned, just something printed.

"Any expression that is syntactically valid has a value, which is itself a type."

Operators on ints and floats

- Sum, difference and product, between i and j:
  - If both are ints, result is int.
  - If either or both are floats, result is float.
- Division i/j: result is float.
- Int Division i//j: result is int, quotient without reminder.

- i%j: the remainder when i is divided by j.
- i**j: i to the power of j.

6 Video 6: Variables

Equal sign is an **asignment** of a value to a variable name. An assignment binds name to value.

Interesting tip

radius = radius + 1, can be replaced by, radius +=1 .

7 Video 7: Operators and Branching

Comparison operators on int and float

- i>j
- i>=j
- i < j
- i <= j
- i == j, equality test, True if i=j
  - Interesting point about comparisons. Python interpreter compares two numbers by **promotion** both to the more complex type. So for example in the comparison 10 == 10.0, the value of the expression is True, since both are promoted to float and are therefore equal.
- i != j, inequality test, True if i not equal to j

Logic operators on bools

- not a, True if a is False.
- a and b, True if both are True.
- a or b, True if either or both are True.

Branched programs

- Indentation is important: each indented set of expressions denotes a block of instructions. This indentation also provides a visual structure that reflects the semantic structure of the program.
- Branching programs allow us to make choices and do different things. But still the case that at most, each statement gets executed once.

8 Video 8: Bindings

Variables / Bindings

- Name
  - Descriptive, meaningful.
  - Helps you re-read code.
  - Cannot be keywords (int, float, etc.).
- Value
  - Information stored.
  - Can be updated.

Example Needing to create a "temp" variable in order to exchange the values between two variables. Can't just do it without it.

9 Video 9: Strings

Types of variables: int, float, bool, string!

String: letters, special characters, spaces, digits. Enclosed in "" or ''.

Difference with Matlab treatment of strings: Python will let you "sum" two strings in order to concatenate them (exactly like strcat would do in Matlab).

Concept of overloading an operation

- When using addition between strings, we say we are 'overloading' it because we are telling the operation to do something different to the arguments due to their type.

Operations on strings

- Succesive concatenation: 3*'eric' outputs 'ericericeric'
  - This is yet another example of overloading an operation.
- Length: len('eric') outputs 4
- Indexing out of an object: 'eric'[1] outputs 'r'. Counting starts at zero. 'eric'[0] will output 'e'.
- Slice a string apart:
  - 'eric'[1:3] will output 'ri'
  - 'eric'[:3] will output 'eri'
  - 'eric'[1:] will output 'ric'
  - 'eric'[:] will output 'eric'

Other operators

- 'in' operator tests for collection membership (a 'collection' refers to a string, list, tuple or dictionary).
  - element in coll, evalues to True if elemenet is a member of the collection coll, and False otherwise.
- 'not in' is exactly the same but the value is the opposite.

Advanced String Slicing

- You can slice a string such a s[i:j], which gives you a portion of string s from index i to index j-1.
- You can also slice a string like this s[i:j:k], which gives a slice of the string s from index i to index j-1 with step size k.
  - A cool idea in this area is using the following, s[::-1], to print string s in its entirety but backwards!

10 Video 10: Input/Output

Print: used to **output** stuff to console.

[1]:
```
x = 1
print(x)
```

```
1
```

[2]:
```
x_str = str(x)
print("my fav num is", x, ".", "x = ", x)
```

```
my fav num is 1 . x =  1
```

Print is printing each element followed by a space.

[3]:
```
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

```
my fav num is 1. x = 1
```

Printing a concatenation of strings, letting me control the spaces more directly. Of course the possibility of choosing a separator works.

Input

- Input expects everything to be a string.

[4]:
```
text = input("type something'")
```

```
type something'7
```

[5]:
```
text
```

[5]:
```
'7'
```

The double quotes are there because I typed them in. Input expects everything to be a string, so if I don't want the quote marks, I don't need to add them there. But what happens if I want to input a number (and have it saved as type = int, for example)? I need to cast it.

[6]:
```
num = int(input("Type a number..."))
```

```
Type a number...4
```

[7]:
```
print(5*num)
```

```
20
```

11 Video 11: IDEs

- Integrated development enviroment (IDE) comes with:
  - Text editor.
  - Shell.
  - Integrated debugger.

12 Video 12: Control Flow

Branching program: Software that has at least two branches of different actions that will (or not) happen depending on the result of a test.

Up until now, with only if (and elif and else) statements, there is no way to implement an infinite 'loop' withot using infinite lines of code. Therefore, while loops are introduced.

While loops

- **condition** evaluates to a Boolean.
- If **condition** is True, do all the steps inside the while code block.
- Check **condition** again.
- Repeat until **condition** is False.

One can see that while loops can have some inconvenients, and that is what gives place to a more convenient kind of loop, called the **for loop**.

For loops

- Each time through the loop, **variable** takes a value.
- First time, **variable** starts at the smallest value.
- Next time, **variable** gets the prev value +1.

[8]:
```
for n in range(5):
    print(n)
```

```
0
1
2
3
4
```

- A useful tool within for loops, is the keyword **break**. It will stop the execution of the loop at that point, breaking out of it.

For loops vs while loops

- For loops have a known number of iterations, while loops have an unbounded number of iterations.
- Both loops can end early via **break**.
- For loops use a counter, while loops may use a counter.
- Can always rewrite a for loop using a while loop. May not be able to rewrite a while loop using a for loop.

Useful nomenclature for loops

a += b is equivalent to a = a + b

a -= b is equivalent to a = a - b

a = *b is equivalent to a = a* b

a /= b is equivalent to a = a / b

13 Video 13: Iteration

The same code can be used multiple times. First run a test, then run the loop body and keep running the test until the test evaluates to False. Some properties of iteration loops:

- Need to set an iteration variable outside the loop.
- Need to test variable to determine when done.
- Need to change variable within the loop, in addition to other work.

Information about the 'range' built-in function

- Form 1: range(stop) - 0,1,…,stop
- Form 2: range(start,stop) - start,start+1,…,stop
- Form 3: range(start, stop, stepsize). Ie: range(2,10,2) = 2,4,6,8.

14 Video 14: Guess and check

Guess and check methods are very useful, yet not always the most efficient solution to a given problem. It is based on a systematic approach at generating guesses to the answer of the problem.

- First version of the guess and check method. Created an iterative proposer of possible cubic roots of a given integer, by adding 1, trying whether it was or not the cubic root, and so on. Worked (for integers only), but slow and inefficient.

Useful to think about a **decrementing function**:

- Maps set of program variables into an integer.
- When loop is entered, value is non-negative.
- When value is <=0, loop terminates, and
- Value is decreased every time through loop

What can go wrong within a loop?

- The variable is not initialized. We are likely to get a **NameError**.
- The variable is not changed inside the loop. We get an **infinite loop** because it never reaches what is referred to as a **Terminating condition**.

So a **guess and check** algorithm:

- You are able to guess a value for solution.
- You are able to check if the solution is correct.
- Keep guessing until find solution or guessed all values.
- The process is exhaustive enumeration.

- **Second version** of the guess and check algorithm is created using a for loop.

6.00.1.x - Week 2 - Simple Programs

September 17, 2020

1 Video 1: So far

Reviewing strings:

- A sequence of case sensitive characters.
- Can compare them with ==, <, >, etc.
- len() is a function used to retrieve their **length**.
- Square brackets are used to perform **indexing** into a string to get the value at a certain index/position.
- Can **slice** strings using [start:stop:step].
- Strings are "**inmutable**" - cannot be modified.

[4]:
```
s = 'hello'
print(s)
type(s)
s[0]
```

```
hello
```

[4]:
```
'h'
```

[5]:
```
s[0] = 'y'
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-5-88dd6fabac11> in <module>
----> 1 s[0] = 'y'

TypeError: 'str' object does not support item assignment
```

[7]:
```
s = 'y' + s[1:]
print(s)
```

```
yello
```

2 Video 2: Approximate Solutions

Observations

1. Step could be any small number.
    1. If too small, takes a long time to find square root.
    2. If too large, might skip over answer without getting close enough.
2. In general, will take x/step times through code to find solution.
3. We need a more efficient way to do this.

3 Video 3: Bisection Search

At each stage, reduce range of values to search by half. This ia a great example of a logarithmic time algorithm finding an answer very quickly. This idea will be further developed later on in the course.

This should work on any problem with **ordering property** - value of function being solved varies monotonically with input value.

4 Video 4: Floats and Fractions

How do float represent real numbers? It is useful to get a sense of how the machine actually stores the numbers.

Internally, the computer represents every number in binary, whether it's an integer or a float. The algorithm of how it does it is slightly complex and was shown using two examples. Follow those if interested in the actual method.

Some implications

- If there is no integer p such that x*(2^p) is a whole number, then internal representation is always an approximation.
- Suggest that testing equality of floats is not exact:
  - Use abs(x-y) < some small number, rather than x == y.
- Why does print(0.1) return 0.1, if not exact?
  - Because Python designers set it up this way to automatically round.

5 Video 5: Newton-Raphson

- General approximation algorithm to find roots of a polynomial in one variable.
- Newton showed that if g is an approximation to the root of p, then g - p(g)/p'(g) is a better approximation, where p' is a derivate of p.

- So far we've seen the following methods of generating guesses:
  - Exhaustive enumeration.
  - Bisection search.
  - Newton-Raphson (for root finding).

6 Video 6: Decomposition and Abstraction

Good programming should be measured by the amount of functionality: the ability to make computations easily.

In order to become better programmers we're gonna introduce the concept of a function. But before that, we need to introduce two concepts: decomposition and abstraction.

Abstraction: do not need to know how something works in order to be able to use it.

```
* Suppress details of method to compute something from use of that computation.
```

Decomposition: different devices work together to achieve an end goal.

```
* Break problem into different, self-contained, pieces.
```

This lecture, we will achieve decomposition with **functions**. In a few weeks, achieve decomposition with **classes**.

We will achieve abstraction with **function specifications** or **docstrings**.

7 Video 7: Introducing Functions

- Characteristics
  - Name
  - Parameters (0 or more).
  - Docstring (optional but recommended)
  - Body.
- How to write it?

[3]:
```
def is_even(i):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("hi")
    return i%2 == 0
is_even(3)
```

```
hi
```

[3]:
```
False
```

Between """ we have the specification (docstring) of the function. Indented is the "body" of the function.

8 Video 8: Calling Functions and Scope

- **Formal parameters** gets bound to the value of **actual parameter** when function is called.
- New **scope/frame/environment** created when enter a function.
- **Scope** is a mapping of names to objects.

When a variable is changed within a function, it is changed in the function's scope but not in the global scope. Anytime a function is invoken, I create a new frame. Once I'm done with the body of the function, the frame is erased, because I no longer need it.

Return vs Print

- Return only has meaning **inside** a function, while print can be used **outside** functions.
- Only **one** return executed inside a function, while print can be executed many times.
- Code inside function but after return statement are not executed.
- Return has a value associated with it, **given to function caller**, while print has a value associated with it **outputted** to the console.

Arguments can take on any type, even functions.

Scope

- Inside a function, **can access** a variable defined outside.
- Inside a function, **cannot modify** a variable defined outside.

9 Video 9: Keyword Arguments

You may use an argument that changes how the function works. It branches out the functionalities provided by the function itself.

Interesting functionality: One may invoke the function inputting the parameters in a different order. Example: def fun(a,b,c): return a * b + c z = fun(b = 7, a = 5, c = 2)

Another interesting tool is being able to define a default value for a parameter such that, in case it is not explictly passed as an input when invoking the function, the parameter will take specifically that value.

10 Video 10: Specification

A **contract** between the implementer of a function and the clients who will use it.

- **Assumptions**: conditions that must be met by clients of the function; typically constraints on values of parameters.
- **Guarantees**: conditions that must be met by function providing it has been called in manner consistent with assumptions.

11 String Methods

11.1 Activity 13 of 27

String methods used:

- replace
- count
- find
- index
- swapcase
- capitalize
- islower
- isupper
- upper

https://docs.python.org/3/library/stdtypes.html#string-methods

12 Video 11: Iteration vs Recursion

Recursion

- A way to design solutions to problems by divide-and-conquer or decrease-and-conquer.
- A programming technique where a function **calls itself**.
- The goal is to NOT have infinite recursion:
  - Must have 1 or more base cases that are easy to solve.
  - Must solve the same problem on some other input with the goal of simplifying the larger problem input.

There's two structures that need to be identified in a problem that is solved by recursion:

1. Recursive step: how to reduce problem to a **smaller/simpler** version of itself.
2. Base case: Reduce the problem until you reach a simple case that can be **solved directly**.

[1]:
```
# Multiplication between 2 numbers
def mult(a,b):
    # Base case
    if b == 1:
        return a
    # Recursive step
    else:
        return a + mult(a,b-1)

mult(4,5)
```

[1]: 20

[3]:
```
# Factorial of a number n
def fact(n):
    # Base case
    if n ==1:
        return 1
    # Recursive step
    else:
        return n*fact(n-1)

fact(6)
```

[3]: 720

- Each recursive call to a function creates its **own scope/environment**.
- **Bindings of variables** in a scope is not changed by recursive call.
- Flow of control passes back to **previous scope** once function call returns value.

13 Video 12: Inductive Reasoning

How do we know that our recursive code will work?

- Invoking the recursive function with an ever decreasing input parameter.
- Another tool: Mathematical Induction. To prove a statement indexed on integers is true for all values of n:
  - Prove it is true when n is smallest value (e.g. n=0 or n=1).
  - Then prove that if it is true for an arbitrary value of n, one can show that it must be true for n+1.

14 Video 13: Towers of Hanoi

Somewhat complex problem that can be solved "pretty easily" by implementing recursive functions.

15 Video 14: Fibonacci

Can have recursion with multiple base cases too!

To generate the Fibonacci sequence:

- Base cases:
  - Females(0) = 1
  - Females(1) = 1
- Recursive case:
  - Females(n) = Females(n-1)+Females(n-2)

[1]:
```
def fib(x):
    """assumes x an int >= 0
       returns Fibonacci of x"""
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)

print(fib(7))
```

```
21
```

16 Video 15: Recursion on non-numerics

Solving recursively whether a string is or not a palindrome.

[1]:
```
def isPalindrome(s):

    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans

    def isPal(s):
        if len(s) <= 1:
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1])

    return isPal(toChars(s))

print("")
print('Is eve a palindrome?')
print(isPalindrome('eve'))

print('')
print('Is able was I ere I saw Elba a palindrome?')
print(isPalindrome('Able was I, ere I saw Elba'))
```

```
Is eve a palindrome?
True

Is able was I ere I saw Elba a palindrome?
True
```

17 Video 16: Files

Module: a .py file containing a collection of Python definitions of statements. Example: circle.py.

How to use a file?

1. Import it by writing: import *circle*
   - In order to avoid using circle.function1(arg) everytime, I can instead use the following command: from circle import *.
2. Open it by using the command open:

   nameHandle = open('filename','w') / ('w' for write or 'r' for read).

   Then one might write inside it by invoking the method nameHandle.write(whatever i want to write). Finally one might close the file by invoking the method nameHandle.close()

6.00.1.x - Week 3 - Structured Types

September 24, 2020

1 Video 1: Tuples

- An ordered sequence of elements, can mix element types.
- **Immutable**, cannot change element values.
- Represented with parentheses.

Something strange happens when you try to invoke part of the tuple.

[2]:
```
t = (2,"one",3)
t[0]
```

[2]:
```
2
```

[3]:
```
t[1:2]
```

[3]:
```
('one',)
```

[4]:
```
t[1:3]
```

[4]:
```
('one', 3)
```

- They can be conveniently used to swap variables:

(x,y) = (y,x)

This works perfectly well and no information is lost.

- Tuples can be used to return more than one value from a function, using the command **return** (q,r)
- Tuples can be used to iterate over them.

When a tuple has only one element, you must specify it as follows: (elt,).

2 Video 2: Lists

- Ordered sequence of information, accessible by index.
- A list is denoted by square brackets.
- A list contains elements (usually homogeneous, can contained mixed types too).
- List elements can be changed so a list is **mutable**.

- The index can be a **Variable or expression**, must evaluate to an int.
- I can iterate over a list too.

3 Video 3: List Operations

- **Add** elements to end of list with L.append(element). This **mutates** the list!

[7]:
```
L = [2,1,3]
L.append(5)
print(L)
```

```
[2, 1, 3, 5]
```

What is the dot?

- Lists are Python objects, everything in Python is an object.
- Objects have data.
- Objects have methods and functions.
- Access this information by object_name.do_something()

- Can also use **concatenation** with the + operator.

[9]:
```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
print(L3)
```

```
[2, 1, 3, 4, 5, 6]
```

- Can also **mutate** with the command extend.

[10]:
```
L1.extend([0,6])
print(L1)
```

```
[2, 1, 3, 0, 6]
```

- Can also **delete an element** with del(L[index]).
- Remove element at **end of list** with L.pop(), returns the removed element.

[12]:
```
L = [2,1,3,6,3,7,0]
L.remove(2)
print(L)
L.remove(3)
print(L)
del(L[1])
print(L)
L.pop()
print(L)
```

```
[1, 3, 6, 3, 7, 0]
[1, 6, 3, 7, 0]
```

```
[1, 3, 7, 0]
[1, 3, 7]
```

- Convert **string to list** with list(s), returns a list with every character from s an element in L.
- Can use s.split(), to split a string on a character parameter, splits on spaces if called without a parameter.

Other operations are: sort(), sorted(), reverse(), and many more.

https://docs.python.org/2/tutorial/datastructures.html

[13]:
```
# Difference between sort and sorted
L = [9,6,0,3]
sorted(L)
```

[13]:
```
[0, 3, 6, 9]
```

[14]:
```
L.sort()
print(L)
```

```
[0, 3, 6, 9]
```

[15]:
```
L.reverse()
print(L)
```

```
[9, 6, 3, 0]
```

- **range** is a special procedure:
  - Returns something that behaves like a tuple.
  - Doesn't generate all elements at once, rather it generates the first element, and provides an iteration method by which subsequent elements can be generated.
  - While in a foor loop, what the loop variable iterates over behaves like a list.

4 Video 4: Mutation, Aliasing, Cloning

Lists in memory

- Mutable.
- Behave differently than immutable types.
- Is an object in memory.
- Variable name points to object.
- Any variable pointing to that object is affected.
- Key phrase to keep in mind when working with lists is **side effects**.

We are gonna look into three ideas: first aliasing, then mutation, then cloning.

[16]:
```
warm = ['red','yellow','orange']
hot = warm
print(warm)
print(hot)
```

```
['red', 'yellow', 'orange']
['red', 'yellow', 'orange']
```

[17]:
```
hot.append('pink')
print(hot)
print(warm)
```

```
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```

hot is an **alias** for *warm.

Another concept: "If two lists print the same thing, that does not mean they are the same structure."

This is because there are 2 different slots in memory, one being accesed through each one of these handles.

[18]:
```
cool = ['blue','green','grey']
chill = ['blue','green','grey']
print(cool)
print(chill)
```

```
['blue', 'green', 'grey']
['blue', 'green', 'grey']
```

[19]:
```
cool.append('white')
print(cool)
```

```
print(chill)
```

```
['blue', 'green', 'grey', 'white']
['blue', 'green', 'grey']
```

Another idea in this realm is **cloning a list**, which creates a new list and **copies every element**.

[22]:
```
cool = ['blue','green','grey']
chill = cool[:]
print(cool)
print(chill)
```

```
['blue', 'green', 'grey']
['blue', 'green', 'grey']
```

[23]:
```
chill.append('black')
print(chill)
print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```

Sorting lists

- calling sort() **mutates** the list, returns nothing.
- calling sorted() **does not mutate** list, must assign result to a variable.

Careful with lists of lists!

[24]:
```
warm = ['yellow','orange']
hot = ['red']
brightcolors = [warm]
print(warm)
print(hot)
print(brightcolors)
```

```
['yellow', 'orange']
['red']
[['yellow', 'orange']]
```

Now let's add something to brightcolors

[25]:
```
brightcolors.append(hot)
print(brightcolors)
```

```
[['yellow', 'orange'], ['red']]
```

[26]:
```
hot.append('pink')
print(hot)
```

```
['red', 'pink']
```

[27]:
```
print(brightcolors)
```

```
[['yellow', 'orange'], ['red', 'pink']]
```

When I change hot, I also indirectly changed brightcolors. This is **aliasing** and we need to be careful about it.

[28]:
```
print(hot+warm)
```

```
['red', 'pink', 'yellow', 'orange']
```

Lastly, avoid mutating a list as you are iterating over it.

[29]:
```
def remove_dups(L1,L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
remove_dups(L1,L2)
print(L1)
```

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

Why was number 2 not deleted?

- Python uses an internal counter to keep track of the index it is in the loop.
- Mutating changes the list length but Python doesn't update the counter.
- Loop never sees element 2.

How do I solve this? I make a copy before mutating it.

[30]:
```
def remove_dups_new(L1,L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
remove_dups_new(L1,L2)
print(L1)
```

```
[3, 4]
```

5 Video 5: Functions as Objects

Functions are **first class objects**:

- Have types.
- Can be elements of data structures like lists.
- Can appear in expressions: part of an assignment statement, or as an argument to a function.

Particularly useful to use functions as arguments when coupled with lists, aka **higher order programming**.

[3]:
```
def applyToEach(L, f):
    """assumes L is a list, f a function
    mutates L by replacing each element,
    e, of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])

L = [1,-2,3.4]
applyToEach(L,abs)
print(L)
applyToEach(L,int)
print(L)
```

```
[1, 2, 3.4]
[1, 2, 3]
```

It could be done the other way too, applying a list of functions to a given argument. This idea can be generalized.

Python provides a general purpose HOP (higher order procedure), *map*.

[5]:
```
map(abs,[1,-2,3,-4])
# Produces an iterable
for elt in map(abs,[1,-2,3,-4]):
    print(elt)
```

```
1
2
3
4
```

Another example is the following:

[6]:
```
L1 = [1,28,36]
L2 = [2,57,9]
for elt in map(min,L1,L2):
    print(elt)
```

1
28
9

6 Video 6: Quick Review

Strings, tuples, ranges, lists

- Can get the ith element.
- Can get its length.
- Can concatenate them (not range).
- Can repeat them n times (not range).
- Can slice them.
- Can ask if an element is within them.
- Can iterate over its elements.
- Only lists are **mutable**.

7 Video 7: Dictionaries

Using multiple lists to keep information can be **messy**. It forces me to maintain many lists and use integers as indexes. It also forces me to remember to change multiple lists everytime I need to make a change.

Dictionaries

- Nice to index item of interest directly (not necesarily in *int*). We call the index *keys* when refering to dictionaries.
- Nice to use one data structure instead of many separate lists.
- Stores pairs of data - key & value.

[2]:
```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
print(grades['John'])
```

```
A+
```

[3]:
```
grades['Sylvan'] = 'A'
print(grades['Sylvan'])
```

```
A
```

[4]:
```
# Can test if a key is in dictionary
'John' in grades
```

[4]: True

[5]:
```
'Jorge' in grades
```

[5]: False

[6]:
```
# Remove an entry
del(grades['Ana'])
print(grades)
```

```
{'John': 'A+', 'Denise': 'A', 'Katy': 'A', 'Sylvan': 'A'}
```

[7]:
```
# Can also get the keys - Gives me an iterable!
grades.keys()
```

[7]: dict_keys(['John', 'Denise', 'Katy', 'Sylvan'])

[ ]:
```
# Can also get the values - Gives me an iterable!
grades.values()
```

Values

- Can be any type (immutable and mutable).
- Can be duplicates.
- Can be lists, even other dictionaries.

Keys

- Must be **unique**.
- **Immutable** type (int, float, string, tuple, bool).
- Careful with float type as a key.
- **No order** to keys or values!

8 Video 8: Example with a Dictionary

1. Create a frequency dictionary.
2. Find word that occurs the most and how many times.
3. Find the words that occur at least X times.

[12]:
```
def lyrics_to_frequencies(lyrics):
    myDict = {}
    for word in lyrics:
        if word in myDict:
            myDict[word] += 1
        else:
            myDict[word] = 1
    return myDict


she_loves_you = ['she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',

'you', 'think', "you've", 'lost', 'your', 'love',
'well', 'i', 'saw', 'her', 'yesterday-yi-yay',
"it's", 'you', "she's", 'thinking', 'of',
'and', 'she', 'told', 'me', 'what', 'to', 'say-yi-yay',

'she', 'says', 'she', 'loves', 'you',
'and', 'you', 'know', 'that', "can't", 'be', 'bad',
'yes', 'she', 'loves', 'you',
'and', 'you', 'know', 'you', 'should', 'be', 'glad',

'she', 'said', 'you', 'hurt', 'her', 'so',
'she', 'almost', 'lost', 'her', 'mind',
'and', 'now', 'she', 'says', 'she', 'knows',
"you're", 'not', 'the', 'hurting', 'kind',

'she', 'says', 'she', 'loves', 'you',
'and', 'you', 'know', 'that', "can't", 'be', 'bad',
'yes', 'she', 'loves', 'you',
'and', 'you', 'know', 'you', 'should', 'be', 'glad',

'oo', 'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'with', 'a', 'love', 'like', 'that',
'you', 'know', 'you', 'should', 'be', 'glad',

'you', 'know', "it's", 'up', 'to', 'you',
'i', 'think', "it's", 'only', 'fair',
```

```
'pride', 'can', 'hurt', 'you', 'too',
'pologize', 'to', 'her',

'Because', 'she', 'loves', 'you',
'and', 'you', 'know', 'that', "can't", 'be', 'bad',
'Yes', 'she', 'loves', 'you',
'and', 'you', 'know', 'you', 'should', 'be', 'glad',

'oo', 'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'with', 'a', 'love', 'like', 'that',
'you', 'know', 'you', 'should', 'be', 'glad',
'with', 'a', 'love', 'like', 'that',
'you', 'know', 'you', 'should', 'be', 'glad',
'with', 'a', 'love', 'like', 'that',
'you', 'know', 'you', 'should', 'be', 'glad',
'yeah', 'yeah', 'yeah',
'yeah', 'yeah', 'yeah', 'yeah'
]

beatles = lyrics_to_frequencies(she_loves_you)
print (beatles)
```

```
{'she': 20, 'loves': 13, 'you': 36, 'yeah': 28, 'think': 2, "you've": 1, 'lost':
2, 'your': 1, 'love': 5, 'well': 1, 'i': 2, 'saw': 1, 'her': 4, 'yesterday-yi-
yay': 1, "it's": 3, "she's": 1, 'thinking': 1, 'of': 1, 'and': 8, 'told': 1,
'me': 1, 'what': 1, 'to': 3, 'say-yi-yay': 1, 'says': 3, 'know': 11, 'that': 7,
"can't": 3, 'be': 10, 'bad': 3, 'yes': 2, 'should': 7, 'glad': 7, 'said': 1,
'hurt': 2, 'so': 1, 'almost': 1, 'mind': 1, 'now': 1, 'knows': 1, "you're": 1,
'not': 1, 'the': 1, 'hurting': 1, 'kind': 1, 'oo': 2, 'with': 4, 'a': 4, 'like':
4, 'up': 1, 'only': 1, 'fair': 1, 'pride': 1, 'can': 1, 'too': 1, 'pologize': 1,
'Because': 1, 'Yes': 1}
```

[15]:
```
# How to define the most common words?

def most_common_words(freqs):
    # First I find the number of repetition
    values = freqs.values()
    best = max(freqs.values())
    words = []
    # Now I find which words reached that number
    for k in freqs:
        if freqs[k] == best:
            words.append(k)
    return (words, best)

(w,b) = most_common_words(beatles)
```

```
print(w,b)
```

```
['her', 'with', 'a', 'like'] 4
```

[17]:
```
def words_often(freqs, minTimes):
    result = []
    done = False
    while not done:
        temp = most_common_words(freqs)
        if temp[1] >= minTimes:
            result.append(temp)
            for w in temp[0]:
                del(freqs[w])  #remove word from dictionary
        else:
            done = True
    return result

print(words_often(beatles, 5))
```

```
[]
```

9 Video 8: Fibonacci and Dictionaries

Going back to standard Fibonacci code to see why it is **inefficient**.

[ ]:
```
def fib(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1) + fib(n-2)
```

It is inefficient because many calculations are repeated over and over unnecesarily. It really starts getting slower.

Instead of recalculating the same values many times, we can keep track of what we've already done.

[18]:
```
def fib_efficient(n, d):
    # Do a lookup first in case I've already calculated the value
    if n in d:
        return d[n]
    else:
        # Modify dictionary as progress through function calls
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
        d[n] = ans
        return ans

# The base cases need to be added to the initial dictionary
d = {1:1, 2:2}
print(fib_efficient(6, d))
```

```
13
```

This is often called **memoization**: using the dictionary to hold on to values that I've already computed and I don't need to compute again.

10 Video 9: Global Variables

Can be dangerous to use:

- Breaks the scoping of variables by function call.
- Allows for side effects of changing variables values in ways that affect other computation.

But can be convenient when want to keep track of information inside a function. Let's use the Fibonacci efficient and standard form of the function to test this concept.

[27]:
```
# Standard way
def fib(n):
    global numFibCalls
    numFibCalls += 1
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1)+fib(n-2)

# Efficient way
def fibef(n, d):
    global numFibCalls
    numFibCalls += 1
    if n in d:
        return d[n]
    else:
        ans = fibef(n-1,d)+fibef(n-2,d)
        d[n] = ans
        return ans

# Testing the idea
numFibCalls = 0
fibArg = 30

print(fib(fibArg))
print('function calls', numFibCalls)

numFibCalls = 0

d = {1:1, 2:2}
print(fibef(fibArg, d))
print('function calls', numFibCalls)
```

```
1346269
function calls 1664079
1346269
function calls 57
```

6.00.1.x - Week 4 - Good Programming Practices

October 11, 2020

1 Video 1: Programming Challenges

Three strategies to keep your code free of bugs:

1. Testing.
    - Compare input/output pairs to specification.
    - How can I break my program?
2. Defensive programming.
    - Write specifications for functions.
    - Modularize programs.
    - Check conditions on inputs/outputs (assertions).
3. Debugging.
    - Study events leading up to an error.
    - "Why is it not working?"
    - "How can I fix my program?"

2 Video 2: Classes of Tests

Setting yourself up for easy testing and debugging:

- From the **start**, design ode to ease this part.
- Break program into **modules** that can be tested and debugged individually.
- Docuemnt constraints on modules.
- Document assumptions behind code design.

When are you ready to test?

- Ensure **code runs**.
- Have a **set of expected results**.

Classes of tests

- Unit testing.
  - Validate each piece of program.
  - Testing each function separately.
- Regression testing.
  - Add test for bugs as you find them in a function.
  - Catch reintroduced errors that were previously fixed.
- Integration testing.
  - Does overall program work?
  - DON'T rush to do this, but do it eventually.

Testing approaches

- **Intuition** about natural boundaries to the problem.
  - Can you come up with some natural partitions?
- If no natural partitions, might do **random testing**.
- **Black box testing**.
  - Designed **without looking** at the code.
  - Can be done by someone other than the implementer to avoid some implementer **biases**.
  - Testing can be **reused** if implementation changes.
  - **Paths** through especification.
- **Glass box testing**.

- **Use code** directly to guide design of test cases.
- Called **path complete** if every potential through code is tested at least once.
- Drawbacks: it can go through loops arbitrarily many times; you may also miss paths.
- A path complete test suite could **miss a bug**.

In glass box testing, we try to sample as many paths through the code as we can. In the case loops, we want to sample three cases:

1. Not executing the loop at all.
2. Executing the loop exactly once.
3. Executing the loop multiple times.

3 Video 3: Bugs

Once you know there are bugs, you will want to:

- Isolate the bugs.
- Eradicate the bugs.
- Retest until code runs correctly.

Overt vs. covert

- Overt has an obvious manifestation - code crashes or runs forever.
- Covert has no obvious manifestation - code returns a value, which may be incorrect but hard to determine.

Persistent vs. intermittent

- Persistent occurs every time code is run.
- Intermittent only occurs some times, even if run on same input.

4 Video 4: Debugging

Some tools for debugging:

- Built in to IDLE and Anaconda.
- Python Tutor.
- print statement.
- use your brain, be systematic in your hunt.

Print statements

- When to print:
  - enter function.
  - parameters.
  - function results.
- Use bisection method:
  - put print halfway in code.
  - decide where bug may be depending on values.

Types of errors:

- IndexError
- TypeError
- NameError
- SyntaxError

Logic errors are hard:

- Think before writing new code.
- Draw pictures, take a break.
- Explain the code to: someone else, a rubber ducky.

Debugging steps

- Study program code.
- Use the **scientific method** in order to debug.

5 Video 5: Debugging Example

Treat as a search problem: looking for explanation for incorrect behavior.

- Study available data.
- Form an hypothesis consistent with the data.
- Design and run a repeatable experiment with potential to refute the hypothesis.
- Keep record of experiments performed: use narrow range of hypotheses.

6 Video 6: Exceptions

Exceptions

What happens when procedure execution hits an **unexpected condition**?

Get an exception to what was expected:

- Trying to access beyond list limits.
- Trying to convert an inappropriate type.
- Referencing a non-existing variable.
- Mixing data types without coercion.

Other types of exceptions:

- SyntaxError.
- NameError.
- AttributeError.
- TypeError.
- ValueError.
- IOError.

What to do with exceptions?

- Fail silently: bad idea. User gets no warning.
- Return an **error** value: complicates code having to check for a special value.
- Stop execution, **signal error** condition. In Python: **raise an exception**.

[2]:
```
raise Exception("descriptive String")
```

```
␣
↪---------------------------------------------------------------------------

Exception                                 Traceback (most recent call␣
↪last)

<ipython-input-2-6295980b03e8> in <module>
----> 1 raise Exception("descriptive String")

Exception: descriptive String
```

[3]:
```
try:
    a = int(input('Tell me one number:'))
    b = int(input('Tell me another number:'))
    print(a/b)
    print ('Okay')
except:
    print('Bug in user input.')
print('Outside')
```

```
Tell me one number:0
Tell me another number:1
0.0
Okay
Outside
```

Exceptions **raised** by any statement in body of **try** are **handled** by the **except** statement and execution continues after the body of the except statement. Very useful!

This can also be done saying the type of error after the except, to distinguish between the different exceptions that might come up.

Other options are:

- else: body of this is executed when executions of associated **try** body **completes with no exceptions**.
- finally: body of this is **always executed** after try, else and except clauses, even if they raised another rror or executed a break, continue or return. This is useful for clean-up code that should be run no matter what else happened (e.g. close a file).

7 Video 7: Exceptions Examples

First example of how to use the structure mentioned in the video 6.

[4]:
```
while True:
    try:
        n = input('Please enter an integer: ')
        n = int(n)
        break
    except ValueError:
        print('Input not an integer; try again')
print('Correct input of an integer!')
```

```
Please enter an integer: 2
Correct input of an integer!
```

This only handles ValueErrors.

A really nice example of how to use this to actually implement code is shown in the following case.

[5]:
```
data = []
file_name = input('Provide a name of a file of data ')

try:
    fh = open(file_name, 'r')
except IOError:
    print('cannot open', file_name)
else:
    for new in fh:
        if new != '\n':
            addIt = new[:-1].split(',') #remove trailing \n
            data.append(addIt)
finally:
    fh.close() # close file even if fail

gradesData = []
if data:
    for student in data:
        try:
            name = student[0:-1]
            grades = int(student[-1])
            gradesData.append([name, [grades]])
        except ValueError:
            gradesData.append([student[:], []])
```

```
Provide a name of a file of data a
cannot open a
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-5-ac0d75feac4c> in <module>
        12             data.append(addIt)
        13 finally:
   ---> 14     fh.close() # close file even if fail
        15
        16 gradesData = []


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

8 Video 8: Exceptions as Control Flow

A new keyword: raise.

- Don't return special values when an error occurred and then check whether 'error value' was returned.
- Instead, **raise an exception** when unable to produce a result consistent with function's specification.

Example:

[6]:
```
def get_ratios(L1, L2):
    ratios = []
    for index in range(len(L1)):
        try:
            ratios.append(L1[index]/float(L2[index]))
        except ZeroDivisionError:
            ratios.append(float('NaN')) #NaN = Not a Number
        except:
            raise ValueError('get_ratios called with bad arg')
    return ratios

L1 = [1,2,3,4]
L2 = [5,6,7,8]
get_ratios(L1,L2)
```

[6]:
```
[0.2, 0.3333333333333333, 0.42857142857142855, 0.5]
```

[7]:
```
L3 = [5,6,7]
get_ratios(L1,L3)
```

```
␣
↪---------------------------------------------------------------------------

IndexError                                Traceback (most recent call␣
↪last)

<ipython-input-6-049f68bfb879> in get_ratios(L1, L2)
      4         try:
----> 5             ratios.append(L1[index]/float(L2[index]))
      6         except ZeroDivisionError:


IndexError: list index out of range


During handling of the above exception, another exception occurred:
```

```
       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-7-f74906737b77> in <module>
         1 L3 = [5,6,7]
   ----> 2 get_ratios(L1,L3)


       <ipython-input-6-049f68bfb879> in get_ratios(L1, L2)
         7             ratios.append(float('NaN')) #NaN = Not a Number
         8         except:
   ----> 9             raise ValueError('get_ratios called with bad arg')
        10     return ratios
        11


       ValueError: get_ratios called with bad arg
```

[8]:
```
L4 = [5,6,7,0]
get_ratios(L1,L4)
```

[8]:
```
[0.2, 0.3333333333333333, 0.42857142857142855, nan]
```

9 Video 9: Assertions

- Want to be sure that **assumptions** on state of computation are as expected.
- Use an **assert statement** to raise an AssertionError exception if assumptions not met.
- An example of good **defensive programming**.

Really interesting idea! Let's see an example.

[10]:
```
def avg(grades):
    assert not len(grades) == 0, 'no grades data'
    return sum(grades)/len(grades)
```

[11]:
```
avg([1,2,3])
```

[11]:
```
2.0
```

[12]:
```
avg([])
```

```
␣
↪---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call␣
↪last)

<ipython-input-12-938d6a1744f6> in <module>
----> 1 avg([])


<ipython-input-10-2221a6b79207> in avg(grades)
  1 def avg(grades):
----> 2     assert not len(grades) == 0, 'no grades data'
  3     return sum(grades)/len(grades)


AssertionError: no grades data
```

- Raises an AssertionError if it is given an empty list for grades, otherwise runs ok.

Properties of assertions

- Assertions don't allow a programmer to control response to unexpected conditions.
- Ensure that **execution halts** whenever an expected condition is not met.
- Typically used to check inputs to fuctions procedures, but can be used anywhere.
- Can be used to check outputs of a function to avoid propagating bad values.

- Can make it easier to locate a source of a bug.

Where to use assertions?

- Check types of arguments or values (e.g. no string stresses).
- Check that invariants on data structures are met.
- Check constraints on return values (e.g. no negative distances).
- Check for violations of constraints on procedure (e.g. no duplicates in a list).

10 Notes during Problem Set #4

A more organized way of checking whether a key is or not in a dictionary is shown.

[ ]:
```
hand = {'a':1, 'q':1, 'l':2, 'm':1, 'u':1, 'i':1}
hand['e']
```

[ ]:
```
hand.get('e', 0)
```

6.00.1.x - Week 5 - Object Oriented Programming

October 24, 2020

1 Video 1: Object Oriented Programming

- Every **object** has:
  - A type.
  - An internal data representation (primitive or composite).
  - A set of procedures for interaction with the object.
- Each instance is a particular type of object.
  - 1234 is instance of an int.
  - a = 'hello= is an instance of a string.

- Everything in python is an **object** and has a type.
- Objects are a **data abstraction** that caputre:
  - Internal representation through data attributes.
  - Interface for interacting with object through methods (procedures), defines behaviors but hides implementation.
- Can **create new instances of objects**.
- Can also **destroy objects**.

Built in data objects:

- Lists.
- Tuples.
- Strings.

We want to explore ability to create our own data object types.

How are lists represented internally? They are a linked list of cells. But it doesn't matters. What matters is how to **manipulate** them. How do you do that?

- L[i], L[i:j], L[i,j,k]

- len(), min(), max(), etc.

Internal representation should be **private**. Correct behavior may be compromised if you manipulate it directly - instead you use **defined interfaces** to do so.

How to create your own objects with classes?

- Distinction: creating a class and using an instance of said class.
  - A list is a class. L1 = [1,2,3] is an instance of said class.
- Creating the class involves:
  - Defining the class name.
  - Defining class attributes.
- Using the class involves:
  - Creating new instances of objects.
  - Doing operations on the instances.

Advantages of OOP

- Bundle data into packages together with rpocedures that work on them through well-defined interfaces.
- Divide-and-conquer development.
  - Implement and test behavior of each class separately.
  - Increased modularity reduces complexity.
- Classes make it easy to reuse code.
  - Many Python modules define new classes.
  - Each class has a separate environment (no collision on function names).
  - Inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior.

2 Video 2: Class Instances

Use the class keyword to define a new type:

class Coordinate(object): (define attributes here)

- Similar to def, indent code to indicate which statements are part of the **class definition**
- 'class' shows class definition.
- 'Coordinate' is the class name.
- The word 'object' means that Coordinate is a Python object and inherits all its attributes.
  - Coordinate is a subclass of object.
  - Object is a superclass of Coordinate.

What are attributes?

- Data and procedures that 'belong' to the class.
  - Data attributes.
  - Procedural attributes (methods). Think of methods as functions that only work with this class.

How to create an instance of an object?

- Use a special method called **init** to initialize **some** data attributes:

[65]:
```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

- 'self' is a parameter to refer to an instance of the class.
- x,y is the data that initializes a 'Coordinate' object.

Example of actually creating an instance of a class:

[66]:
```
# Creating a new object of type Coordinate and passing in 3 and 4 to the␣
 ↪__init__ method.
c = Coordinate(3,4)
# Note that argument for 'self' is automatically supplied by Python.
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

```
3
0
```

Think of c as pointing to a frame:

- Within the scope of that frame we bound values to data attribute variables.
- c.x is interpreted as getting the value of c (a frame) and then looking up the value associated with x within that frame (thus the specific value for this instance).

3 Video 3: Methods

What is a method?

- Procedural attribute, like a function that works only with this class.
- Python always passes the actual object as the first argument, convention is to use self as the name of the first argument of all methods.
- The "." operator is used to access any attribute:
  - A data attribute of an object.
  - A method of an object.

Example:

[67]:
```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5

c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.distance(origin))
```

```
5.0
```

How to use a method from the Coordinate class?

- c: object on which to call method.
- distance: name of method.
- origin: parameters not including self.

In this case, c.distance inherits the *distance* from the class definition, and automatically uses c as the first argument.

Another way:

Coordinate.distance gets the value of Coordinate, then looks up the value associated with distance (a procedure), then invokes it (which requires two arguments).

[68]:
```
# Name of class - Name of method - Both parameters, c and origin
print(Coordinate.distance(c,origin))
```

```
5.0
```

How to print a representation of an object?

- Uninformative print representation by default.
- Define a **str** method for a class.

[69]:
```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<" + str(self.x) + "," + str(self.y) + ">"

c = Coordinate(3,4)
# This prints it in a new way
print(c)
```

```
<3,4>
```

How to check if something is an instance of a class, in this case Coordinate?

[70]:
```
print(isinstance(c,Coordinate))
```

```
True
```

Special operators

Full list: https://docs.python.org/3/reference/datamodel.html#basic-customization

- Like print, can override these to work with your class.
- Define them with double underscores before/after:
  - **add**
  - **sub**
  - **eq**
  - **lt**
  - **str**
  - and many others.

4 Video 4: Classes Examples

Example #1: Fractions

- Internal representation is two ints: numerator, denominator.
- Interface a.k.a. methods a.k.a. how to interact with *Fractions* objects.
  - Print representation.
  - Add, subtract.
  - Convert to a float.

[71]:
```
# Create the class and create the print representation.
class fraction(object):
    def __init__(self,numer,denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return str(self.numer) + ' / ' + str(self.denom)

oneHalf = fraction(1,2)
twoThirds = fraction(2,3)
print(oneHalf)
print(twoThirds)
```

```
1 / 2
2 / 3
```

One additional thing that is very important is being able to access data attributes. It is important to use a method that gives me an attribute of the object, and not call the attribute of the object directly. Here's how it's done:

[72]:
```
class fraction(object):
    # This was already here
    def __init__(self,numer,denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return str(self.numer) + ' / ' + str(self.denom)
    # This is the new part
    def getNumer(self):
        return self.numer
    def getDenom(self):
        return self.denom

oneHalf = fraction(1,2)
print(oneHalf.getNumer())
# Can also invoke it in this way
print('another way')
```

```
fraction.getDenom(oneHalf)
```

```
1
another way
```

[72]:
```
2
```

[73]:
```
# Adding the sum and subtraction

class fraction(object):
    # This was already here
    def __init__(self, numer, denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return str(self.numer) + ' / ' + str(self.denom)
    def getNumer(self):
        return self.numer
    def getDenom(self):
        return self.denom
    # This is new
    def __add__(self, other):
        numerNew = other.getDenom() * self.getNumer() \
                   + other.getNumer() * self.getDenom()
        denomNew = other.getDenom() * self.getDenom()
        return fraction(numerNew, denomNew)
    def __sub__(self, other):
        numerNew = other.getDenom() * self.getNumer() \
                   - other.getNumer() * self.getDenom()
        denomNew = other.getDenom() * self.getDenom()
        return fraction(numerNew, denomNew)
    def convert(self):
        return self.getNumer() / self.getDenom()

oneHalf = fraction(1,2)
twoThirds = fraction(2,3)
new = oneHalf + twoThirds
print(new)
```

```
7 / 6
```

[74]:
```
threeQuarters = fraction(3,4)
secondNew = twoThirds - threeQuarters
print(secondNew)
```

```
-1 / 12
```

[75]:
```
# Now I want to convert to a float.

class fraction(object):
    # This was already here
    def __init__(self, numer, denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return str(self.numer) + ' / ' + str(self.denom)
    def getNumer(self):
        return self.numer
    def getDenom(self):
        return self.denom
    def __add__(self, other):
        numerNew = other.getDenom() * self.getNumer() \
                   + other.getNumer() * self.getDenom()
        denomNew = other.getDenom() * self.getDenom()
        return fraction(numerNew, denomNew)
    def __sub__(self, other):
        numerNew = other.getDenom() * self.getNumer() \
                   - other.getNumer() * self.getDenom()
        denomNew = other.getDenom() * self.getDenom()
        return fraction(numerNew, denomNew)
    # This is new
    def convert(self):
        return self.getNumer() / self.getDenom()

oneHalf = fraction(1,2)
twoThirds = fraction(2,3)
oneHalf.convert()
```

[75]:
```
0.5
```

We are going to create yet another example: **A set of integers**

- Initially the set is empty.
- A particular integer appears only once in a set: representational invariant enforced by the code.

For this I will need:

- Internal data representation: use a list to store the elements of a set.
- Interface:
  - Insert - insert integer e into set if not there.
  - Member - return True if integer e is in set, False else.
  - Remove - remove integer e from set, error if not present.

[76]:
```
class intSet(object):
    """An intSet is a set of integers
    The value is represented by a list of ints, self.vals.
    Each int in the set occurs in self.vals exactly once."""

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if not e in self.vals:
            self.vals.append(e)

    def member(self, e):
        """Assumes e is an integer
        Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def __str__(self):
        """Returns a string representation of self"""
        self.vals.sort()
        result = ''
        for e in self.vals:
            result = result + str(e) + ','
        return '{' + result[:-1] + '}'

s = intSet()
print(s)
s.insert(3)
s.insert(4)
s.insert(3)
print(s)
print(s.member(3))
print(s.member(5))
s.insert(6)
print(s)
s.remove(3)
print(s)
```

```
s.remove(3)
```

```
{}
{3,4}
True
False
{3,4,6}
{4,6}
```

```
      ␣
↪---------------------------------------------------------------------------
```

```
       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-76-c470139e6128> in remove(self, e)
        23         try:
   ---> 24             self.vals.remove(e)
        25         except:

       ValueError: list.remove(x): x not in list


   During handling of the above exception, another exception occurred:

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-76-c470139e6128> in <module>
        46 s.remove(3)
        47 print(s)
   ---> 48 s.remove(3)

       <ipython-input-76-c470139e6128> in remove(self, e)
        24             self.vals.remove(e)
        25         except:
   ---> 26             raise ValueError(str(e) + ' not found')
        27
        28     def __str__(self):

       ValueError: 3 not found
```

5 Video 5: Why OOP

- Bundle together objects that share common attributes and procedures that operature on those attributes.
- Use **abstraction** to make a distinction between how to implement an object vs how to use the object.
- Build **layers** of object abstractions that inherit behaviors from other classes of objects.
- Create our **own classes of objects** on top of Python's basic classes.

Two kinds of attributes a group of objects have:

Data attributes

- How can you represent your object with data?
- What it is
- For a coordinate: x and y values.
- For an animal: age and name.

Procedural attributes

- What kinds of things can you do with the object?
- What it does?
- For a coordiante: find distance between two.
- For an animal: make a sound.

Let's now build an animal.

[77]:
```
class Animal(object):
    def __init__(self, age):
        # I can define other data attributes even if I don't pass them as␣
↪parameters when creating an instance of this class.
        self.age = age
        self.name = None
    # Getter methods
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    # Setter methods
    def set_age(self, newage):
        self.age = newage
    # Very interesting! Default value for one of the parameters.
    def set_name(self, newname=""):
```

```
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)

myAnimal = Animal(3)
print(myAnimal)
```

```
animal:None:3
```

[78]:
```
myAnimal.set_name('foobar')
print(myAnimal)
```

```
animal:foobar:3
```

[79]:
```
# This is how we should access the internal representation.
print('Correct method')
print(myAnimal.get_age())
# This is another way - which we should NOT use.
print('Incorrect method')
print(myAnimal.age)
```

```
Correct method
3
Incorrect method
3
```

One should **always use the getters** instead of accesing the data attributes directly:

- Good style.
- Easy to maintain code.
- Prevents bugs.

Python is not great at information hiding

- Allows you to access data from outside class definition.
- Allows you to write to data from outside class definition.
- Allows you to create data attributes for an instance from outside class definition.

It's **not good style** to do any of these.

6 Video 6: Hierarchies

- Parent class (superclass) - Animal.
- Child class (subclass) - Person/Cat/Rabbit.
  - Inherits all data and behaviors of parent class.
  - Add more info.
  - Add more behavior.
  - Override behavior.

— Example

[80]:
```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)

# Inherits all attributes of Animal
# Add new functionality through new methods
# Override the __str__ method to identify an instance of this class as a Cat␣
 ↪and not just any Animal.
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)

class Rabbit(Animal):
    def speak(self):
        print("meep")
    def __str__(self):
        return "rabbit:"+str(self.name)+":"+str(self.age)

class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        Animal.set_name(self, name)
```

```
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        # alternate way: diff = self.age - other.age
        diff = self.get_age() - other.get_age()
        if self.age > other.age:
            print(self.name, "is", diff, "years older than", other.name)
        else:
            print(self.name, "is", -diff, "years younger than", other.name)
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

[81]:
```
# Examples of using this new information
jelly = Cat(1)
jelly.get_name()
```

[82]:
```
jelly.set_name('JellyBelly')
jelly.get_name()
```

[82]:
```
'JellyBelly'
```

[83]:
```
print(jelly)
```

```
cat:JellyBelly:1
```

[84]:
```
print(Animal.__str__(jelly))
```

```
animal:JellyBelly:1
```

[85]:
```
# Python looks for the definition of speak inside Cat, because jelly is an␣
↪instance of Cat.
jelly.speak()
```

```
meow
```

While a subclass inherits all methods from upper parts of the hierarchy, the superclass can't access the methods from the lower parts of the hierarchy.

Which method to use?

- Subclass can have methods with same name as superclass.

- Subclass can have methods with same name as other subclasses.
- For an instance of a class, look for a method name in current class definition.
- If not found, look for a method name up the hierarchy.
- Use first method up the hierarchy that you found with that method name.

7 Video 7: Class Variables

- Different from an instance variable (which is what we normally use, or have used so far).

Let's see them with an example

[86]:
```
class Animal(object):
    # These are instance variables
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
# New code
class Rabbit(Animal):
    # Class variable - OUTSIDE of the init definition.
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        # This is the part that changes the class variable over and over.
        Rabbit.tag += 1
    def get_rid(self):
        # Technique to make sure all numbers are the same size.
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
    def __add__(self, other):
        # returning object of same type as this class
        # This is using the __init__ definition of the Rabbit class. So age,␣
 ↪parent1, parent2.
        return Rabbit(0, self, other)
    def __eq__(self, other):
        parents_same = self.parent1.rid == other.parent1.rid \
                       and self.parent2.rid == other.parent2.rid
        parents_opposite = self.parent2.rid == other.parent1.rid \
```

```
                    and self.parent1.rid == other.parent2.rid
        return parents_same or parents_opposite
```

The class is keeping track of the tag, and every time I create a new instance of a Rabbit object, it'll receive a unique ID, the latest value.

- rid is an instance variable.
- tag is a class variable.

Addition procedure

- Allows for r4 = r1 + r2, where r1 and r2 are Rabbit instances.
- f4 is a new Rabbit instance with age 0.
- r4 has self as one parent, and other as the other parent.
- In **init**, should change to check that parent1 and parent2 are of type Rabbit.

8 Video 8: Building a Class

- Explore in some detail an example of building an application that organizes info about people.

[87]:
```
# Use it to calculate age
import datetime

class Person(object):
    def __init__(self, name):
        """create a person called name"""
        self.name = name
        self.birthday = None
        # Name is a string, so split into a list of strings based on spaces,␣
 ↪then extract last element.
        self.lastName = name.split(' ')[-1]

    def setBirthday(self,month,day,year):
        """sets self's birthday to birthDate"""
        self.birthday = datetime.date(year,month,day)

    def getAge(self):
        """returns self's current age in days"""
        if self.birthday == None:
            raise ValueError
        return (datetime.date.today() - self.birthday).days

    def getLastName(self):
        """return self's last name"""
        return self.lastName

    def __str__(self):
        """return self's name"""
        return self.name

    def __lt__(self, other):
        """return True if self's ame is lexicographically
       less than other's name, and False otherwise"""
        if self.lastName == other.lastName:
            return self.name < other.name
        return self.lastName < other.lastName
```

[88]:
```
# example usage

p1 = Person('Mark Zuckerberg')
p1.setBirthday(5,14,84)
p2 = Person('Drew Houston')
p2.setBirthday(3,4,83)
p3 = Person('Bill Gates')
```

```
p3.setBirthday(10,28,55)
p4 = Person('Andrew Gates')
p5 = Person('Steve Wozniak')

personList = [p1, p2, p3, p4, p5]

for e in personList:
    print(e)
```

```
Mark Zuckerberg
Drew Houston
Bill Gates
Andrew Gates
Steve Wozniak
```

[89]:
```
personList.sort()
for e in personList:
    print(e)
```

```
Andrew Gates
Bill Gates
Drew Houston
Steve Wozniak
Mark Zuckerberg
```

9 Video 9: Visualizing the Hierarchy

Let's define a subclass of the Person class - an MIT person.

[90]:
```
class MITPerson(Person):
    # Class attribute
    nextIdNum = 0 # next ID number to assign

    def __init__(self, name):
        # Using the person class initialization - no need to reinvent the wheel!
        Person.__init__(self, name) # initialize Person attributes
        # new MITPerson attribute: a unique ID number
        self.idNum = MITPerson.nextIdNum
        MITPerson.nextIdNum += 1

    def getIdNum(self):
        return self.idNum

    # sorting MIT people uses their ID number, not name!
    def __lt__(self, other):
        return self.idNum < other.idNum

    def speak(self, utterance):
        return (self.getLastName() + " says: " + utterance)

m3 = MITPerson('Mark Zuckerberg')
Person.setBirthday(m3,5,14,84)
m2 = MITPerson('Drew Houston')
Person.setBirthday(m2,3,4,83)
m1 = MITPerson('Bill Gates')
Person.setBirthday(m1,10,28,55)

MITPersonList = [m1, m2, m3]
print(m3)
print(m1.speak('hi there'))
```

```
Mark Zuckerberg
Gates says: hi there
```

[91]:
```
for e in MITPersonList:
    print(e)
```

```
Bill Gates
Drew Houston
Mark Zuckerberg
```

[92]:
```
# Sorting by ID
personList.sort()
```

```
for e in MITPersonList:
    print(e)
```

```
Bill Gates
Drew Houston
Mark Zuckerberg
```

[93]:
```
# MITPerson have IDs, no problem comparing them.
p1 =  MITPerson('Eric')
p2 = MITPerson('John')
p3 = MITPerson('John')
# Not an MITPerson, does not have an ID number!
p4 = Person('John')

p1<p2
```

[93]:
```
True
```

[94]:
```
p1<p4
```

```
␣
↪---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-94-e4b6153c0891> in <module>
----> 1 p1<p4


<ipython-input-90-d8c6494e3657> in __lt__(self, other)
     15     # sorting MIT people uses their ID number, not name!
     16     def __lt__(self, other):
---> 17         return self.idNum < other.idNum
     18
     19     def speak(self, utterance):


AttributeError: 'Person' object has no attribute 'idNum'
```

[ ]:
```
p1>p4
```

Why does p4 < p1 work but p4 > p1 doesn't?

- p4 < p1 is equivalent to p4.___lt___(p1) which means use the **lt** method associated with the

type of p4, namely a Person.

- p1 < p4 is equivalent to p1.__lt__(p4) which means we use the **lt** method associated with the type of p1, namely an MITPerson and since p4 is a Person, it does not have an IDNum.

10 Video 10: Adding another class

We keep creating classes

[95]:
```
class UG(MITPerson):
    def __init__(self, name, classYear):
        # Use the inherited method from the MITPerson class.
        MITPerson.__init__(self, name)
        self.year = classYear

    def getClass(self):
        return self.year

    def speak(self, utterance):
        # Use the same from MITPerson, but adding something!
        return MITPerson.speak(self, " Dude, " + utterance)

class Grad(MITPerson):
    pass

def isStudent(obj):
    return isinstance(obj,UG) or isinstance(obj,Grad)

s1 = UG('Matt Damon', 2017)
s2 = UG('Ben Affleck', 2017)
s3 = UG('Lin Manuel Miranda', 2018)
s4 = Grad('Leonardo di Caprio')

studentList = [s1, s2, s3, s4]

print(s1)
print(s1.getClass())
print(s1.speak('where is the quiz?'))
print(s2.speak('I have no clue!'))
```

```
Matt Damon
2017
Damon says:  Dude, where is the quiz?
Affleck says:  Dude, I have no clue!
```

Suppose we now want to create another class of students, a transfer student. But in order to modify the isStudent method, I'd need to change some things. So let's look at this possibility.

[96]:
```
# Create a superclass that covers all students
class Student(MITPerson):
    # Pass is a special keyword
    pass
```

```
class UG(Student):
    def __init__(self, name, classYear):
        MITPerson.__init__(self, name)
        self.year = classYear

    def getClass(self):
        return self.year

    def speak(self, utterance):
        return MITPerson.speak(self, " Dude, " + utterance)

class Grad(Student):
    pass

class TransferStudent(Student):
    pass

def isStudent(obj):
    return isinstance(obj,Student)

s1 = UG('Matt Damon', 2017)
s2 = UG('Ben Affleck', 2017)
s3 = UG('Lin Manuel Miranda', 2018)
s4 = Grad('Leonardo di Caprio')
s5 = TransferStudent('Robert deNiro')

studentList = [s1, s2, s3, s4, s5]

print(s1)
print(s1.getClass())
print(s1.speak('where is the quiz?'))
print(s2.speak('I have no clue!'))
```

```
Matt Damon
2017
Damon says:  Dude, where is the quiz?
Affleck says:  Dude, I have no clue!
```

Substitution principle: Important behaviors of superclass should be supported by all subclasses.

In this case, Student was created in order to make all the subclasses (UG, grad and transfer) support the behaviors from the main Student class.

11 Video 11: Using Inherited Methods

This time we create yet another class.

[97]:
```
class Professor(MITPerson):
    def __init__(self, name, department):
        MITPerson.__init__(self, name)
        self.department = department

    def speak(self, utterance):
        # This will shadow MITPerson speak method
        newUtterance = 'In course ' + self.department + ' we say '
        return MITPerson.speak(self, newUtterance + utterance)

    def lecture(self, topic):
        # Uses own speak method, not MITPerson's
        return self.speak('it is obvious that ' + topic)

faculty = Professor('Doctor Arrogant', 'six')
```

[98]:
```
print(s1.speak('I have no idea'))
```

```
Damon says:  Dude, I have no idea
```

[99]:
```
print(m1.speak('Hey there'))
```

```
Gates says: Hey there
```

[100]:
```
# Uses Professor speak method, which uses MITPerson method
print(faculty.speak('hi there'))
```

```
Arrogant says: In course six we say hi there
```

[101]:
```
# Uses Professor speak method
print(faculty.lecture('hi there'))
```

```
Arrogant says: In course six we say it is obvious that hi there
```

Let's now change the original MITPerson class.

[102]:
```
class MITPerson(Person):
    nextIdNum = 0 # next ID number to assign

    def __init__(self, name):
        Person.__init__(self, name) # initialize Person attributes
        # new MITPerson attribute: a unique ID number
        self.idNum = MITPerson.nextIdNum
        MITPerson.nextIdNum += 1
```

```
    def getIdNum(self):
        return self.idNum

    # sorting MIT people uses their ID number, not name!
    def __lt__(self, other):
        return self.idNum < other.idNum

    def speak(self, utterance):
        return (self.name + " says: " + utterance)

class Student(MITPerson):
    pass

class UG(Student):
    def __init__(self, name, classYear):
        MITPerson.__init__(self, name)
        self.year = classYear

    def getClass(self):
        return self.year

    def speak(self, utterance):
        return MITPerson.speak(self, " Dude, " + utterance)

class Grad(Student):
    pass

class TransferStudent(Student):
    pass

def isStudent(obj):
    return isinstance(obj,Student)

# Testing changes
s1 = UG('Matt Damon', 2017)
s2 = UG('Ben Affleck', 2017)
s3 = UG('Lin Manuel Miranda', 2018)
s4 = Grad('Leonardo di Caprio')
s5 = TransferStudent('Robert deNiro')


studentList = [s1, s2, s3, s4, s5]
print(s1)
print(s1.getClass())
print(s1.speak('where is the quiz?'))
print(s2.speak('I have no clue!'))
print(faculty.lecture('hi there'))
```

Matt Damon
2017
Matt Damon says: Dude, where is the quiz?
Ben Affleck says: Dude, I have no clue!
Doctor Arrogant says: In course six we say it is obvious that hi there

12 Video 12: Gradebook Example

Idea: create a class that includes instances of other classes within it.

[106]:
```
class Grades(object):
    """A mapping from students to a list of grades"""
    def __init__(self):
        """Create empty grade book"""
        self.students = []  # list of Student objects
        self.grades = {}    # maps idNum -> list of grades
        self.isSorted = True # true if self.students is sorted

    def addStudent(self, student):
        """Assumes: student is of type Student
           Add student to the grade book"""
        if student in self.students:
            raise ValueError('Duplicate student')
        self.students.append(student)
        self.grades[student.getIdNum()] = []
        self.isSorted = False

    def addGrade(self, student, grade):
        """Assumes: grade is a float
           Add grade to the list of grades for student"""
        try:
            # Grades is a dictionary. Keys = IDs, Values = list of grades.
            self.grades[student.getIdNum()].append(grade)
        except KeyError:
            raise ValueError('Student not in grade book')

    def getGrades(self, student):
        """Return a list of grades for student"""
        try:    # return copy of student's grades
            return self.grades[student.getIdNum()][:]
        except KeyError:
            raise ValueError('Student not in grade book')

    def allStudents(self):
        """Return a list of the students in the grade book"""
        if not self.isSorted:
            self.students.sort()
            self.isSorted = True
        return self.students[:]
        #return copy of list of students

    def gradeReport(course):
        """Assumes: course if of type grades"""
        report = []
```

```
        for s in course.allStudents():
            tot = 0.0
            numGrades = 0
            for g in course.getGrades(s):
                tot += g
                numGrades += 1
            try:
                average = tot/numGrades
                report.append(str(s) + '\'s mean grade is '
                              + str(average))
            except ZeroDivisionError:
                report.append(str(s) + ' has no grades')
        return '\n'.join(report)

# Loading the data
ug1 = UG('Matt Damon', 2018)
ug2 = UG('Ben Affleck', 2019)
ug3 = UG('Drew Houston', 2017)
ug4 = UG('Mark Zuckerberg', 2017)
g1 = Grad('Bill Gates')
g2 = Grad('Steve Wozniak')

six00 = Grades()
six00.addStudent(g1)
six00.addStudent(ug2)
six00.addStudent(ug1)
six00.addStudent(g2)
six00.addStudent(ug4)
six00.addStudent(ug3)

six00.addGrade(g1, 100)
six00.addGrade(g2, 25)
six00.addGrade(ug1, 95)
six00.addGrade(ug2, 85)
six00.addGrade(ug3, 75)
```

13 Video 13: Generators

An issue is raised: What if I don't (or don't want to) generate a copy of a huge list every single time I want to operate with it?

The answer is **generators**.

Any procedure or method with a 'yield' statement inside it is called a generator.

6.00.1.x - Week 6 - Algorithmic Complexity

October 22, 2020

1 Video 1: Program Efficiency

- Normally there is a trade off between **time and space** efficiency of a program.
- For the most part, we will focus on time efficiency.

- While there are many ways to implement a program, there are only a handful of algorithms that can solve a given problem.

How can I evaluate efficiency?

- Measure with a timer.
- Count the operations.
- Abstract notion of **order of growth**. This is the most appropriate way of assessing the impact of choices of algorithm in solving a problem.

Timing a program

[8]:
```
import time
def c_to_f(c):
    return c*9/5 + 32
t0 = time.clock()
c_to_f(100000)
t1 = time.clock() - t0
print('t =', t0, ':', t1, 's,')
```

```
t = 1512.6299181 : 0.00010560000009718351 s,
```

```
C:\Users\pablo\anaconda3\lib\site-packages\ipykernel_launcher.py:4:
DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be
removed from Python 3.8: use time.perf_counter or time.process_time instead
  after removing the cwd from sys.path.
C:\Users\pablo\anaconda3\lib\site-packages\ipykernel_launcher.py:6:
DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be
removed from Python 3.8: use time.perf_counter or time.process_time instead
```

Problems with this idea

- Time varies between algorithms.
- Time varies between implementations.
- Time varies between computers.
- Time is not predictable based on small inputs.
- Time varies for different inputs but cannot really express a relationship between inputs and time.

Counting operations

- Assume certain steps (comparisons, assignments, accesing objects in memory, etc.) take a constant time.
- Then count the number of operations executed as function of size on input.

Problems with this idea

- Count depends on implementations.
- No real definition of which operations to count.

What would I like to do?

- I want to evaluate algorithm.
- I want to evaluate scalability.
- I want to evaluate in terms of input size.

- When choosing which case we'll study, we'll normally pick the worst case scenario.

Goals

- Evaluate efficiency when input is big.
- Express the growth of program's run time as input size grows.
- I want to put an upper bound on growth.
- I do not need to be precise: an order of growth will do.

Types of orders of growth

- Constant
- Linear
- Quadratic

- Logarithmic
- nlogn
- Exponential

2 Video 2: Big Oh Notation

- Big Oh notation measures an **upper bound on the asymptotic growth**, often called order of growth.
- Big Oh or O() is used to describe worst case:
  - Worst case occurs often and is the bottleneck when a program runs.
  - Express rate of growth of program relative to the input size.
  - Evaluate algorithm, not machine or implementation.

Let's see an example:

[10]:
```
def fact_iter(n):
    """assumes n an int >= 0"""
    # 1 step here
    answer = 1
    # 5 steps inside the loop
    while n > 1:
        answer *= n
        n -= 1
    # 1 final step
    return answer
```

- The number of the steps is 1 + 5n + 1.
- The order of this algorithm is O(n), since the addition and multiplicative factors are not relevant.

Simplification examples

- n^2 + 2n + 2 = O(n^2)
- log(n) + n + 4 = O(n)
- 0.0001 n log(n) * 300n = O(n log n)
- 2 n^30 + 3^n = O(3^n)

We focus on the **dominant terms**.

- O(1) - constant
- O(log n) - logarithmic.
- O(n) - linear
- O(n log n) - loglinear.
- O(n^c) - polynomial.

- O(c^n) - exponential.

C is a constant, n is the size of the input.

- The higher I am in the hierarchy, the more efficient the algorithm is.

Law of Addition for O():

- Used with **sequential** statements.

O(n) + O(n^2) = O(n + n^2) = O(n^2)

Law of Multiplication for O():

- Used with **nested** statements/loops.

O(n) * O(n) = O(n^2) because the outer loop goes n times and the inner loop goes n times for every outer loop tier.

---

3 Video 3: Complexity Classes

Let's see examples of each one of the complexity classes.

O(1) - Constant

- Complexity independent of the input.
- Few interesting algorithms in this class.

O(log n) - Logarithmic

- Bisection search.
- Binary search of a list.

O(n) - Linear

- Searching a list in sequence to see if an element is present.
- Add characters of a string, assumed to be composed of decimal digits.
- factorial_iterative() and factorial_recursive() are both O(n). One does n for loops, the other does n function calls.

Log-Linear

- Very commonly used log-linear algorithm is merge sort.

4 Video 4: Analyzing Complexity

Polynomial

- Most common is quadratic.
- Commonly occurs when we have nested loops or recursive function calls.

5 Video 5: More Analyzing Complexity

Exponential

- Recursive functions where more than one recursive call for each size of problem.
- Many important problems are inherently exponential. So sometimes, we rather approximate solutions quickly than finding the most accurate guess.

6 Video 6: Recursion Complexity

- Interesting comments on methods that appear to be of a given order of growth with respects to the length of its contents, and a different order of growth with respects to its actual length. Re-Check.
- Fibonacci analyzed again.
  - Iterative is O(n).
  - Recursive is O(c^n).
- Interesting analysis regarding the complexity that comes with lists vs the complexity that comes with dictionaries.

7 Video 7: Search Algorithms

Search algorithm: Method for finding an item or group of items with specific properties within a collection of items.

- Collection could be implicit. Example: find square root as a search problem.
- Collection could be explicit. Example: Is a student record in a stored collection of data?

Searching algorithms

- Linear search
  - Brute force search
  - List does not have to be sorted.
  - Complexity is O(n), where n is len(L).
  - Accesing a list item is constant time, regardless the list being homogeneous or heterogeneous. Explanation in video.
- Bisection search.
  - List MUST be sorted to give correct answer.
  - Different implementations of the algorithm are possible.

8 Video 8: Bisection Search

- Linear search on sorted list is still O(n) - still might have to go all the way through the list.

Bisection search

- Complexity is O(log n) - where n is len(L).

Implementation 1

- Recursive way.
- Making copies of the list adds up to a large complexity.
- O(n log n). O(n) for a tighter bound because length of list is halved each recursive call.

Implementation 2

- Define a function and a helper function.
- Instead of copying the list, just remember the indexes that you used in the last step in order to do the recursive call.
- O(log n).

- Using linear search, search for an element is O(n) - regardless of whether the list is sorted or not.
- Using binary search, search for an element is O(log n), assuming the list is sorted.

When does it make sense to sort first then search?

- When sorting is less than O(n). And that is **never** true.

Amortized cost

- Maybe I sort the list once and perform multiple searches.

9 Video 9: Bogo Sort

- Randomly assign the elements in a certain order and check whether the elements are in order or not.
- Complexity of the bogo sort: O(?). It is unbounded.

10 Video 10: Bubble Sort

- Compare consecutive pairs of elements.
- Swap elements in pair such that smaller is first.
- When reach end of list, start over again.
- Stop when no more swaps have been made.
- Complexity = O(n^2) where n is len(L).

11 Video 11: Selection Sort

- First step
  - Extract minimum element.
  - Swap it with element at index 0.
- Subsequent step.
  - Remaining sublist, extract minimum element.
  - Swap it with the element at index 1.
- Keep the left portion of the list sorted
  - At ith step, first i elements in list are sorted.
  - All other elements are bigger than first i elements.

Is there a loop invariant?

- given prefix of list L[0:i] and suffix L[i+1:len(L)], then prefix is sorted and no element in prefix is larger than smallest element in suffix.
- I prove this is true by induction.

1. base case: prefix empty, suffix whole list – invariant true.
2. induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append.
3. when exit, prefix is entire list, suffix empty, so sorted.

Implementation

- I don't want to copy the list! It is very inefficient.
- Complexity is O(n^2) where n is len(L). Timing wise probably more efficient than the bubble sort.

12 Video 12: Merge and Sort

- Use a divide and conquer approach.

1. If list is of length 0 or 1, already sorted.
2. If list has more than one element, split into two lists and sort each.
3. Merge sorted sublists
    1. Look at first element of each, move smaller to end of the result.
    2. When one list empty, just copy rest of other list.

Example of merging (see slide 36/42).

Complexity of the merging: linear in the length of the lists.

Implementation

The merge sort algorithm itself is recursive.

Final complexity analysis

- At first recursion level
  - n/2 elements in each list.
  - O(n) + O(n) = O(n) where n is len(L).
- At second recursion level
  - n/4 elements in each list.
  - two merges: O(n) where n is len(L).
- Each recursion level is O(n).
- Dividing list in half with each recursive call. O(log n) where n is len(L).
- Overall complexity is **O(n log n)) where n is len(L)**.

O(n log n) is the fastest a sort can be.