Python syntax

- Automatically typed (no type declarations before declaring variable), e.g.
  - var1 = True
  - var2 = "Brian" → equivalent to → var2 = 'Brian' → strings can be surrounded by double or single quotes
  - var3 = 3
  - var4 = 3.0 → operations can be freely performed between ints and floats (unlike OCaml)
- 0-indexed (nothing strange here)
- Comments:
  - # single-line comment → single-line comment

    """ Multi-line
    comment
    """ → multi-line comment
- Operations
  - a + b → addition (also string concatenation—nicely intuitive)
  - a - b→ subtraction
  - a * b→ multiplication
  - a / b→ division
  - a // b → floor division (always rounds down to int)
  - a % b → modulo
  - a ** b → a to the power of b
  - can put operations in front of equals sign just like in C, e.g.
    - a += a → a = a + 2
- need to escape apostrophes with "\" e.g.

  'They\'re no CS internships looking for freshman, wahhhhhh'
  'This isn\'t a broken string because I escaped the apostrophe'
- for multiple line statements, end every line but the last with \
  - unlike C or OCaml, whitespace (tabs and new lines) matter in Python, which is good because no semicolons or brackets are necessary, but can get annoying

---

Strings and Console Output

- can grab chars from strings using their index
  - "variable"[1] → grabs "a"—the second char of variable
- methods:
  - len(string_name) → returns length of string (also works with lists)
  - string_name.lower() → returns new string with all lowercase chars of original
  - string_name.upper() → ,, ,, ,, ,, ,, uppercase ,, ,, ,,
  - str(var_name) → turns value or variable into a string
    - dot notation for lower and upper because they are functions specific to strings—len and str accept non-string inputs

- printing:
  - print var_name → prints value of variable to console
  - can print pretty much any type to console
  - can concatenate within print statement, e.g.
    - print “life “ + “of “ “Brian” → prints life of Brian to the console
    - print “I “ + “have “ + str(2) + “coconuts” -> prints I have 2 coconuts to console
  - Similar to C, can use %s to substitute values of variables into a string you want to print, e.g.
    - print “Ah, so your name is %s, your quest is %s, and your favorite color is %s.” % (name, quest, color)
  - if only printing the value of one variable while printing, can use
    - print “String”, var → prints concatenation of String and value of var
- Can grab user input, using raw_input(“Question you would like to ask user”), e.g.
  - name = raw_input(“What is your name?”) → prints What is your name? to the console, allows the user to type a response, and sets the value of name equal to said response
- Can use datetime to grab current date and time
  - From datetime import datetime → imports datetime functionality (modules and functions covered later)
  - datetime.now() → fetches current date and time, can be printed, e.g. print datetime.now()
  - Can extract information from datetime variable using dot notation, e.g.
    - now = datetime.now() → creates new variable now that contains current date and time
    - current_year = now.year → grabs year from now, i.e. 2014
    - current_month = now.month → grabs month from now, i.e. 03
      - can be done with year, month, day, hour, minute, and second

---

Conditionals & Control Flow

- > → greater than
- < → less than
- == -> equals
- != → not equal (this all look familiar?)
- or → the Python equivalent of ||
- and → Python equivalent of &&
- if condition: → if condition (needs colon!!)
- elif condition: → else if condition (needs colon!!)
- else: → else condition (needs colon!!)
  - Note: 4 –space indents on all lines within if/elif/else conditions

---

Functions

- def function_name(parameter): → syntax for declaring a function
- can have multiple parameters, i.e. def function_name(prmtr1, prmtr 2):
- return → returns some value, can return an operation of function, i.e.
    - return parameter ** 2
- can have functions that don't accept any parameters, e.g.

```
def spam:
    print "Eggs!"
spam()
```

→ prints "Eggs!" to the console

- can call functions within other functions

Modules (Libraries)

- Generic import:
    - import math → imports math module, allows functions like sqrt or sin/cos
    - every time you want to use math function, need to add "math." in front of function, e.g.
        - math.sqrt(25) → will return 5
- Function imports:
    - from module import function → imports specific function from stated module, e.g.
        - from math import sqrt → imports sqrt function from math module
    - benefit: no longer need to type "math" in front of all instances of sqrt
- Universal imports:
    - from module import * → imports all functions from module, e.g.
        - from math import * → imports all math functions
    - same benefit as function imports
    - drawback: can get mixed up with functions you declare yourself if you do it with a lot of libraries, e.g. if you defined a function called sqrt, python would get mixed up—best to stick with generic and function imports
- Some built-in functions:
    - max(list__name) → returns maximum of list_name (list of ints/floats)
    - min(list_name) → returns minimum of list_name
    - abs(var_name) → returns absolute value
    - type(var_name) → returns type of variable/value

Lists and Dictionaries

- List operations:
    - list_name[1:] → returns new list that contains every element after and including the second element

  - list_name[:1] → returns new list that contains every element up to but not including the second element
  - list_name[1:4] → returns new list that contains every element after (including) second up to but not including the fifth
    - Same things can be done with strings (consider a string a list of chars, and corresponding indices are particular chars instead of list elements)
  - list_name.append(item) → returns list with item added on to the end of original list
  - list_name.remove(item) → removes item from list
  - list.sort()
- Dictionaries
  - Like lists, but each value matched with a key, e.g.
    - residents = {'Puffin' : 104, 'Sloth' : 105, 'Burmese Python' : 106}
  - access values using dictionary_name[key], e.g.
    - residents['Puffin'] → will return Puffin's room number, 104
  - dict_name[new_key] = new_value → adds new_value with new_key to dict_name
  - del dict_name[key_name] → deletes value/key pair
  - dict_name[key] = new_value → modifies value associated with key to new_value
- for loops
  - iterates through all of the elements in a list from the left-most to the right-most element, e.g.

```
for item in [1, 3, 21]:
    print item
```

→ prints every item in the list

  - can also be used for dictionaries (NOTE: dictionaries are unordered, meaning that any time you loop through a dictionary, you go through every key but are not guaranteed them in any order), e.g.

```
for key in dict
    print dict[key]
```

→ prints every value in dictionary

  - can use conditions in for loops (nothing new here)

Lists and Functions

- options for removing items from a list
  - list_name.pop(index) → removes the item at index from list and returns it to you
  - list_name.remove(item) → searches list for item and removes item if found
  - del(list_name[index]) → removes the item at index, doesn't return anything
- range function: generates lists, three different versions:
  - range(start, stop, step) → generates a list from start up to but not including step, counting up by step between each list item
  - range(start, stop) → step defaults to 1
  - range(stop) → start defaults to 0, step defaults to 1

- two ways of iterating through a list
  - for item in list: → useful to loop through the list, but doesn't modify list
  - for i in range(len(list)): → uses indexes, making it possible to also modify list

- Pretty printing
  - string.join(list_name) → uses string to combine items in list_name
  - print string , → iterates through every character in a string, prints on same line
- Random:
  - randint(low, high) → generates random integer between low and high inclusive

---

Loops

- while loops → pretty much the same as in C
- break → breaks out of the loop
- while/else → similar to if/else, if loop is never entered or if loop exits normally, else will execute, but if loop exits b/c of break, else will not be executed
- enumerate(list_name) → useful function (built-in), works by supplying corresponding index to each element in the list passed to it (list_name)

  syntax:

  ```
  for index, item in enumerate(list_name):
  ```

- zip(list_a, list_b) → create pairs of elements when passed two lists, and will stop at the end of the shorter list (handles three or more lists as well)

Advanced Topics

- iterating over dictionaries:
  - dict_name.items() → prints key/value pairs in no particular order
  - dict_name.keys() → prints keys in no particular order
  - dict_name.values() → prints values in no particular order
  - can use for loop to iterate through and print as well
- list comprehensions:
  - [i for i in range(51) if i % 2 == 0] → prints all evens between 0 and 50
- list slicing:
  - list_name[start:end:stride] → slices list_name at start (inclusive), ends at end (exclusive), incrementing through original list by stride
  - list_name[::stride] → uses default start and end
  - negative stride progresses through list from right to left
- anonymous functions:
  - use lambda instead of fun (OCaml), rest is pretty similar, e.g.

    ```
    squares = [x ** 2 for x in range(1,11)
    print filter(lambda x: x >= 30 and x <= 70, squares)
    ```

- → prints squares of numbers between 1 and 10 that are between 30 and 70, filter takes two arguments, a filtering function and a list

---

Bitwise Operators

- 0b100 →> 4, "0b" being the prefix for all binary numbers
- int(num, base) → will convert string of number from base that it is in (base) to base ten
- bin_num << num → shifts the binary representation of bin_num left by num places
- bin_num >> num → shifts the binary representation of bin_num right by num places
- bin_num1 & bin_num2 → compares bin_num1 and bin_num2 and returns binary number with 1's in places only where bin_num1 and bin_num2 BOTH have 1's; returns number less than or equal to smaller of the two
- bin_num1 | bin_num2 → returns a number where the bits of that number are turned on if either of the corresponding bits of either number are 1; returns number greater than or equal to the larger of the inputs
- bin_num2 ^ bin_num2 → XOR/exclusive operator, returns number where the bits are turned on if either of the corresponding bits of the two numbers are 1, but not both
- ~bin_num → NOT operator, flips all the bits in a single number, equivalent to adding 1 and making the number negative
- can use masks to find out if the bits in a given number are turned on, or to turn them on, useful operators:
  - & → check if bit is on, have bit you want to check = 1 (with all other bits as 0) as second input, bit is on if result is greater than 0
  - | → turn a bit on if it isn't already, have number with bit you want to turn = 1 on as second input (all other bits as 0)
  - ^ → useful for flipping a bit, have a number with all of the bits you want to flip = 1 as second input

---

Classes

- class NewClass(object): → creates a class called NewClass which inherits from the object class
- pass → placeholder for classes or objects that haven't yet been initialized
- can add variables outside of the init function to make it the same across all members of a class
- functions within classes are called methods, need to pass in self, e.g.

```
        def description(self):
            print self.name
            print self.age
    hippo.description()
```

- Example:

```
class Triangle(object):
    def __init__(self, angle1, angle2, angle3):
        self.angle1 = angle1
        self.angle2 = angle2
        self.angle3 = angle3

    number_of_sides = 3

    def check_angles(self):
        return self.angle1 + self.angle2 + self.angle3 == 180

my_triangle = Triangle(90, 30, 60)

class Equilateral(Triangle):
    angle = 60
    def __init__(self):
        self.angle1 = self.angle
        self.angle2 = self.angle
        self.angle3 = self.angle

print my_triangle.number_of_sides
print my_triangle.check_angles()
```

- Instances of a class:
  - `newObject = ClassName()` → creates object newObject as an instance of ClassName
- member variables: variables declared within a class, accessed with dot notation outside the class
  - `newObject.name` → if name was a member variable, would access the member variable value for newObject
- init function allows additional imputs
- don't need to include self keyword when instantiating a class because it gets added to the list of inputs automatically

---

File I/O

- `f = open("output.txt", "w")` → opens output.txt in write only mode
  - w → write only
  - r → read only
  - r+ → read and write
  - a → append
- `my_file.write("Data to be written")` → writes "Data to be written" to `my_file` if it's already open

- my_file.read() → reads entire file
- my_file.close() → closes file, ALWAYS CLOSE FILE AT END
- my_file.readline() → returns next line of the file every time it's called
- file objects contain a special pair of built-in methods: _ _enter_ _() and _ _exit_ _(), when exit is invoked it automatically closes the file, invoke using:

```
with open("file", "mode") as variable:
    #Read or write to the file
```

- file_name.closed → check if file is closed