Python I

Object-oriented programming

Python is an object-oriented programming (OOP) language. Simply put, this means that data structures (e.g. variables) have certain properties, in particular **attributes** and **methods**. Attributes are usually descriptive features of the object, and methods are certain actions (functions) you can perform on the object itself. For example, consider a book. Books are physical objects with certain **attributes**, such as number of pages, number of words, dimensions (width, height, depth), cover art, etc. Similarly, there are several actions (**methods**) that we can do with books like reading, writing, throwing at people we hate, and sharing with people we love. A given realization of an object is called an **instance**. To continue with the book metaphor, "The Count of Monte Christo" and "Harry Potter" are each instances of book objects.

While these concepts might seem abstract, python syntax and behavior will make much more sense in light of the OOP paradigm upon which the language is based. In particular, attributes and methods are accessed in systematic ways, as follows:

```
>>> # Assume mybook is a book instance

>>> # Call an attribute
>>> mybook.number_of_pages
>>> mybook.height
>>> mybook.width

>>> # Call a method
>>> mybook.read()
>>> mybook.write()
>>> mybook.throw_at_bad_people("Voldemort") # methods might take particular argume
nts, in this case who we're launching books at
```

As you see, attributes and methods are called after a . at the end of the instance name. Unlike attributes, however, methods end with parentheses. Sometimes, methods can take particular **arguments** which relate to their function, for instance when we threw the book at Voldemort.

Operators

Mathematical operators

First, we have the standard operators for addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). Additional mathematical operators include,

1. Modulus operator `%`, which gives the remainder, e.g. `12 % 5` will result in `2`.

2. Exponent operator, `**`, e.g. `5**2` will result in 25.

Logical operators

In addition to symbols used for basic calculations, python (like other languages!) has a series of symbols used to compare statements in a True/False context. Such statements are called **logical statements**, and the symbols we used to compare them are called **logical operators**. The most commonly used logical operators include,

Symbol	What it does	Example
==, is	Equals	`5 == 5` results in True `9 is 9` results in True `5 is 7` results in False
!=, is not	Not equals	`5 != 5` results in False `5 is not 7` results in True
>	Greater than	`5 > 6` results in False `11 > 6.23` results in True
>=	Greater than or equal to	`5 >= 4` results in True
<	Less than	`4 < 5` results in True
<=	Less than or equal to	`4 <= 5` results in True

As the table above indicates, the symbol == is equivalent to using the word `is`, and similarly for the operation not equals, `!=` is equivalent to using the words `is not`. And above all, it is **very** important to remember that the equals logical comparison requires a **double equals sign** (==) – a single equals signs indicates variable assignment (see below).

We can also perform multiple comparisons at once, using the keywords `and` and/or `or`. For example,

```
>>> # For "and," both logical statements must be true to result in True
>>> 5 == 5 and 6 == 6
True
>>> 5 > 7 and 8 < 10
False

>>> # For "or," at least one logical statement needs to be true to result in True
>>> 5 > 7 or 8 < 10
True

>>> # We can use the keyword "not" to negate a logical statement
>>> 3 < 11 and not 4 >= 7
True
```

Variables

We assign values to a variable using the equals sign, =.

```
>>> a = 5
>>> # Check that the variable was correctly assigned using the "print" statement
>>> print a
5
```

All variables have a certain **type**. The variable type deterines what can be done with the variable itself. Standard python types include the following,

Variable Type	Description	Casting
integer	whole number	int()
float	decimal number	float()
string	ordered, immutable character container	str()
list	ordered, mutable container	list()
dictionary	unordered, mutable container	dict()
tuple	ordered, immutable container	tuple()

Remember, every time you create a variable, you are essentially creating an **instance** of that particular variable type. As a consequence, there are certain properties (attributes and methods!) associated with each type of variable.

Integers and floats

Integers and floats are python's primary types for dealing with numbers.

Integers are whole numbers only, but floats include decimal places. Whether a variable is an integer or float turns out to matter a lot – if you perform an operation with integers, the result will be an integer (even if the "real" answer is actually a float!)

```
>>> a = 6
>>> type(a)
<type 'int'>
>>> # We can change the type of a variable by using casting
>>> a = float(a)
>>> type(a)
<type 'float'>

>>> # Caution! Variables can easily be reassigned!
>> a = 11
>>> print a
11
```

```
>>> # By adding a decimal point during assignment, we force the variable to be a f
loat
>>> b = 6.
>>> type(b)
<type 'float'>

>>> # Be careful! If you perform operations with only integers, the result will al
ways be an integer (rounding determines answer)
>>> x = 5
>>> y = 6
>>> x/y
0
>>> # One way to circumvent this issue is by casting the result as a float
>>> float(x / y)
0.8333333333333334
>>> # Note that the above division will NOT change the casting of either x or y th
emselves
>>> x / y
0
>>> # Another solution would be to define either/both x or y as a float from the b
eginning
>>> x = 5.
>>> y = 6
>>> x / y
0.8333333333333334
```

Strings

In python, a string is an **immutable** container of **characters**. By "immutable", we mean that it can't be changed - that is, once you create a string, you can't rewrite certain parts of it. It's an all-or-nothing thing. By characters, we basically mean "not numbers" - consequently, no mathematical operations can be done on strings. Strings are also **ordered** - python remembers the orders of values used in the list. This means we can **index** items in a list, again using brackets `[]`. **Importantly**, python indexes **starting at 0**, meaning the first item in a given string is the 0th entry.

```
>>> # Assign strings using quotation marks
>>> name = "Stephanie"
>>> type(name)
<type 'str'>
>>> # Find length with function len() [note that this function works for most vari
able types!!]
>>> len(name)
9

>>> # Numbers can also be strings!
>>> age = "26"
>>> type(age)
<type 'str'>
```

```
>>> # Even though 26 is a number, we set up the variable age as a string, so no ma
th can be performed with this variable
>>> age - 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'int'

>>> # But!! Since the value 26 is, in fact, a number we can recast age as an integ
er or float and then do maths with it!
>>> int(age) - 7
19
>>> # Again, age is still a string. We'd have to redefine the variable itself to m
ake it an integer (or float) for good
>>> type(age)
<type 'str'>
>>> age = int(age)
>>> type(age)
<type 'int'>

>>> # We can't recast the variable name though, since letters just aren't numbers.
>>> name = float(name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: Stephanie

>>> # Strings are ordered, so we can index them
>>> name[5]
a
>>> # But strings are also immutable, so we can't edit strings in place.
>>> name[5] = "A"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Some useful string methods:

```
>>> my_string = "This is a really nice string for showing examples."

>>> # .upper() converts your string to upper-case
>>> my_string.upper()
'THIS IS A REALLY NICE STRING FOR SHOWING EXAMPLES.'

>>> # .lower() converts your string to lower-case
>>> my_string.lower()
'this is a really nice string for showing examples.'

>>> # .split() takes an *argument* to split the list on and creates a list contain
ing each chunk
>>> my_string.split('i')
['Th', 's ', 's a really n', 'ce str', 'ng for show', 'ng examples.']

>>> # .strip() removes both leading/trailing whitespace, .rstrip() removes only tr
```

```
ailing whitespace, and .lstrip() removes only leading whitespace
>>> # Instead of whitespace, you could also provide an argument to one of these fu
nctions to remove instead
>>> dna_string = "AAAAAGTCGAGGTAGCGAAAA"
>>> dna_string.strip("A")
'GTCGAGGTAGCG'
>>> dna_string # but remember, since strings are immutable, calling .strip() will
not change the dna_string contents!
'AAAAAGTCGAGGTAGCGAAAA'

>>> # .count(arg) returns the number of times "arg" appears in the string
>>> dna_string.count("A")
11

>>> # .replace(old, new, count) replaces all instances of old with new. Last optio
nal argument, count, indicates that the only the first count occurences of old shou
ld be replaced (default - all)
>>> dna_string.replace("T", "U")
'AAAAAGUCGAGGUAGCGAAAA'
>>> dna_string.replace("T", "U", 1)
'AAAAAGUCGAGGTAGCGAAAA'
```

Lists

Lists are defined using brackets (`[]`), and each list item can be any variable type.

```
>>> # This list contains only integers
>>> numbers = [1,2,3,4,5]

>>> # This list contains integers and floats and strings, oh my!
>>> crazy_list = [5, 77.2, -9, "word", -1.32, "more words"]
```

Python lists are incredibly flexible. Like strings, they are ordered, so they support indexing. However, unlike strings, lists are **mutable**, meaning they can be changed! List items can be removed, changed, and new list items can even be added after they are defined. In other words, lists can be changed *in place without (re-)assigning anything*.

```
>>> simple = [1,4,9,2,5,11]
>>> simple[4] # grab the 4th entry in this list
2
>>> simple[15] # what happens when the entry doesn't exist? you get an error messa
ge!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> # Rewrite the list in place! You don't need to redefine simple - just modify a
n element using indexing.
>>> simple[2] = 888
simple
```

```
[1,4,888,2,5,11]

>>> # Some useful functions you can use on a list. These will NOT modify the list
in any way.
>>> len(simple) # How many elements in a list?
6
>>> sorted(simple) # sort the items in ascending order
[1, 2, 4, 5, 11, 888]
>>> max(simple) # but probably only use this when the list has only numbers. (note
 that min() is also around!)
888
```

Some useful list methods:

```
>>> # .append(value) adds value to the end of a list, ultimately modifying the lis
t in place!
>>> simple.append(100.33)
simple
[1,4,888,2,5,11,100.33]
>>> len(simple) # Look, the list length changed!
7

>>> # .remove(value) removes all entries in a list corresponding to the provided a
rgument, value
>>> simple.remove(11)
[1,4,888,2,5,100.33]

>>> # .insert(index, value) inserts the provided value into the specified index of
 the list
>>> simple.insert(0, 500)
[500,1,4,888,2,5,100.33]

>>> # .index(value) returns the index for a given value
>>> simple.index(888)
2
```

Let's delve a bit more into indexing now. In general, indexing follows the paradigm `container[x:y:z]`, where x is the starting index, y is the ending index, and z is the step. However, you do not need to provide all of these value to index. In fact,

```
>>> fib_list = [0,1,1,2,3,5,8,13,21,34,55,89]

>>> # Select the 3rd item
fib_list[3]
1

>>> # Select the 3rd from last item with a negative index
fib_list[-3]
34

>>> # Select multiple items with the syntax [x:y], where x is the starting index a
```

```
nd y is the ending index. Note that y is *not* included!
>>> # Select items indexed 1-4
>>> fib_list[1:5]
[1, 1, 2, 3]


>>> # If you don't provide x or y, python defaults to either the first or last ind
ex
>>> fib_list[:5] # same as writing fib_list[0:5]
[0, 1, 1, 2, 3]
>>> fib_list[5:] # same as writing fib_list[5:12]
[5, 8, 13, 21, 34, 55, 89]

>>> # Change the step of indexing with the format x:y:z (before, z was defaulted t
o 1!)
>>> fib_list[2:10:3]
[1, 5, 21]
```

Dictionaries

Dictionaries are defined using braces ({}), and they are essentially **unordered** lists of key:value pairs, and they are python's version of "associative arrays." Keys and values can be any type, although typically keys are either integers, floats, or strings. Dictionaries are incredibly useful for storing information; all keys must be unique, but values may be repeated.

```
>>> taxonomy = {'gecko':'vertebrate', 'human':'vertebrate', 'squid':'mollusk', 'bu
tterfly':'insect', 'oak tree': 'plant'}
```

Dictionaries are indexed using keys. As dictionaries are unordered, the particular order (e.g. gecko, vertebrate, squid..) is not preserved, but the key:value pairs are fixed.

```
>>> taxonomy["gecko"]
'vertebrate'
>>> # Add a new key:value pair
>>> taxonomy["e. coli"] = "bacteria"
>>> taxonomy
t

>>> # the method .keys() pulls up all dictionary keys as a list
taxonomy.keys()
['butterfly', 'oak tree', 'squid', 'e. coli', 'human', 'gecko']

>>> # the method .values() pulls up all dictionary values as a list
taxonomy.values()
['insect', 'plant', 'mollusk', 'bacteria', 'vertebrate', 'vertebrate']

>>> # the methods .items() pulls up all key:value pairs as tuples
>>> taxonomy.items()
[('butterfly', 'insect'), ('oak tree', 'plant'), ('squid', 'mollusk'), ('e. coli',
'bacteria'), ('human', 'vertebrate'), ('gecko', 'vertebrate')]
```

```
>>> # Values can be all kinds of things, even lists!
>>> meals = {"breakfast": ["coffee", "cereal", "banana"], "lunch": ["salad", "lemo
nade", "chicken fingers"], "dinner": ["steak", "asparagus", "beer", "more beer"], "
dessert": ["ice cream", "chocolate sauce", "sprinkles"] }
```

Tuples

Tuples are basically immutable lists, created with parentheses `()`. We won't use them much, but they're nice to be aware of! Aaand that's all we'll say for now.

Useful functions

Some fun things we can do with these variables!

```
>>> # the range() function creates an arithmetic list, by default starting from ze
ro and with a step of 1
>>> simple_range = range(20)
>>> simple_range
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> complex_range = range(5, 20) # start from 5
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> complexer_range = range(5, 20, 3) # start from 5 with a step of 3. look famili
ar??
[5, 8, 11, 14, 17]

>>> # "sep".join(list) will join items in list as a string together, with items se
parated by "sep"
>>> my_list = ["look", "this", "is", "a", "list"]
>>> " ".join(my_list)
'look this is a list'

>>> # Merge two lists together into a single list of paired tuples with the functi
on zip()
>>> loci = ["locus1", "locus2", "locus3", "locus4"]
>>> snps = ["A/C", "T/G", "G/C", "A/G"]
>>> zip(loci, snps)
[('locus1', 'A/C'), ('locus2', 'T/G'), ('locus3', 'G/C'), ('locus4', 'A/G')]
>>> # Turn these zipped lists into a dictionary!
>>> dict(zip(loci, snps))
{'locus1': 'A/C', 'locus2': 'T/G', 'locus3': 'G/C', 'locus4': 'A/G'}
```

Print statements

Printing information from a script to "stdout" is a critically important compontent of scripting and programming. Using print statements, you can determine if your code is actually doing what you think it is. Most of the time, your code will have some issues and will need to be "debugged." Printing to screen is one of the best and easiest strategies for ensuring that your code is working as intended.

```
>>> # Define a variable and print
>>> a = 6
>>> print a
6
```

You can more or less print anything to screen, and importantly you can print multiple things in the same statement!

```
>>> # Define a variable and print
>>> mystring = "I am writing a full sentence here as a string variable, which I wi
ll save!"
>>> print mystring
I am writing a full sentence here as a string variable, which I will save!

>>> # print two strings together with +
>>> print "Here is my string: " + mystring

>>> + can only be used to join strings when printing!
>>> print mystring  + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> print mystring  + "2"
I am writing a full sentence here as a string variable, which I will save!2

>>> # Use , to print multiple variables, values of different types. Spaces will au
tomatically be inserted.
>>> print "My variable contains", mystring, " Yay! here's a number!", 888
My variable contains I am writing a full sentence here as a string variable, which
 I will save!  Yay! here's a number! 888


# woah all the printing!
>>> a = 55
>>> b = 44.6
>>> mylist = [1,2,3]
>>> print "oh man, I got some numbers:", a, "," b, ",", and even a list:", mylist
, ", and even a string!: ", mystring
oh man, I got some numbers: 55 , 44.6 , and even a list: [1, 2, 3] , and even a st
ring!:  I am writing a full sentence here as a string variable, which I will save!
```