EASC2410 Lecture 2

Python Basics: Variables, Operations, and introduction to Data structures

Dr. Binzheng Zhang
Department of Earth Sciences

Review of Lecture 1

In Lecture 1, we learned how to:

- **What is data science in general**
- **How to open Jupyter notebooks**
- **Insert and execute code in cells**
- **Write your “hello world” program use the print() function**
- **Get help use the build-in help() function**
- **Use your Jupyter notebook as a calculator**
- **Use # in code blocks to comment**

In Lecture 2, you will learn:

- **Variables and types in Python**
- **lists and Operations in Python**

Name your Variables

How to name your variables: some rules for “best practices” here:

1. Variable names are composed of alphanumeric characters, including '-' and '_'.
2. They are case sensitive: 'a' is not the same as 'A'.
3. There are some *reserved words* in Python that you may not use as your variable names because they have pre-defined meanings (for example, *False* or *True*)

A list of **reserved** words:

and	assert	break	class	continue
def	del	elif	else	except
exec	finally	for	from	global
if	import	in	is	lambda
not	or	pass	print	raise
return	try	while		

Do **NOT** use any of the following words either

Data	Float	Int	Numeric	
array	close	float	int	input
open	range	type	write	zeros
acos	asin	atan	cos	e
exp	fabs	floor	log	log10
pi	sin	sqrt	tan	

4. Not too many rules for naming your variables except for avoiding reserved words, but in general, longer more descriptive words are better because they'll remind you what the variable stores.
5. Here are some "best practices" for variable names:
   https://www.python.org/dev/peps/pep-0008/#naming-conventions

Name your Variables

To name your variables properly, here are some popular choices:

- lowercase - use this for variables
- lower_case_with_underscores - or this
- mixedCase - or this
- UPPERCASE - use this for constants
- UPPER_CASE_WITH_UNDERSCORES - or this
- CapitalizedWords (or CapWords, or CamelCase -- so named because of the bumpy look of its letters).
- Capitalized_Words_With_Underscores - usually we don't use this, it's ugly!

Also, some things to avoid:

- Don't use characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names. These are easily confused in some fonts with '1' (one), '0' (zero), for example. If you really want a letter 'el', use 'L'.
- Don't use non-standard symbols like ‘¥’ or ‘。’
- Be careful with special names that use leading or trailing underscores. These are treated differently by Python and you have to know what you are doing before you use them. e.g., ‘_mac_’

Variables

What do variables to: store values

Variables have types, Here are the basic variable types in Python, including:

types	definitions	examples
integers	numbers without a decimal	2, 4, 38
floating point	numbers with a decimal	3.14, 2.0
string	numbers and/or letters enclosed in quotation marks	“Earth”
booleans	True or False	true, false
complex numbers	numbers have real and imaginary parts	2+3j

Variables: Integer (`int`)

In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be:

In [4]:
```
# Lecture 2, Variables, Lists, Operations
#
# In Python 3, there is effectively no limit to how long an integer value can be.
# Of course, it is constrained by the amount of memory your system has, as are all things,
# but beyond that an integer can be as long as you need it to be:

print(123123123123123123123123123123123123123123123321312321233 + 1)
```

Note: Python interprets a sequence of decimal digits without any prefix to be a decimal number:

In [5]:
```
# Python interprets a sequence of decimal digits without any prefix to be a decimal number:

print(10)
```

Python can also do non-decimal numbers:

Prefix	Interpretation	Base
`0b` (zero + lowercase letter `'b'`)	Binary	2
`0o` (zero + lowercase letter `'o'`)	Octal	8
`0x` (zero + lowercase letter `'x'`)	Hexadecimal	16

In [9]:
```
# Python can also do non-decimal numbers
# this line prints out octal, hexadecimal and binary number 10,
# respectively
print(0o10, 0x10, 0b10)
```

Variables: Floating-point numbers (`float`)

The `float` type in Python designates a floating-point number. `float` values are specified with a decimal point.

In [4]:
```
# Floating-point numbers

print(3.14)
print(3.)
print(.3)
```

```
3.14
3.0
0.3
```

Optionally, the character `e` or `E` followed by a positive or negative integer may be appended to specify scientific notation:

In [6]:
```
# Optionally, the character e or E followed by a positive or negative integer
# may be appended to specify scientific notation

print(4.7e2)
print(3.9e0)
print(1.6e-19)
```

```
470.0
3.9
1.6e-19
```

Deep Dive: What's the largest and smallest Floating-Point numbers in your computer?

Almost all platforms represent Python `float` values as 64-bit "double-precision" values, according to the IEEE 754 standard. In that case, the maximum value a floating-point number can have is approximately 1.8×10^{308}. Python will indicate a number greater than that by the string `inf`:

In [18]:
```
# Almost all platforms represent Python float values as 64-bit "double-precision" values,
# according to the IEEE 754 standard. In that case, the maximum value a floating-point number
# can have is approximately 1.8 X 10^308. Python will indicate a number greater than that by the string "inf"

print("the maximum floating-point number is ", 1.79e308)
print("try something large that 1.79e308 is gonna be", 1.8e308)
```

```
the maximum floating-point number is  1.79e+308
try something large that 1.79e308 is gonna be inf
```

Variables: Strings (`str`)

- Strings are sequences of character data. The string type in Python is called `str`.
- String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

In [8]:
```
# Strings are sequences of character data. The string type in Python is called "str".
# String literals can be delimited using either single or double quotes.
# All the characters between the opening delimiter and matching closing delimiter are part of the string:

print("I am a string.")
print('I am too.')
```

```
I am a string.
I am too.
```

- A string in Python can contain as many characters as you wish, limited by your computer's memory resources.
- A string can also be empty:

In [8]:
```
# A string can also be empty
print('')
```

Nothing happens if you run the above code, because it's empty

- To include special characters such as ' and ", use the backslash (\) symbol:

In [9]:
```
# Specifying a backslash in front of the quote character in a string "escapes" it
# and causes Python to suppress its usual special meaning.
# It is then interpreted simply as a literal single quote character.
# The same syntax works in a string delimited by double quotes as well:

print('This string contains a single quote (\') character.')
print("This string contains a double quote (\") character.")
```

```
This string contains a single quote (') character.
This string contains a double quote (") character.
```

Variables: Boolean (`bool`)

- Python 3 provides a Boolean data type. Objects of Boolean type may have one of two values, `True` or `False`:

- As you will see in upcoming tutorials, expressions in Python are often evaluated in Boolean context, meaning they are interpreted to represent truth or falsehood. A value that is true in Boolean context is sometimes said to be “truthy,” and one that is false in Boolean context is said to be “falsy.”

In [10]:
```
# Python 3 provides a Boolean data type. True = 1 and False = 0 when working with integer and/or float numbers
# Objects of Boolean type may have one of two values, True or False.
# As you will see in upcoming tutorials, expressions in Python are often evaluated in Boolean context,
# meaning they are interpreted to represent truth or falsehood.
# A value that is true in Boolean context is sometimes said to be “truthy”,
# and one that is false in Boolean context is said to be “falsy”.
#
# Here are some examples of how the Boolean type looks like

print(True, type(True))
print(True*2.0, type(True*2.0))
print(False, type(False))
print(True+False, type(True+False))
print(True*False, type(True*False))
```

You will learn more about evaluation of objects in Boolean context when you encounter logical operators in the upcoming tutorial on operators and expressions in Python.

Here we used a new Python build-in function named “`type()`”, which gives you the type of the variable goes in to the “()”

```
In [12]: print(True,type(True))
```

First evaluate the type of “True”

Then prints out two objects, True and the type of “True”

Here’s the output

```
True <class 'bool'>
```

Variables: Complex Numbers (`complex`)

Complex numbers are specified as `<real part>+<imaginary part>j`. For example:

In [11]:
```
# complex numbers are specified as <real part>+<imaginary part>j
# for example

print(2+3j)
print(1j*1j)
print(type(2+3j))
```

```
(2+3j)
(-1+0j)
<class 'complex'>
```

Recall: complex numbers

define imaginary number i (or j) $i = \sqrt{-1}$

it works this way: $i^2 = \left(\sqrt{-1}\right)^2 = -1$

More examples:

$$\begin{aligned}(2 - i)(3 + 4i) &= (2)(3) + (2)(4i) + (-i)(3) + (-i)(4i) \\ &= 6 + 8i - 3i - 4i2 = 6 + 5i - 4(-1) \\ &= 6 + 5i + 4 = \mathbf{10 + 5i}\end{aligned}$$

Complex numbers are very useful in various physics and engineering applications

Assign values to variables (=)

To assign a value to a specific variable, simply use the **equal sign** (=):

In [13]:
```
# Let's practice variable types
# Fisrst define the following variables and assign values using "="
number = 1              # an integer
Number = 1.0            # a floating point - notice the decimal point
NUMBER = '1'            # a string - notice the quotation marks
my_string = "HKU"       # a string with three characters, double quotes are also ok
morecomplex = 3 + 1j    # a complex number 3+1i
bools=True              # A boolean variable (True/False or 1/0)

# now use the print() and type() functions to find the type for each variable
# put you code here:
```

Recall the operators we learned in the last lecture:

operation symbol	function
+	adds
-	subtracts
*	multiplies
/	divides
%	gives the remainder
**	raises to the power
==	test equality
!=	test inequality

Let’s try some operations on our variables

Simple Variable Operations

Remember we have already defined the following variables, find the answers of the following variable operations use your Jupyter notebook (pay attention to the **upper** and **lower** case variable names)

```
number = 1              # an integer
Number = 1.0            # a floating point - notice the decimal point
NUMBER = '1'            # a string - notice the quotation marks
my_string = "HKU"       # a string with three characters, double quotes are also ok
morecomplex = 3 + 1j    # a complex number 3+1i
bools=True              # A boolean variable (True/False or 1/0)
```

1. `number + number` — `>>> out [ ]: 2` — 1+1=2, don't ask me why

2. `number + number`
   `number + number` — `>>> out [ ]: 2` — if the LAST statement in the code block is not assigned to a variable, then your notebook will print the outcome.

3. `print( number + number )`
   `print( number + number )` — `>>> out [ ]: 2` `2` — you get two outputs

4. `number + Number` — `>>> out [ ]: 2.0` — integer + float = float

5. `number + NUMBER` — `>>> out [ ]: TypeError` — integer + str = none sense

6. `print( Number, int(Number) )` — `>>> out [ ]: 1.0 1` — int(Number) converts "Number" from float to int

7. `print( number, float(number) )` — `>>> out [ ]: 1 1.0` — float(number) converts "number" from int to float

8. `print( number, str(number) )` — `>>> out [ ]: 1 1` — str(number) converts "number" from int to str

here we've used three new functions, `int()`, `float()` and `str()`, they are brand new, how to get more information of these new functions? try that in your Jupyter notebook and see what you learn.

Useful String Operations

Numbers are just numbers. Strings are much more interesting. They can be denoted with single, double or triple quotes:

In [13]:
```
# String operations
#
# 1. Strings can be denoted with single, double or triple quotes
string1 = 'HKU'
string2 = "is"
string3 = """Great"""

# ====< now write some codes to print out the three strings in three separate rows >====
# put your code here:
```

Try the exercise there.

Strings can also be added together:
```
newstring = string1 + string2 + string3
print (newstring)
```

Then answer is simple:
```
HKUisGreat
```

They can be sliced:
```
newerstring = newstring[0:3]
```

Now, what if I do:
```
print(newerstring) ?
```

The answer is:
```
HKU
```

Why? it's related to how index slicing is done in Python. This is an important feature which you will use throughout the course. So let's see how `newstring[0:3]` works.

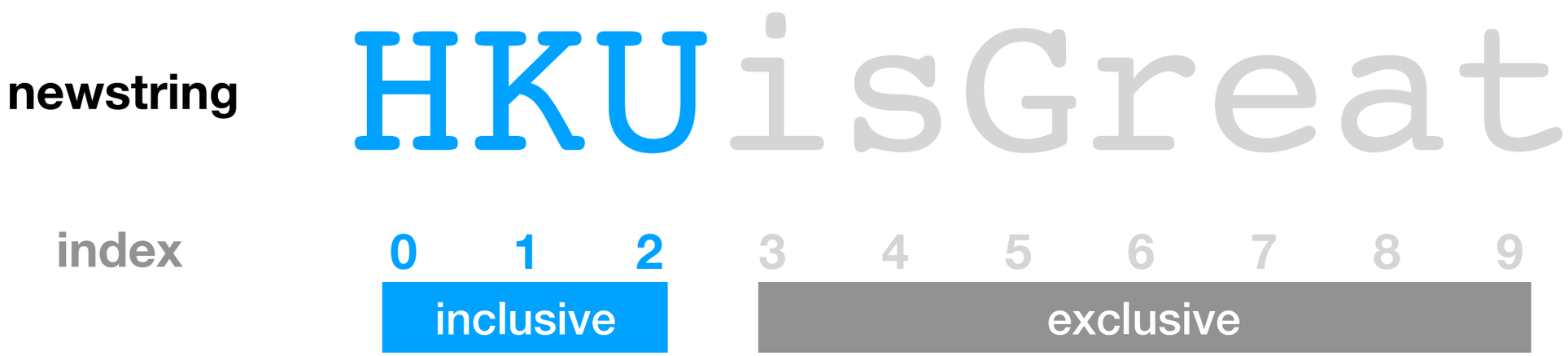
Index slicing in Python

- The variable `newstring` has 10 characters
- The starting index is 0 and the ending index is 9
- The slicing syntax goes like:

```
string_name [index_start : index_end]
```

inclusive exclusive

Now we can see what `newstring[0:3]` gives:



To get the whole string, simply use `newstring[:]`

More Index slicing in Python

newstring **HKUisGreat**

forward index	0	1	2	3	4	5	6	7	8	9

```
string_name [index_start : index_end]
```

inclusive exclusive

```
string_name [index_start :]
```

inclusive Slice all the way to end

```
string_name [: index_end]
```

Slice all the way from beginning exclusive

```
string_name [:]
```

Gives you the whole string

Can strings be changed in place?

Since we have learned how to access a part of a string using index slicing, what happens if you do the following code in your Jupyter notebook?

```
newstring[5:]='Awful'
```

Shall we get

```
HKUisAwful
```

The answer is no (of course not!) Here's what you will get from Jupyter notebook:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-0a7e9bbd8f33> in <module>()
----> 1 newstring[5:]='Awful'

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

Yup, that made Python mad (probably made you mad too…). That means Strings CANNOT be changed in place.

To find more of the things you can and cannot do to strings, see: http://docs.python.org/tutorial/introduction.html#strings

Data structures: group variables together

How? In Python there are many ways to do this: **lists, tuples**, **dictionaries**, and **sets**, among others. These group arbitrary variables together, (e.g., strings and integers and floats) in containers.

Lists

- Lists are denoted with `[ ]` and can contain any arbitrary set of elements (including other lists)
- Elements in a list are referenced by an index number. Similar to indexing strings we learned before, the indices begin at 0.
- Negative indexing: you can count from the end to the beginning by starting with -1 (the last item in the list), -2 (second to last), etc.
- Lists have *methods* that allow items to be sorted, deleted, inserted, sliced, counted, concatenated, replaced, added on, etc. (will learn later)

[14]:
```
# examples on Lists
#
mylist=['a', 2.0 ,'400' ,'spam', 42, [24,2]] # defines a list
print(mylist, type(mylist)) # prints the list and its type using the print() and type() functions
```

```
['a', 2.0, '400', 'spam', 42, [24, 2]] <class 'list'>
```

Now let's try some index sliding on lists

	['a',	2.0,	'400',	'spam',	42,	[24, 2]]
index	0 -6	1 -5	2 -4	3 -3	4 -2	5 -1
element type	str	float	str	str	int	list

Unlike strings discussed before, list elements can be changed directly:

In [17]:
```
# Unlike strings, elements in lists can be change directly using indexing
# for example let's change the second element in mylist to be 26.3
# here's what we do (why index 1 rather than 2?)
mylist[1]=26.3          # replaces the second element
print(mylist)           # print out mylist to check the answer

# you can also delete elements from lists
# for example the following code removes the fourth element
del mylist[3]           # deletes the fourth element
print(mylist)

# Like strings, you can also slice out a chunk of a list, and assign it to another variable:
newlist=mylist[1:3] # takes the 2nd and 3rd values and puts in newlist
print(newlist)
```

Now try access the elements in mylist using your Jupyter notebooks

In [16]:
```
# Remember that mylist is already defined in the above cell,
# now let's practice some operations on lists
#
# ====< print out the third element in "mylist" (Hint: starting from zero) >====
# put your code here:
  print(mylist[2])
# ====< print out the last element in "mylist" >====
# put your code here:
  print(mylist[-1])
# ====< print out the last three in "mylist" >====
# put your code here:
  print(mylist[-3:])
```

List copy can be somewhat different

In [18]:
```
mycopy=mylist # mycopy is now a "copy" of mylist
              # However it is inextricably bound to the original,
              # so if you change one, you will change the other.
print("After making 'mycopy' a copy of 'mylist': ")
print("mycopy = ", mycopy)
print("mylist = ", mylist)
```

Now let's change the second element in the "copied" list

In [21]:
```
# now change the second element of mycopy to be False:
mycopy[1] = False

# print out both lists "mylist" and "mycopy", you get:
print("After you changed the second element of 'mycopy': ")
print("mycopy = ", mycopy)
print("mylist = ", mylist)
```

Both the second element in mycopy and mylist are changed! Why is that? - it's related to how the lists are referenced in the memory system.

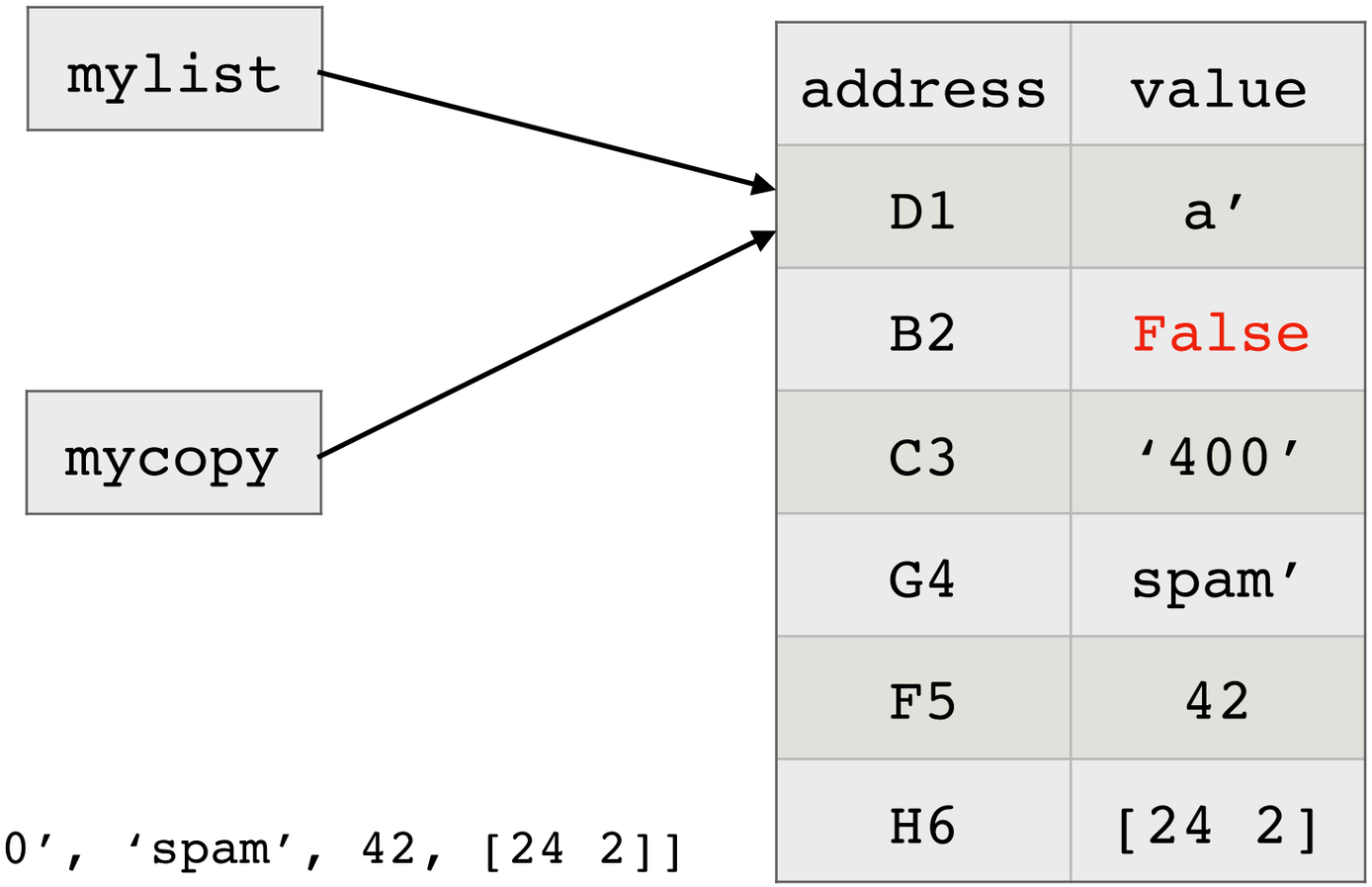
How lists are referenced in your computer memory

To get an independent copy of mylist, simply do this:

```
mycopy = mylist[:]
```

or:

```
mycopy = list(mylist)
```




```
mylist = ['a', 2.0, '400', 'spam', 42, [24 2]]
```

```
mycopy = mylist
```

The direct copy using "=" links the two list through the address where the data values are stored, now when you do:

```
mycopy[1] = False
```

mylist[1] is also changed since the two lists are pointing to the same address in the memory, and mylist becomes:

```
mylist = ['a', False, '400', 'spam', 42, [24 2]]
```

Making a numeric lists using the `range()` function

Another way to generate a list is to use the function called **`range( )`**, which is one of the python *built-in functions*

`list( range( start, end, interval) )`

inclusive exclusive

Try the following example:

In [22]:
```
# now use the range() function to create lists with numbers
# e.g., creates a list from 2 to 20 (not including 20!) at intervals of 4
numlist=list(range(2,20,4))
print (numlist)

# ====< now creat a new list named 'newlist', from 2 to 20 (including 20) at intervals of 2 >==
# put in your code here:
```

```
[2, 6, 10, 14, 18]
```

Another useful Data structure: Tuples

Tuples are another important data structure in python that are similar to lists, but have significant differences. They are denoted by parentheses ( ) and consist of values separated by commas.

Like lists, *Tuples* can contain different elements;

Like both lists and strings, Tuples can be sliced, concatenate, etc.;

Unlike lists, the elements in *Tuples* **CANNOT** be changed in place.

For more info about Tuples, see: http://docs.python.org/tutorial/datastructures.html#tuples-and-sequences

For example:

In [24]:
```
# Here's another important data structure in Python called "Tuple"
# Similar to lists but dennoted by prentheses ()
# Elements in tuples cannot be changed, you can regard tuples as constant lists
mytuple = (1, '28', 3.3e-2, False, [-2, 3])
print(mytuple)

# you can slice or access elements in tuples using indices denoted by []:
mytuple_slice = mytuple[2:4]
print(mytuple_slice)

# you can add two tuples together like
mytuple = mytuple + (2, '3', True)
print(mytuple)

# but you can't change elements in place
mytuple[2] = 'let me change it'
```

Another useful Data structure: Sets

- There is more data structure that comes in handy and one is the *set*. They are denoted with curly braces { }.
- A set "contains an unordered collection of unique and immutable objects.”
- You can create *sets* in several ways, e.g., use the python built-in **set( )** function on a list, or just use the curly braces {}. Here are some example of how to creating sets in Python:

In [30]:
```
# There are more data structures that comes in handy in Python, and one is the set.
# They are denoted with curly braces { }.
# Definition: A set "contains an unordered collection of unique and immutable objects."

# You can create sets in several ways. The first would be the use the python built-in set( ) fu
# For example:
myset1 = set(['Earth',9.8, 'Saturn', 10.3, 'Jupiter', 24.6])
print('myset1 = ', myset1) # print out the sets see how elements are changed

# you can also define a set simply using curly brackets {}
myset2 = {'Mercury',5.4, 'Venus', 8.9, 'Earth', 9.8, 'Mars', 3.9}
print('myset2 = ', myset2)

# if you violate the uniqueness requirement of creating a set:
myset2=set(['Earth',9.8, 'Saturn', 10.3, 'Jupiter', 24.6, 'Jupiter', 12.2])
print('myset3 = ', myset2) # see how many 'Jupiter's are there in myset2
```

- A set is like a list, but it’s unordered, and the elements must be unique (ignored if you have duplicated elements

Introduction to Objects and Methods

Now we've learned variables and lists, they are quite straightforward. One distinctive feature of Python variables and lists (also other data structures) is that they are all ***Objects***. It's a little bit unfamiliar but it's also straightforward to understand. Once you get used to them, you'll see why Python is awesome.

So what is an ***object*** in Python?

Definition: An object in Python is a collection of *attributes* and *methods*

For example, the variable "mylist" we've been working on for a while is an object, which is known as a "list object".

So what is so special about *objects*? Python objects have *methods* which allow you to do things to the object. Methods are called in the following the form using the "dot" sign:

```
object.method( )
```

or

```
object.method(parameter1, parameter2, …)
```

For example, if you type in `help(list)` in you Jupyter notebook cell, you will get all the methods that can be applied to any list objects. Let's use the `append()` method as an example.

```
In [20]: mylist.append(-18)
         mylist

Out[20]: ['a', False, '400', 42, [24, 2], -18]
```

-18 is now added to the end of the list object "mylist"

Introduction to Objects and Methods

A couple of handy methods that are used frequently in the list objects:

In [28]:
```
print(mylist.count(False)) # returns the number of times the argument occurs in the list.
print(mylist.index(42))    # returns the position(index) of the argument in the list
```

```
1
3
```

These are just a few of the methods for lists. To view all of the methods for a list object, see: https://docs.python.org/3.6/tutorial/datastructures.html

A couple of handy methods that are used frequently in the set objects:

In [38]:
```
# a couple of useful methods for a set object:

# clear an existing set
myset1.clear()
print('now myset1 = ', myset2)

# copy a set to be a new one, NOTE: the new set is independent, not like lists
myset3=set(['Mars',3.9, 'Saturn', 10.3, 'Jupiter', 24.6]) # re-define myset3
myset_steal = myset3.copy()
print('my stealing set is', myset_steal)

# add an element in an existing set (like the 'append' method in lists):
myset3.add('Uranus')
print('now myset3 = ', myset3)

# show the difference between two sets
diff = myset3.difference(myset_steal)
print('the difference between myset3 and myset_steal is: ', diff)

# you can also see what the same elements are in two sets:
inter = myset3.intersection(myset_steal)
print('the intersection between myset3 and myset_steal is: ', inter)
```

```
now myset1 =  {3.9, 5.4, 8.9, 'Earth', 9.8, 'Mercury', 'Venus', 'Mars'}
my stealing set is {'Mars', 3.9, 24.6, 'Jupiter', 10.3, 'Saturn'}
now myset3 =  {3.9, 10.3, 'Uranus', 'Mars', 24.6, 'Jupiter', 'Saturn'}
the difference between myset3 and myset_steal is:  {'Uranus'}
the intersection between myset3 and myset_steal is:  {3.9, 10.3, 'Mars', 24.6, 'Jupiter', 'Sa
turn'}
```

Knowledge Check - Game Time!

Step 1. Go to www.kahoot.it either on your phone or computer

Step 2. Type in the Game PIN shown on screen

Step 3. Name yourself by using:

the first letter of your first name + last three digits of your UID, e.g. B519

Step 4. Click “OK go!”