# Python Basics

In this lecture, we will learn about Objects and Data Structures Python and how to use them.

We'll learn about the following topics:

```
1.) Types of numbers in Python
2.) Variable Assignment and Working with Numbers
3.) Strings
4.) Lists
5.) Dictionaries
6.) Tuple
7.) Sets
```

## Types of Numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Here is a table of the two main types we will spend most of our time working with some examples:

| Examples | Number "Type" |
|---:|---:|
| 1,2,-5,1000 | Integers |
| 1.2,-0.5,2e2,3E2 | Floating-point numbers |

# Variable Assignment and Working with Numbers

We will start with assigning values to variables. the values can be of "integer" type or "float" type. Please consider the following rules before naming the variables.

1. Names can not start with a number.
2. There can be no spaces in the name, use _ instead.
3. Can't use any of these symbols :'",<>/?|\()!@#$%^&*~-+
4. It's considered best practice that names are lowercase.
5. Avoid using words that have special meaning in Python like "list" and "str"

Using variable names can be a very useful way to keep track of different variables in Python. For example:

In [52]:
```
#assigning different number types to different variables
my_income = 100
tax_rate = 0.1
```

In [53]:
```
#Printing my_income
print( my_income)
```

```
100
```

In [54]:
```
#performing calculation by using variables
my_taxes = my_income * tax_rate
```

In [55]:
```
# printing the variable
print(my_taxes)
```

```
10.0
```

## Task 1

Calculate the area of a rectangle having length of 15 cm and width of 10 cm. Use variable assignment and perform calculations using variables. Also display the result.

In [56]:
```
# Assign values to variables
length = 15
width = 10
```

In [57]:
```
# Perform Calculation
result = length * width
```

In [58]:
```
addition_of_variables = length + width
print(addition_of_variables)
```

```
25
```

In [59]:
```
# Display the result
print(result)
```

```
150
```

## Booleans

Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0). It also has a placeholder object called None. Let's walk through a few quick examples of Booleans (we will dive deeper into them later in this course).

In [60]:
```
# Set object to be a boolean
a = True
```

In [61]:
```
#Show
print (a)
```

```
True
```

We can also use comparison operators to create booleans. We will go over all the comparison operators later on in the course.

In [62]:
```
# Output is boolean
1 > 2
```

Out[62]:
```
False
```

## Determining variable type with `type()`

You can check what type of object is assigned to a variable using Python's built-in `type()` function. Common data types include:

- **int** (for integer)
- **float**
- **str** (for string)
- **list**
- **tuple**
- **dict** (for dictionary)
- **set**
- **bool** (for Boolean True/False)

In [63]:
```
type(4)
```

Out[63]:
```
int
```

In [64]:
```
type(3.14)
```

Out[64]:
```
float
```

In [65]:
```
type(False)
```

Out[65]:
```
bool
```

In [66]:
```
type("True")
```

Out[66]:
```
str
```

# Strings

Strings are used in Python to record text information, such as names. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order.

In this lecture we'll learn about the following:

```
1.) Creating Strings
2.) Printing Strings
```

## Creating and Printing a String

To create a string in Python you need to use either single quotes or double quotes and use print function to display strings in your output

In [67]:
```
# Single word
b = "hello123"
```

In [68]:
```
#Print single word
print(b)
```

```
hello123
```

In [69]:
```
# Entire phrase
print('This is also a string')
```

```
This is also a string
```

In [19]:
```
# We can also use double quote
print("String built with double quotes")
```

```
String built with double quotes
```

In [20]:
```
# Be careful with quotes!
print(' I'm using single quotes, but this will create an error')
```

```
  File "<ipython-input-20-8a2657b45c3a>", line 2
    print(' I'm using single quotes, but this will create an error')
               ^
SyntaxError: invalid syntax
```

The reason for the error above is because the single quote in `I'm` stopped the string. You can use combinations of double and single quotes to get the complete statement.

In [70]:
```
print( "I'm using single quotes, but this will create an error")
```

```
I'm using single quotes, but this will create an error
```

In [71]:
```
print('Hello World 1')
print('Hello World 2')
print('Use \n to print a new line')
print('\n')
print('See what I mean?')
```

```
Hello World 1
Hello World 2
Use
 to print a new line


See what I mean?
```

## String Properties

It's important to note that strings have an important property known as *immutability*. This means that once a string is created, the elements within it can not be changed.

Something we *can* do is concatenate strings!

In [72]:
```
s='hello'
```

In [73]:
```
print(s)
```

```
hello
```

In [74]:
```
# Concatenate strings!
s + ' concatenate me!'
```

Out[74]:
```
'hello concatenate me!'
```

In [75]:
```
# We can reassign s completely though!
s = s + ' concatenate me!'
print(s)
```

```
hello concatenate me!
```

We can use the multiplication symbol to create repetition!

In [76]:
```
letter = 'z'
```

In [77]:
```
letter*10
```

Out[77]:
```
'zzzzzzzzzz'
```

In [78]:
```
 string = ''' ” inside a string I means “sky is blue”but sun is yellow " '''

print(string)
```

```
 ” inside a string I means “sky is blue”but sun is yellow "
```

### Task 2

Make a string having your name and print it 5 times.

In [80]:
```
# string assignment
name = "Albert"
```

In [81]:
```
# printing the string
print (name * 5)
```

```
AlbertAlbertAlbertAlbertAlbert
```

# Lists

Lists can be thought of the most general version of a *sequence* in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!

In this section we will learn about:

1.) Creating lists
2.) Indexing and Slicing Lists
3.) Nesting Lists

Lists are constructed with brackets [] and commas separating every element in the list.

Let's go ahead and see how we can construct lists!

In [82]:
```
# Assign a list to an variable named my_list
my_list=[1,2,3]
```

In [83]:
```
my_list
```

Out[83]:
```
[1, 2, 3]
```

We just created a list of integers, but lists can actually hold different object types. For example:

In [84]:
```
my_list = ['A string',23,100.232,'o']
```

In [85]:
```
my_list[0]
```

Out[85]:
```
'A string'
```

## Indexing and Slicing

We know Lists are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

In Python, we use brackets `[]` after an object to call its index. We should also note that indexing starts at 0 for Python.

We can use a `:` to perform *slicing* which grabs everything up to a designated point.

Let's create a new object called `my_list` and then walk through a few examples of indexing.. Let's make a new list to remind ourselves of how this works:

In [86]:
```
my_list = ['one','two','three',4,5]
```

In [87]:
```
# Grab element at index 0
my_list[0]
```

Out[87]:
```
'one'
```

In [88]:
```
# Grab index 1 and everything past it
my_list[1:]
```

Out[88]:
```
['two', 'three', 4, 5]
```

In [89]:
```
# Grab everything UP TO index 3
# indexes before the colons are inclusive but values after the colon are exculsive
my_list[:3]
```

Out[89]:
```
['one', 'two', 'three']
```

You can always access the indices in reverse. For example working according to the index, `my_list[0]` will be the first item and `my_list[-1]` will be the last one. Try the fowwlowing code.

In [90]:
```
# Grab the last index in reverse
my_list[-1]
```

Out[90]:
```
5
```

In [91]:
```
# Grab the second last index in reverse
my_list[-2]
```

Out[91]:
```
4
```

## Checking the type

In [92]:
```
type(my_list)
```

Out[92]:
```
list
```

In [93]:
```
# identifying type of specific object in list
type(my_list[1])
```

Out[93]:
```
str
```

## Task 3

Suppose we have a list containing areas of different rooms. Complete the given tasks using indexing and slicing.

In [94]:
```
# The list having areas of 6 rooms respectively
area=[28.3, 45.9, 123.4, 555, 213, 121]
```

In [95]:
```
# Show the area of third room in the list
area[2]
```

Out[95]:
```
123.4
```

In [96]:
```
# Show the areas of rooms first three rooms in the list
area[:3]
```

Out[96]:
```
[28.3, 45.9, 123.4]
```

In [97]:
```
# Show the area of rooms from 2 to 5
area[1:5]
```

Out[97]:
```
[45.9, 123.4, 555, 213]
```

We can also use '+' to concatenate lists.

In [98]:
```
my_new=my_list + ['new item']
```

In [99]:
```
my_new
```

Out[99]:
```
['one', 'two', 'three', 4, 5, 'new item']
```

Note: This doesn't actually change the original list!

In [100]:
```
my_list
```

Out[100]:
```
['one', 'two', 'three', 4, 5]
```

Note that lists are mutable objects i.e. a separate index can be changed through indexing

In [101]:
```
#mutable list objects can be changed
my_new[0]= 1
my_new
```

Out[101]:
```
[1, 'two', 'three', 4, 5, 'new item']
```

You would have to reassign the list to make the change permanent.

In [102]:
```
# Reassign
my_list = my_list + [1]
```

In [103]:
```
my_list
```

Out[103]:
```
['one', 'two', 'three', 4, 5, 1]
```

We can also use the * for a duplication method similar to strings:

In [104]:
```
my_list = my_list * 2
```

In [105]:
```
my_list*3
```

Out[105]:
```
['one',
 'two',
 'three',
 4,
 5,
 1,
 'one',
 'two',
 'three',
 4,
 5,
 1,
 'one',
 'two',
 'three',
 4,
 5,
 1,
 'one',
 'two',
 'three',
 4,
 5,
 1,
 'one',
 'two',
 'three',
 4,
 5,
 1,
 'one',
 'two',
 'three',
 4,
 5,
 1]
```

In [106]:
```
# Again doubling not permanent
my_list
```

Out[106]:
```
['one', 'two', 'three', 4, 5, 1, 'one', 'two', 'three', 4, 5, 1]
```

# Dictionaries

We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python. If you're familiar with other languages you can think of these Dictionaries as hash tables.

This section will serve as a brief introduction to dictionaries and consist of:

1.) Constructing a Dictionary
2.) Accessing objects from a dictionary
3.) Nesting Dictionaries

So what are mappings? Mappings are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

## Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

In [107]:
```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}
```

In [108]:
```
# Call values by their key
my_dict['key2']
```

Out[108]:
```
'value2'
```

Its important to note that dictionaries are very flexible in the data types they can hold. For example:

In [117]:
```
my_dict = {'key1':123,'key2':[12,"SDADAS",33],'key3':['item0','item1','item2']}
```

In [120]:
```
# Let's call items from the dictionary
my_dict['key1']
```

Out[120]: 123

In [121]:
```
# finding out the type
type(my_dict)
```

Out[121]: dict

**Task: Check type of 'key2'**

In [122]:
```
# Try here!
type(my_dict['key2'])
```

Out[122]: list

We can affect the values of a key as well. For instance:

In [123]:
```
my_dict['key1']
```

Out[123]: 123

In [124]:
```
# Subtract 123 from the value
my_dict['key1'] = my_dict['key1'] - 123
```

In [125]:
```
#Check
my_dict['key1']
```

Out[125]: 0

We can also create keys by assignment. For instance if we started off with an empty dictionary, we could continually add to it:

In [126]:
```
# Create a new dictionary
d = {}
```

In [127]:
```
# Create a new key through assignment
d['animal'] = ['Dog','Cat']
```

In [128]:
```
# Can do this with any object
d['answer'] = 42
```

In [129]:
```
d['answer'] = 21
```

In [130]:
```
#Show
print(d)
```

```
{'animal': ['Dog', 'Cat'], 'answer': 21}
```

## Nesting with Dictionaries

Hopefully you're starting to see how powerful Python is with its flexibility of nesting objects and calling methods on them. Let's see a dictionary nested inside a dictionary:

In [131]:
```
# Dictionary nested inside a dictionary nested inside a dictionary
d = {'key1':{'nestkey':{'subnestkey':'value'},'nestkey2':{'subnestkey':'value'}}}
```

In [132]:
```
# Keep calling the keys
print (d['key1'])
```

```
{'nestkey': {'subnestkey': 'value'}, 'nestkey2': {'subnestkey': 'value'}}
```

In [133]:
```
d
```

Out[133]:
```
{'key1': {'nestkey': {'subnestkey': 'value'},
  'nestkey2': {'subnestkey': 'value'}}}
```

In [134]:
```
# for getting the inner most value
d['key1']['nestkey']['subnestkey']
```

Out[134]:
```
'value'
```

## Dictionaries Exercise

In [135]:
```
# Definition of countries and capital
countries = ['spain', 'france', 'germany', 'norway']
capitals = ['madrid', 'paris', 'berlin', 'oslo']

# From string in countries and capitals, create dictionary europe
europe1 = {'spain': 'madrid'}
europe = {countries[0]: capitals[0], countries[1]: capitals[1], countries[2]: capitals[2], countries[3]: capi
tals[3]}

#print europe
print (europe.values())
```

```
dict_values(['madrid', 'paris', 'berlin', 'oslo'])
```

# Tuples

In Python tuples are very similar to lists, however, unlike lists they are *immutable* meaning they can not be changed. You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

In this section, we will get a brief overview of the following:

```
1.) Constructing Tuples
2.) Immutability
3.) When to Use Tuples
```

You'll have an intuition of how to use tuples based on what you've learned about lists. We can treat them very similarly with the major distinction being that tuples are immutable.

## Constructing Tuples

The construction of a tuples use () with elements separated by commas. For example:

In [136]:
```
# Create a tuple
t = (1,2,3)
```

In [137]:
```
# Can also mix object types
t = ('one',2)

# Show
t
```

Out[137]:
```
('one', 2)
```

In [138]:
```
# Use indexing just like we did in lists
t[0]
```

Out[138]:
```
'one'
```

In [34]:
```
# Slicing just like a list
t[-1]
```

Out[34]:
```
2
```

## Immutability

It can't be stressed enough that tuples are immutable. To drive that point home:

In [35]:
```
t[0]= 'change'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-1257c0aa9edd> in <module>
----> 1 t[0]= 'change'

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

Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it.

In [36]:
```
t.append('nope')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-36-b75f5b09ac19> in <module>
----> 1 t.append('nope')

AttributeError: 'tuple' object has no attribute 'append'
```

## When to use Tuples

You may be wondering, "Why bother using tuples when they have fewer available methods?" To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary. If in your program you are passing around an object and need to make sure it does not get changed, then a tuple becomes your solution. It provides a convenient source of data integrity.

You should now be able to create and use tuples in your programming as well as have an understanding of their immutability.

# Sets

Sets are an unordered collection of *unique* elements. We can construct them by using the set() function. Let's go ahead and make a set to see how it works

In [37]:
```
x = set()
print(x)
```

```
set()
```

In [38]:
```
type(x)
```

Out[38]:
```
set
```

In [39]:
```
# We add to sets with the add() method
x.add(1)
```

In [40]:
```
#Show
x
```

Out[40]:
```
{1}
```

Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

We know that a set has only unique entries. So what happens when we try to add something that is already in a set?

In [41]:
```
# Add a different element
x.add(2)
# show
x
```

Out[41]:
```
{1, 2}
```

In [42]:
```
# Try to add the same element
x.add(1)
# show
x
```

Out[42]:
```
{1, 2}
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements! We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

In [43]:
```
# Create a list with repeats
list1 = [1,1,2,2,3,4,5,6,1,1]
```

In [44]:
```
# Cast as set to get unique values
set(list1)
```

Out[44]:
```
{1, 2, 3, 4, 5, 6}
```

# Object Type Casting

You can cast type of any object in Python. Common data types casting functions include:

- **int()** (for integer)
- **float()**
- **str()** (for string)
- **bool()** (for Boolean True/False)

You can type cast any object throught the following code. The given example is converting **float** to **int**.

```
int(5.8)
```

## Examples

In [45]:
```
int(True)
```

Out[45]: 1

In [46]:
```
True + True
```

Out[46]: 2

In [47]:
```
bool(0)
```

Out[47]: False

In [48]:
```
# convert 80 into float type
float(80)
```

Out[48]: 80.0

In [49]:
```
# convert 20.9 into a string and check its type
str(20.9)
```

Out[49]: '20.9'

In [50]:
```
# convert a boolean valua into int
int(False)
```

Out[50]: 0

In [51]:
```
# convert '123' into float
float('123')
```

Out[51]: 123.0

## Pluralsight:

**You can access it for free in April (No credit card required)**

For beginners: https://app.pluralsight.com/library/courses/getting-started-python-core/table-of-contents (https://app.pluralsight.com/library/courses/getting-started-python-core/table-of-contents)

## Learn X in Y minutes

For quickly reviewing: https://learnxinyminutes.com/docs/python/ (https://learnxinyminutes.com/docs/python/)

## Practice

https://www.tutorialspoint.com/execute_python_online.php (https://www.tutorialspoint.com/execute_python_online.php)

In [ ]:

In [ ]: