Data Types

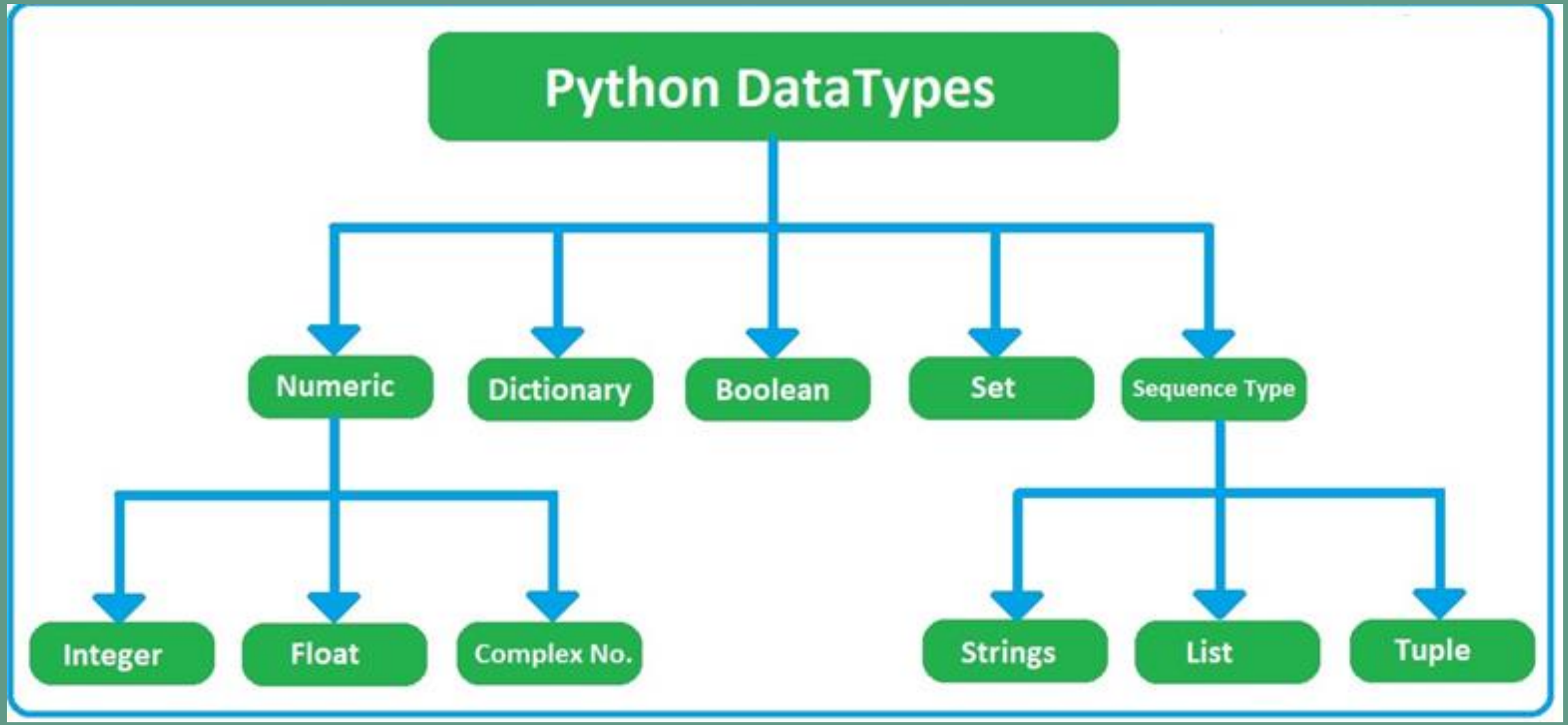
002_Python Evening Class

What is a Data Type

A Data Type describes the characteristic of a variable.

It represents the kind of value (data), that can be stored in a variable and tells what operations can be performed on a particular data.

It's important to point out that Python usually doesn't require you to specify what data type you are using and will assign a data type to your variable based on what it "*thinks*" you meant.


Python DataTypes
Numeric
Dictionary
Boolean
Set
Sequence Type
Integer
Float
Complex No.
Strings
List
Tuple

Assignment Operators

The assignment operator " = " (**Note** not to be confused with the " == " equals comparison operator), is used to store a value in memory.

As such, it can be recalled within the execution of the programme.

You can also use it along with mathematical operations to update or modify an existing value reference.

This could be done to increment or decrement a given value.

```
age = 21
print(age)     #-> 21

age +=1
print(age)     #-> 20

age -=1
print(age)     #-> 21

cost =100.00
print(cost)    #-> 100

cost *=.90
print(cost)    #-> 90.00
```

Naming conventions

The most typical naming convention within Python is referred to as snake case, snake case consist of all lowercase letters with any words separated by single underscores.

I would also strongly urge the use of meaningful names when declaring variables

Dynamically Typed

Python is a dynamically/Weak typed language, As opposed to a static/strongly typed language.

The Python's interpreter, does type checking only as the code runs.

As such the type of the variable is may to change over its lifetime. This could be done explicitly by the developer or implicitly by the Python interpreter.

```
var = "Hello World"
print(var, type(var))

var = 4
print(var, type(var))

var = 4/2
print(var, type(var))

var = 4//2
print(var, type(var))

var = 4
var += 0.1
print(var, type(var))
```

Type *(Built in function)*

Luckily, Python has a built-in function type, type takes one argument and returns the class name of the argument. As such, letting us know what the data type is.

This can be particularly handy when it comes to troubleshooting your code. As a can help clarify what type of object that is being processed.

```
age = 21
print(type(age))

print(type(1.1))

print(type([]))

print(type(" "))

print(type('1.2'))
```

"Fun" Fact: arguments vs parameters,

Although often use interchangeably arguments refers to the actual value that is being used and parameters refers to the placeholder defined in the functions/methods signature.

Immutable VS Mutable *(In Concept)*

Essentially, each different datatype object can be described to be mutable or immutable.

Mutable meaning the value stored at a particular memory location is changed.

Immutable means of value. Once created at a memory location cannot be changed.

> However, somewhat misleadingly variables can be reassigned to different locations in memory. Given the impression that the value has changed within memory.

For most operations. We don't need to think about this concept. However, when we come to looking at objects that point to the same location in memory, How Python behaves can be somewhat counterintuitive without discussing this concept

Authors note:
This is a little confusing, but will make sense over time, please bear with me and ask as many questions as you can think of.

id() *(Built in function)*

To get insight on how Python is handling each individual variable. We can use a built-in function known as id, id takes one argument and returns an integer value representing a memory location.

And as such we can use it to verify if two variables direct to the same location in memory.

```
x= 5
y= 5
print(id(x))
print(id(y))


#Note
x= [5]
y= [5]
print(id(x))
print(id(y))
```

Python the technique of having different variables, referencing the same location in memory in an effort to reduce memory consumption, by removing redundant values where possible.

As requested:

It is possible to retrieve the value by using the object id number. (Although I wouldn't recommend)

```
import ctypes
x= 5
print(ctypes.cast(id(x), ctypes.py_object).value)
```

is() *(Built in function)*

Building on this Python provides as a keyword called "is",

This is a comparison operator that checks to see if two variables/objects reference the same location in memory

This can also be achieved by comparing the two IDs returned.

```
x= [5]
y= [5]
print(x is y)
print(id(x)==id(y))
```

```
x= [5]
y= [5]
print(x is y)
print(id(x)==id(y))
```

Authors note:
Notice, How Python handles mutable and Immutable datatypes differently. You will see that the technique reduce memory consumption is bypassed with mutable objects

Numeric Data Types

Integers, floating point numbers and complex numbers fall under Python numbers category.

They are defined as **int**, **float** and **complex** classes in Python.

Data Types	Casting	Sample	Description	mutable
Integers	int()	17	Whole number negative or positive	**m**utable
Float	float()	2.8	Number negative or positive with decimal point	**m**utable
Complex	Complex()	3+5.6j	Number made of a <.Real()>+<.Imaginary()>j	**m**utable

Integer *(Int)*

We can use int data type to represent whole numbers (integral values). This means values like 0, 1, -2 and -15, and **not** numbers like 0.5, 1.01, -10.8, etc.

We can use the type() function to know which class a variable or a value belongs to. Integers can be of any length; it is only limited by the memory available.

If you give Python the following code, it will see that x is an integer and will assign the int data type to it.

```
x= 5
print(type(x))
```

We could have been more specific and said something along these lines, to make sure Python understood our 5 as an integer, though, it'll automatically does this exact same thing under the hood.

```
x= int(5)
print(type(x))
```

Ways to represent Integer values

Within maths, their different ways of representing the same quantity, just like in everyday speech, we might say six eggs or we might say half a dozen, But we would agree on the quantity of eggs required.

In computing, sometimes we want to represent numbers in decimal, as we do in a normal day-to-day lives. But other times you might want to represent them in binary or hexadecimal.

Python allows us to specify the particular number system were using to represent the quantity in the value.

```
x= 5
print(type(x))

x= 0B101
print(type(x))

x= 0O10
print(type(x))

x= 0XA
print(type(x))
```

Authors note:
Don't panic, it really isn't that big component. If it's not completely clear (It's okay).

System	Base	Single Digits
Binary	2	0,1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Float

The float type in Python represent real numbers and are written with a decimal point splitting the integer and fractional parts.

Floats may be also represented in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).

> **Note**: We can represent int values in decimal, binary, octal and hexadecimal forms. But we **can** represent float values only by using decimal form.

We can also use underscore "_" to split up longer number in thousands, Python ignores these underscores "_"

```
x= 5.2
print(x, type(x))

x= 4e-4
print(x, type(x))

x= 4e4
print(x, type(x))

x= 400_000_000
print(x, type(x))
```

```
x = 1
print(x, type(x))

y= x+1.1
print(y, type(y))
```

With the nature of Python been weakly typed, it will automatically change variable types to preserve the most precise accuracy.

So, for example, if I was to add an integer and they float together. It would convert the integer to float, so it can preserve the fractional value on the float variable

Float

However, as previously mentioned, if we only say 5, Python will consider it an int data type.

If, for some reason, we wanted a float variable that has the value 5, we'd need to explicitly let Python know

```
y = 5.0
print(y, type(y))

y= float(5)
print(y, type(y))
```

```
print(type(float('infinity')))
print(type(float('-infinity')))
print(type(float('nan')))
```

Float, data type can be used to represent some special "numbers" like the NaN ("Not a Number") and +/-infinity

The maximum value a floating-point number can have is approximately 1.79 × 10308. Python will indicate a number greater than that by the string **inf**

```
x= 1.79e308
print(x, type(x))

x*=2
print(x, type(x))
```

Complex Numbers

Complex numbers allow for greater precision on very large datasets, they may be used for a strong me physics and many other advance calculations. Although outside of these fields.

There made out of two parts ; a real and the imaginary part

Complex numbers are set out as :

<real part>+<imaginary part>. J

Both real and imaginary part, can be stated in int &/or float datatype In the real parts;

if we use int value for the real part, then we can specify that either by decimal, octal, binary or hexadecimal form.

But the imaginary part should be specified only by using decimal form.

```
x= 1+3j
print(x, type(x))


# Methods()

print(x.real)
print(x.imag)
print(x.conjugate())

x= 0B101+3j
print(x, type(x))
```

Boolean *(Not-Numeric)*

You can think of boolean type as your light switch. It's always either on or off. In Pythofirst,it's either True or False.

We can use this data type to represent Boolean values, The only allowed values for this data type are, True and False;

You'll see how price and will implicitly convert values to Boolean in certain cases. This can be a little off putting at first, but it can be useful in a few different circumstances.

Note:
That, boolean type **T**rue or **F**alse is never in quotes, which separates it from strings. So in short, True and "True" are two different data types in Python, the latter is a str type while former is a bool.

Mathematical operators applied to Boolean

This can seem quite strange at first but there are times when you may want to do something like adding Boolean values, perhaps counting flags raised.

As internally Python stores, Booleans as 1 or 0, it will be implicitly converted into an integer value when used in a mathematical equation.

That is the say True +True is the same as 1+1, and False +False is 0+0

```
print(True + 1)
print(True + True)
print(True + True + False)

print(True * False)
print((1>0)+1)

print((1>0)+(3<2))

print(1>0)
```

Boolean Evaluation

When evaluating datatypes as Boolean:

Any nonzero integer value will be seen as true.

Any nonzero float type value will be seen as true.

Any complex value will be seen as true if either the real or imaginary side are nonzero.

Any collection (string, list, set, dictionary) whose length is equal or greater to one will be seen as true.

Authors note:
There are sometimes where this can be quite useful when iterating over a list . For example.

```
print(bool(0))
print(bool(1))
print(bool(9))
print(bool("False"))
print(bool([1]))
print(bool((0)))

print(bool((0+0j)))
print(bool((0+1j)))
print(bool((1+1j)))

print(1<2)
print(1>2)
```

Sequence Type

Sequences are quite useful in Python, when we think about how the language is quite popular areas requiring large datasets (AI, for example) A focus on sequences is not too surprising.

Examples of sequences are **Tuple**, **Lists** & **Strings**.

The key things to note about sequences will be the mutability & their mixed datatypes & the ability to "nest" sequences.

But as you get hands-on experience. It will all become more intuitive.

Data Types	Casting	Sample	Description	mutable
Strings	str()	"name"	A Iterable of characters, order is preserved	**i**mmutable
Lists	list()	[‘a’,1,1.2]	A Iterable of values, order is preserved	**m**utable
Tuple	tuple()	3+5.6j	A Iterable of values, order is preserved	**i**mmutable

Tuple

Tuples in Python are quite common, there immutable, which means in practice. If we want to change the values in a Tuple,

The simplest way is often to create a new Tuples , (with a new id value).

The Tuples are declared usually within round brackets "( )", with each value separated by a "," however, the round brackets are not necessary if the "," is present

```
ages = (1,2,3)
print(ages)

ages = 10,20,30
print(ages)


ages = (10)
print(ages)

ages = 10,
print(ages)
```

```
my_tuple= (1,[1],'A')
print(my_tuple)

my_tuple[1].append(2)
print(my_tuple)
```

Warning:
although Tuples are immutable, they can hold mutable values within them.

Tuple Indexing

Tuples, preserves insertion order, that is the say the order in which the values are passed into the data structure is the order in which they are accessible (retrieve).

The first value passed in will have an index of zero, then the next one, and so on.

To access the value we use the variable name followed by square brackets [] containing the index of the value we want returned.

Python supports negative indexing; *(this is very helpful)*

What is meant by negative indexing is the very last element can be accessed by using the index of -1, the second last element -2, and so on.

```
ages = (1,2,3)
print(ages[1])

ages = (1,2,3)
print(ages[0])

ages = (1,2,3)
print(ages[-1])

ages = (1,2,3)
print(ages[-2])
```

```
#Why is this erroneous?
ages = (1,2,3)
ages[1]= 9
```

Tuple Slicing

Slicing allows us to retrieve a subset of the Sequence.

This is done by providing the starting index and the ending index,this will return every value up to in this range (**Exclusively**).

These values will be separated by a colon ":"

```
ages = (1,2,3,4,5)
print(ages[1:4])

ages = (1,2,3,4,5)
print(ages[:])

ages = (1,2,3,4,5)
print(ages[::-1])
```

```
Sequence[starting index: ending index: step value]
```

We have the option of including what is referred to as the step Value.
The step allows us to retrieve values in greater intervals, or descending intervals.

I.e. retrieving every second value in the provided range,or retrieving every value in the range from greatest to smallest index.

Note if the starting value is omitted, the very beginning index is assumed, if the ending value is omitted, the very last index is assumed, and if the step value is omitted +1 is assumed.

Tuple Methods *(returning)*

The Tuple object, has quite few methods. If we consider how the Tuple is immutable. This seems intuitive. As most methods alter the values within the object.

Therefore, we have only really too interesting methods to use with the Tuple

Authors note

Functions() vs .Methods()	
Functions and methods very simmer similar, both can Returns values/objects, both can potential accept arguments/args, both when called end with open closed Parentheses(). ***Where they differ is how we call them.***	
Functions()	**.Methods()**
Name of the arg functions(arg)	Obj.mehtod() my list.appent(args)
Note: we can change *Method calls like so* Obj.mehtod().mehtod().mehtod()	

```
ages = (1,2,3,4,5)


print(ages.count(2))
print(ages.index(2))
```

Methods	Returns ("Getters")
.count(e)	Returns the number of times a specified **e**lement occurs in a tuple
.index(e)	Searches the tuple for a specified **e**lement and returns the position of where it was found

Lists

Lists are quite similar to Tuple, the key differences that they are mutable, therefore, not only can we access values, but we can change them.

Listen defined by square brackets "[]" containing values separated by commas.

Order insertion is preserved, they are indexable, and can be sliced

```
ages = [1,2,3]
print(ages)

ages = [10,20,30]
print(ages)
```

List Indexing

Lists indexing begins at zero, and supports negative indexing.

Lists being mutable allows us to change value stored within them

Positive index	0	1	2	3	4	5
value	A	B	C	D	E	F
Negative index	-6	-5	-4	-3	-2	-1

C would have a positive index of 2 and on the negative index of -4 in the above example.

```
ages = [1,2,3]
print(ages[1])

ages = [1,2,3]
print(ages[0])

ages = [1,2,3]
print(ages[-1])

ages[1]= 9
print(ages)
```

```
#Why is this erroneous?
ages = [1,2,3]
ages[1]= 9
```

List Slicing

Slicing Lists takes the same format as we seen before

```
Sequence[starting index: ending index: step value]
```

The Rangers also **Exclusively** Up to but not including the ending index.

Positive index	0	1	2	3	4	5
value	A	B	C	D	E	F
Negative index	-6	-5	-4	-3	-2	-1

```
ages = [1,2,3,4,5]
print(ages[1:4])

ages = [1,2,3,4,5]
print(ages[:])

ages = [1,2,3,4,5]
print(ages[::-1])
```

```
letter= ['A','B','C','D','E','F']
print(letter[2:4])
```

```
letter= ['A','B','C','D','E','F']
print(letter[-4:-2:])
```

Lists Methods *(return)*

List has methods that do not modify the values within the data.

One of the more significant, methods of this type is copy, as it is required to make a "deep" copy of the data type, as opposed to a reference or a "shallow" of the data type. A reference to an object will share the same identity.

```
ages = [1,2,3,4,5]

print(ages.count(2))
print(ages.index(2))

ages_2 = ages.copy()
print(ages_2)
```

Methods	Returns (“Getters”)
.count(e)	Returns the number of **e**lements with the specified value
.index(e)	Returns the index of the first **e**lement with the specified value
.copy()	Returns a copy of the list

Deep copies versus shallow copies

If we execute the script we might notice something interesting happened,

we may modify letter_1 by appending G to the list, it also changes the original letter list.

Authors note:
Hopefully this alludes to the importance of understanding mutability and the id function.

```
letter= ['A','B','C','D','E','F']

letter_1= letter

print(id(letter))
print(id(letter_1))
print(letter is letter_1)

letter_1.append("G")

print(letter)
```

Lists Methods

These methods will change the existing list.

```
ages = [1,2,3,4,5]

ages.append(6)
print(ages)

ages.clear()
print(ages)

ages = [1,2,3,4,5]
ages.extend([7,8,9,])
print(ages)
```

Methods	"Setters"
`.append(e)`	Adds an **e**lement at the end of the list
`.clear()`	Removes all the elements from the list
`.extend(obj)`	Add the elements of obj (or any iterable), to the end of the current list.
`.insert(i,e)`	Adds an element at the specified index

Lists Methods

These methods will change the existing list.

```
ages = [1,2,3,4,5]
ages.insert(0,9)
print(ages)

ages.pop(0)
print(ages)

ages.remove(5)
print(ages)

ages.reverse()
print(ages)

ages.sort()
print(ages)
```

Methods	"Getters" & "Setter"
.pop(**i**)	remove fist instances of give **e**lements

Methods	Affect ("Setters")
`.pop(i)`	Removes the element at **i**ndex
`.remove(e)`	Removes the **e**lements
`.reverse()`	Reverses the order of the list
`.sort()`	Sorts the list in ascending order

Strings

Strings are sequences of character data. The string type in Python is called str.

String literals may be Outline using either single or double quotes.

Multi-line strings can be represented using triple quotes, ''' or """.

All the characters between the opening quote/s and matching closing quote/s are part of the string.

Strings

Strings can be considered as a special type of sequence, where all its elements are characters.
For example,

string "Hello World" is basically a list sequence ['H', 'e', 'l', 'l', 'o',' ', 'W', 'o', 'r', 'l', 'd']

You will see a lot of similarities with lists, such as indexing and slicing much the same practices can be applied to strings. However, you will see some extra methods found in strings. Such as methods explicitly related to handling texts

Authors note:
In python there is no character (chr) data type, a character is a string of length one

String formatting (Introduction)

There really are a few different ways of formatting strings , but for now, I would just like to introduce you to concatenation and using the F-String .

This is mostly because it is required to make programmes a little bit more engaging.

```
name="James"
age= 21

print("Hello " + name + "I hope you like being " + str(age))
print("Hello ",  name ,"I hope you like being ",str(age))
print(f"Hello {name} I hope you like being {age}")
```

Strings Escape Sequences & Quotations

The\character is referred to as the escape character. This is the "\" escapes the normal functionality of the next preceding character (letter).

This is used to insert invisible or special characters, and example of this could be "a new line" or carriage return , Another example could be tab

Escape sequence	Output
\\	\
\n	New line
\"	"
\'	'
\t	Tab

```
print("hello,\nHow are you")
```

```
print('I "like" Python')
```

You can avoid the need for escape characters by using alternate quotation marks, Python will accept both single quotes and double quotes the start and close the string.

This means if you wish to use quotation marks within your string. You can open them with single quotes and as such double quotes will be seen as characters within your string

RAW Strings

Another way we can avoid the use of escape characters is with R strings.

R strings are really helpful , particularly when it comes the file path. They simply bypass any escape characters

for example , without the use of force strings each \would need to be doubled up like so

```
print('C:home\\files\\names.txt')
```

But with the use of r string

```
print(r'C:home\files\names.txt')
```

Multi-line strings

Python Also makes it possible to remove the need for carriage returns\ new lines, with the use of multi-line strings.

Multi-line strings preserve all indentation online breaks found within the text, they can be F strings & R strings. They can be started with either single or double quotes

```
message= """
Hello,
I do hope you well, remember the quote

    "stitch in time"

All the best
James Barrett
"""
print(message)
```

String Methods

Methods	"Getters"
`.split()`	**.split(string argument):** Returns a list to the parts of a string split on the string argument give. If no string argument it will split on a white space " "
`.find()`	**.find(string argument):** Returns the index of the string argument given, if the string argument is not found it Returns -1
`.index()`	**.index(string argument):** Returns the index of the string argument given, if the string argument is not found it Returns "ValueError"
`.replace()`	**.replace(old_string_argument, new_string_argument):** Returns an string with all occasion of the old_string_argument replaced with the new_string_argument.
`.strip()`	**.strip():**Returns an string with all preceding & trailing white spaces removed
`.rstrip()`	**.strip():**Returns an string with all trailing white spaces removed
`.lstrip()`	**.strip():**Returns an string with all preceding white spaces removed

Authors note:
Remember, strings are immutable. Therefore, all these methods return new objects

Dictionary *(Not-Sequence Type)*

Dictionaries are a set of key pair values, they can be seen as a lookup table mapping a key to a given value.

Dictionaries are mutable, it can be a bit technical, when we consider if dictionaries preserve insertion order, they are not technically indexable in the same way strings tuple and lists are; however, practically their keys can serve as indexes for all intensive purposes.

Dictionaries are a set of keys Values separated by a colon, which in turn are then separated by commas and all encapsulated within curly braces

```
phonebook= {
     "James":"085 788 234",
     "Tommy":"087 285 094",
     }

print(phonebook["James"])
print(phonebook["Tommy"])
print(phonebook)
```

Authors note:
Remember, keys **must** always be unique; values, however, can repeat

Dictionary Access By Key (Not- Indexing)

Although slightly different from indexing, we can retrieve values and change values within the dictionary, in a very similar way that we do to list, we use square brackets and place the key inside the brackets.

One significant difference is the fact that the key can be any immutable datatype, not just in integer

```
phonebook= {
     "James":"085 788 234",
     "Tommy":"087 285 094",
     }

print(phonebook["James"])
phonebook["James"]="062 346 2367"
print(phonebook["James"])
```

Dictionary Methods

Methods	"Getters"
`.get(key)`	Returns a value for a given key
`.keys()`	Returns all dict_keys for a Dictionary
`.values()`	Returns all dict_ values for a Dictionary
`.items()`	Returns all dict_items for a Dictionary
`.copy()`	Remove **e**lements from Dictionary

```
phonebook= {"James":"085 788 234","Tommy":"087 285 094"}


print(phonebook.get("James"))
print(phonebook.keys())
print(phonebook.values())
print(phonebook.items())
```

Dictionary Methods

Methods	"Setters"
`.update({K:V})`	Adds new item/s from {key:Values}
`.popitem()`	Returns a tuple containing the {key:values}
`.clear()`	Remove all item in Dictionary
`.fromkeys(obj,obj)`	Returns a dictionary with the specified keys and the specified value
`.setdefault(K,obj)`	method returns the value of the item with the specified key.If the key does not exist, insert the key, with the specified value,

```
dict ={'K':'V'}
del dict['K']
print(dict)
```

```
dict ={'K':'V'}
del dict
print(dict)
```

Sets & Frozen Set *(Not-Sequence Type)*

Sets of similar to other collection datatypes, We looked at.

The key differences do not preserve insertion order therefore they are not indexable or sliceable and they disregard duplicate values.

Sets come in two variations set and frozen sets
Sets are mutable and frozen immutable

Sets

Sets are defined by a set of curly braces with values separated by commas.

However, to create an empty set, you have to use the set constructor, otherwise you create an empty dictionary.

```
primes={2, 3, 5, 7,2}
print(primes)

not_set={}
is_set=set({})

print(type(not_set))
print(type(is_set))
```

Sets Methods

Methods		"Setter"
.add()		adds new_**e**lement to Set
.clear()		remove all **e**lements in Set
.remove(**e**)		remove fist instances of give **e**lements Will not throw an error if **e**lement is not found
.discard(**e**)		remove fist instances of give **e**lements Will not throw an error if **e**lement is not found
.update(e)	|=	updates a sets adding item in **e**

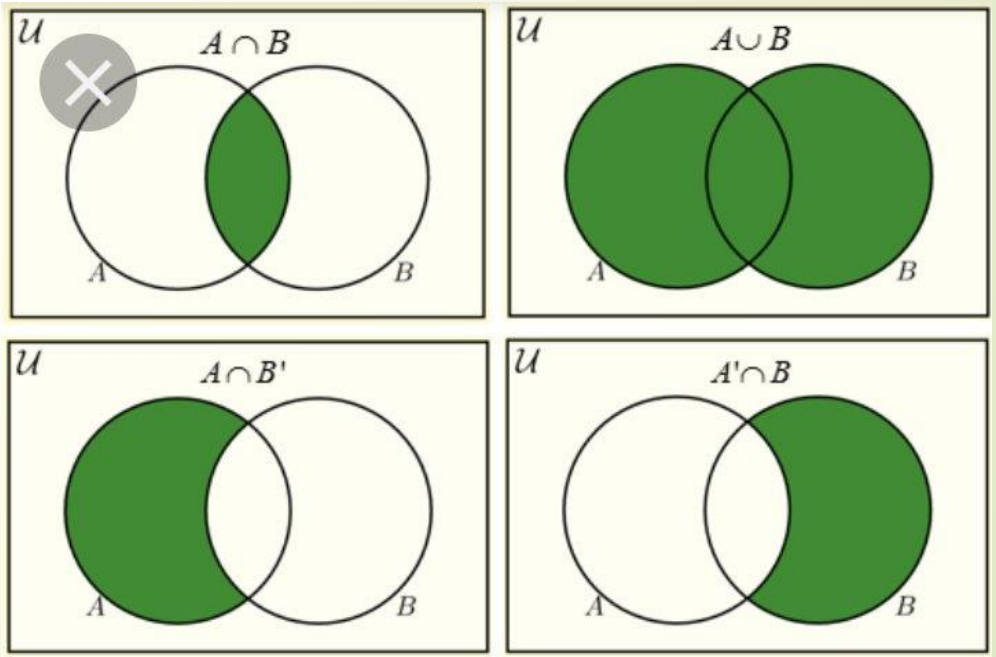
Methods	"Getters" & "Setter"
.pop(**i**)	remove fist instances of give **e**lements Will not throw an error if **e**lement is not found

Sets Methods

Methods	OP	"Getters"
union()	|	Returns a set containing all item in boat sets
intersection()	&	Returns set a containing all shared item boat sets.
symmetric_difference()	^	Returns set containing all **not** shared boat.
difference()	-	Returns set a containing all item a but not in **b** set.

a key reason to used sets is how they can be manipulated, a Venn diagrams one way of illustrating in this.

As this functionality can be used to quickly sort through data values



Frozen Set

Frozen sets are just like sets in every way except the fact they are immutable.

```
f_set=frozenset({})
f_set=frozenset({1,2,3,1})
print(f_set)

print(type(f_set))
```

```
#what will happen

f_set=frozenset({})
f_set.add(1)
```

Methods	OP	"Getters"
union()	|	Returns a set containing all item in boat sets
intersection()	&	Returns set a containing all shared item boat sets.
symmetric_difference()	^	Returns set containing all **not** shared boat.
difference()	-	Returns set a containing all item a but not in **b** set.

None Object Data Type

The None Object

Note that the `PyTypeObject` for `None` is not directly exposed in the Python/C API. Since `None` is a singleton, testing for object identity (using `==` in C) is sufficient. There is no `PyNone_Check()` function for the same reason.

`PyObject` ***Py_None**

The Python `None` object, denoting lack of value. This object has no methods and is immortal.

> *Changed in version 3.12:* `Py_None` is immortal.

Py_RETURN_NONE

Return `Py_None` from a function.

Note:
The None keyword is used to define a null value, or no value at all.

None is not the same as 0, False, or an empty string. None is a data type of its own (NoneType) and only None can be None.

https://docs.python.org/3/c-api/none.html

Explicit Type Casting

The process of changing one datatype from another is referred to as casting. An integer can be changed into a float quite easily.

But strings have the potential of also being turned into a number value, of course, this only works when it makes sense.

```
my_number=1
print(type(my_number))

my_number= str(my_number)
print(type(my_number))

my_number= float(my_number)
print(type(my_number))

my_list=[1,2,4]
print(type(my_list))
```

```
#Why is this erroneous?
number= "Two"
my_list= int(number)
print(type(my_list))
```

```
my_list= set(my_list)
print(type(my_list))

number= "1"
my_list= int(number)
print(type(my_list))
```

Implicit Type Casting

Python will when necessary and appropriate, automatically change datatypes within calculations.

A good rule of thumb is to think about preserving as much data as possible

```
number= 8
ans= number * 2
print(type(ans))

ans= number / 2
print(type(ans))

ans= number // 2
print(type(ans))
```

```
number= 8
ans= number * 2
print(type(ans))

ans= number / 2
print(type(ans))

ans= number // 2
print(type(ans))
```

Comparison Operators

Operators	Description
<	Returns True if the value on the left is smaller the value on the right;else False.
>	Returns True if the value on the left is bigger the value on the right; else False.
==	Returns True if the value on the left is equal the value on the right; else False.
>=	Returns True if the value on the left is Greater than or Equal the value on the right, otherwise Returns False.
<=	Returns True if the value on the left is less than or Equal the value on the right, otherwise Returns False.
!=	Returns True if the value on the left is Not equal the value on the right, otherwise Returns False.

```
print(3==3)
print(3!=3)
```

```
print(3==3)
print(3!=3)
```

```
print(3>=1)
print(1<=3)
```

```
#What about ?
print("James"== "James")
print("James"== "The Best")
print("James"== "james")
```

```
#What about ?
print("James" > "James")
print("James" > "The Best")
print("James" < "james")
```

```
print(1.0==1)
print(1.0==1.000)
```

Comparison Strings

Comparing strings is quite an odd, but it is possible to do so in Python.

How Python calculates this is by converting each letter of the string into its point code value. Then the comparison operation is done on each individual point code value until the comparison can be resolved.

String_1	H	e	l	l	o
Letter point code	72	101	108	108	111
	=	=	<		
	=	=	>		
Letter point code	72	101	121		
String_2	H	e	y		

We can use the built-in functions ord() to return the point code value of a given character.

```
print('A'<'B')
print(ord("A"))
print(ord("B"))
print(65<66)
```

We can use the char() built-in functions to return the character of a point code value

```
print(chr(65))
```

Point Code of characters

Character	Code	Character	Code	Character	Code	Character	Code
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111
(DLE)	16	0	48	P	80	p	112
(DC1)	17	1	49	Q	81	q	113
(DC2)	18	2	50	R	82	r	114
(DC3)	19	3	51	S	83	s	115
(DC4)	20	4	52	T	84	t	116
(NAK)	21	5	53	U	85	u	117
(SYN)	22	6	54	V	86	v	118
(ETB)	23	7	55	W	87	w	119
(CAN)	24	8	56	X	88	x	120
(EM)	25	9	57	Y	89	y	121
(SUB)	26	:	58	Z	90	z	122
(ESC)	27	;	59	[	91	{	123
(FS)	28	<	60	\	92	|	124
(GS)	29	=	61	]	93	}	125
(RS)	30	>	62	^	94	~	126
(US)	31	?	63	_	95		127

Containment *(Built in function)*

This in keyword can be used to evaluate if an element is present in a collection.

It will return a Boolean value of true if it is otherwise a value false.

```
name= "James"
print('J' in name)

names= ["James","Joy"]
print("James" in names)

names= ("James","Joy")
print("James" in names)

names= {"James","Joy"}
print("James" in names)
```

Arithmetic Operators

Python can perform all standard mathematical operations on number values,

And, quite strangely , we can also use it to multiply collections.

Operators	Description
+	Add
-	Subtract
/	Division always retunes the Float
//	Floor division always retunes the int
*	Multiple
**	a//b : a to the power b
%	Modulo The remainder of a division

```
#What about?

name= "Hello "
print(name*3)
```

Order Of Operations

Please pardon my Dear and Sally

```
print(2*1+1)
```