



# *Python Programming*

## Module 1: Python Fundamentals

### *Lesson 2: Python Data Structures*

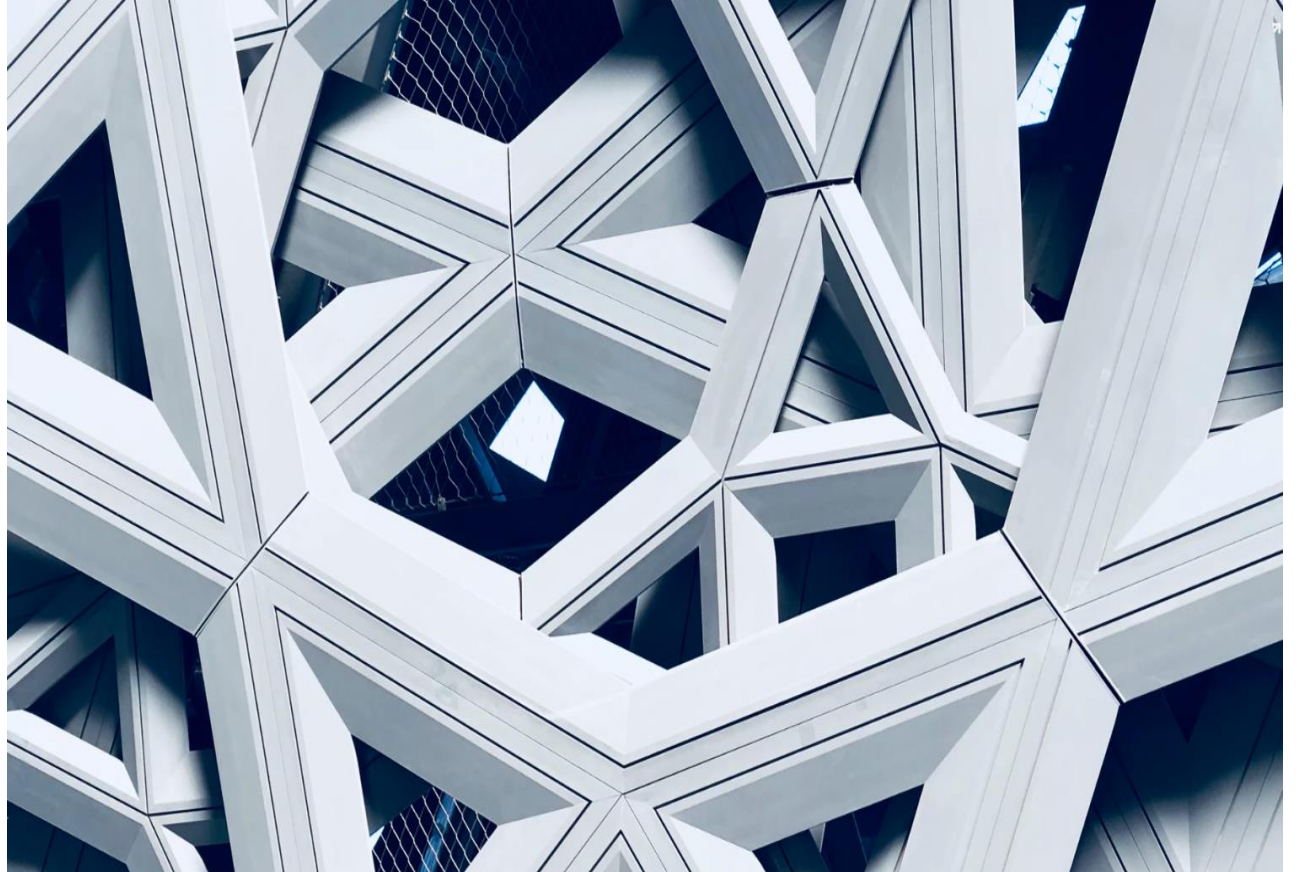
Neba Nfonsang  
University of Denver

# Lesson 2: Python Data Structures

*“Python’s data structures are simple but powerful.*

*Mastering their use is a critical part of becoming a proficient Python programmer” –*

Wes McKinney



# Lesson 2: Python Data Structures

- Intro to Python Object Types
- Numeric Types
  - Integers
  - Floats
  - Decimal
  - Fraction
- Strings
- Booleans
- Lists
- Tuples
- Sets
- Dictionaries
- Booleans or bools




# Intro to Python Object Types

- A Python program basically consist of statements that process some data. These data values take the form of objects of specific types.
- Objects are also known as data structures.
- Data structures built into Python are built-in types
- Examples of some basic Python built-in types include numeric, string, Boolean, list, tuple, set, and dictionary.
- Other built-in types are functions, modules and classes – which will be discussed in subsequent lessons.

# Intro to Python Object Types

- In Python, there is room for programmers to build their own objects or data structures using classes or external libraries.
- However, built-in types are more powerful and efficient than custom data structures.
- Let's begin to explore built-in object types in Python.





# Generic and Type-Specific Operations

- Object types in Python have generic and type-specific operations.
- Generic operations span across different types but type-specific operations are unique to a particular object type.
- Generic operations are executed through functions or expressions while type-specific operations are executed through methods.
- Methods are functions that belong a specific object type.

# Generic and Type-Specific Operations

- A function such as **len()** perform a generic operation as it can be used to find the number of items in a lists, sets, tuples etc.
- Note that an expression such as `var_name[0]` also performs generic operations on sequences.
- A function such as `.upper()` performs a type-specific operation because it works only with strings (used to create upper case letters).
- Moreover, an object's type can be checked using **type(object)**





# Attributes and Methods of an Object

- An object has attributes and methods.
- Attributes describe the properties of an object.
- Methods are functions belonging to objects. Methods carry out operations on objects.
- Call the **dir(object)** function on an object to see what methods and attribute belong to that object.
- Similarly, to explore the features of an unfamiliar package, import the module and call `dir()` on it or call `dir()` on objects created with classes from the module.



# Numeric Types

- Some basic numeric types in Python are integers, floats, decimals and fractions.
- **Integers(int)** are whole numbers. E.g. 1, 2, 3, and 4
- **Floating-points (float)** are numbers with decimal or fractional parts. E.g. 1.5 and 2.0
- **Decimals** are floating-points with a fixed number of decimal points.
- **Fractions** are rational numbers with a numerator and denominator.
- An object's type can be checked using the **type()** function.

# Numeric Types and Operations

- Numeric types are supported by mathematical operations.
- Here are some arithmetic operators that can be used with numeric types.
- The order of precedence should be applied if an expression has mixed operators.
- Parenthesis could be used to change the order of precedence.

Operators by Precedence	Description
**	Exponentiation
-	Negation
*	Multiplication
/	Division
%	Remainder
+	Addition
-	Subtraction

# Arithmetic Operations with Numbers

1	<i># addition</i>
2	<code>2 + 5</code>
7	
1	<i># subtraction</i>
2	<code>10 - 6</code>
4	
1	<i># mixed operators</i>
2	<i># follow order of precedence</i>
3	<code>3 + 10*2</code>
23	
1	<i># change precedence with</i>
2	<i># parantheses</i>
3	<code>(3 + 10)*2</code>
26	

1	<i># raise to the 2nd power</i>
2	<code>4**2</code>
16	
1	<i># division (/) always</i>
2	<i># returns a float</i>
3	<code>4/2</code>
2.0	
1	<code>5/3</code>
1.6666666666666667	
1	<i># floor division</i>
2	<i># discards the fractional part</i>
3	<code>5//3</code>
1	
1	<i># remainder</i>
2	<code>10%3</code>
1	

# Fractions and Decimals

- Some floats are less precise. In situation where precision matters, it might be preferable to use fractions and decimals instead of floats.
- $1/3$  is more accurate than  $0.33333333333333...$

```
1 # fractions
2 from fractions import Fraction
3 Fraction(4, 2)
```

Fraction(2, 1)

```
1 # Fraction(numerator, denominator)
2 Fraction(1, 3)
```

Fraction(1, 3)

```
1 Fraction(1, 3) + 5
```

Fraction(16, 3)

# Fractions and Decimals

- Now, let's examine the decimal type.
- From the calculations, you can see that the decimal type is more precise compared to floats.

```
1 # Let's add two floats
2 1.3 + 1.303
```

2.6029999999999998

```
1 # Let's use Decimal()
2 # the answer is more precise
3 Decimal("1.3") + Decimal("1.303")
```

Decimal('2.603')

# Strings

- Strings are useful in recording textual information.
- A string is a sequence of characters.
- Characters consist of letters, numbers and symbols.
- In Python, a string value is surrounded by quotes.

```
1 # Strings
2
3 "This is a string"
```

'This is a string'

```
1 number = "30"
2 type(number)
```

str

```
1 number
```

'30'

```
1 greetings = "Hi, there"
2 type(greetings)
```

str

```
1 print(greetings)
```

Hi, there

# Strings

- Just like a list and a tuple, a string is a sequence because it is a positionally ordered collection of other objects (characters).
- Therefore, generic sequence operations such as **len()** and slicing/indexing would work for a string.
- Generally, indexing involves extracting a single item while slicing extracts more than a single item

```
1 # Length of string
2 name = "Johnson"
3 len(name)
```

7



# String Indexing/Slicing

- The syntax for indexing is **var\_name[index]** where index is the position of the character you want to extract or return.
- Note that Python starts counting from zero. That means the first character is at the zeroth index.
- The syntax for slicing is **var\_name[start:end]**. This implies, slice from the specified start position up to but not including the end position.

The diagram illustrates string indexing for the string "Johnson". It consists of a 3x7 grid of characters and their corresponding indices. Arrows point from labels to the first column of the grid.

String	J	o	h	n	s	o	n
Index position	0	1	2	3	4	5	6
Negative index	-7	-6	-5	-4	-3	-2	-1

# String Indexing/Slicing

```
1 # slicing a string
2 # return all items
3 name[:]
```

'Johnson'

```
1 # index the first character
2 name[0]
```

'J'

```
1 # index the third character
2 # note that Python starts
3 # counting from zero
4 name[2]
```

'h'

```
1 # negative indexing
2 # return the last character
3 name[-1]
```

'n'

J	o	h	n	s	o	n
0	1	2	3	4	5	6
-7	-6	-5	-4	-3	-2	-1

```
1 # return the second to the last
2 # character
3 name[-2]
```

'o'

```
1 # slice from the 3rd to 5th
2 # character
3 name[2:5]
```

'hns'

```
1 # slicing with negative indexes
2 name[-3:]
```

'son'

# String Methods

- Again, you can call the `dir()` function on a string object to see what methods are available for string processing.
- For now, we will not focus on names with double underscores as these are usually used for implementation

```
1 print(dir(name))
```

```
['__add__', '__class__', '__contains__', '__delattr__',  
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',  
 '__getattr__', '__getitem__', '__getnewargs__', '__i  
gt__', '__hash__', '__init__', '__init_subclass__', '__i  
ter__', '__le__', '__len__', '__lt__', '__mod__', '__mul  
__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',  
 '__repr__', '__rmod__', '__rmul__', '__setattr__', '__si  
zeof__', '__str__', '__subclasshook__', 'capitalize', 'c  
asefold', 'center', 'count', 'encode', 'endswith', 'expa  
ndtabs', 'find', 'format', 'format_map', 'index', 'isaln  
um', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',  
 'islower', 'isnumeric', 'isprintable', 'isspace', 'istit  
le', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'mak  
etrans', 'partition', 'replace', 'rfind', 'rindex', 'rju  
st', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitli  
nes', 'startswith', 'strip', 'swapcase', 'title', 'trans  
late', 'upper', 'zfill']
```

# String concatenation

- Note that the + operator is used to concatenate or combine strings.
- Note: Strings are immutable. That means, you cannot modify a string, shorten or increase its length. You cannot replace the characters.
- However, a new string can be assigned to an existing string variable name. Concatenation creates a new string

```
1 # string concatnation
2 first_name = "Elizabeth"
3 last_name = " Mckenzie"
4 full_name = first_name + last_name
5 full_name
```

'Elizabeth Mckenzie'

```
1 "Hello" + " " + full_name + "!"
```

'Hello Elizabeth Mckenzie!'

# Change Cases in Strings

```
1 # change string to title case
2 first_name = input("Please, enter your first name: ")
3 last_name = input("Enter last name: ")
4 name = first_name + " " + last_name
5 full_name = name.title()
```

Please, enter your first name: naomi  
Enter last name: livingstone

```
1 print("Name as Provided: ", name)
2 print("Formatted Full Name: ", full_name)
```

Name as Provided: naomi livingstone  
Formatted Full Name: Naomi Livingstone

```
1 # Change string to upper case
2 full_name.upper()
```

'NAOMI LIVINGSTONE'

```
1 # Change string to lower case
2 "NAOMI LIVINGSTONE".lower()
```

'naomi livingstone'

**.title(), .lower() and .upper()**  
methods are used to change cases

# Removing Whitespace from Strings

```
1 # strip or remove whitespace on right side
2 name = "    Brett Wiggins    "
3 name.rstrip()
```

```
'Brett Wiggins'
```

```
1 # remove whitespace on left side
2 name.rstrip()
```

```
'Brett Wiggins'
```

```
1 # remove whitespace on both side
2 # of the string
3
4 name.strip()
```

```
'Brett Wiggins'
```

**.rstrip(), lstrip(), and .strip()** methods are used to remove white space on the right, left and both sides of a string respectively.

# String Formatting with f “ ”

```
1 # string formatting using f""
2 name = input("What is your name? ")
3 ice_cream = input("What is your favorite ice cream? ")
4 print("****Message****")
5 print(f"Hi {name}, do you want medium-size {ice_cream}")
```

What is your name? Nicole

What is your favorite ice cream? vanilla

\*\*\*\*Message\*\*\*\*

Hi Nicole, do you want medium-size vanilla

We can  
format a  
string using  
**f “ ” string**



# String Formatting with .format()

```
1 # string formatting using .format()
2 x = 10
3 y = 5
4 total = x + y
5 # here, the order of the arguments matters
6 print("The sum of {} and {} is {}".format(x,y, total))
```

The sum of 10 and 5 is 15

```
1 # string formatting using .format()
2 x = 10
3 y = 5
4 total = x + y
5
6 # here, the order of the arguments does not matters
7 print("The sum of {x} and {y} is {total}".format(total=total, y=y, x=x))
```

The sum of 10 and 5 is 15

# String Formatting with .format()

```
1  # using .format() while specifying positions
2  # of arguments
3  value = x/y
4  my_string = "{0} plus {1} is {2} while {0} divided by {1} is {3}"
5  print(my_string.format(x, y, total, value))
```

10 plus 5 is 15 while 10 divided by 5 is 2.0

# Splitting a String

```
1 # .split() creates a list
2 full_name = "Marc Hudson"
3 full_name.split()
```

['Marc', 'Hudson']

```
1 # we can then extract first name
2 first_name = full_name.split()[0]
3 first_name
```

'Marc'

```
1 # extract the last name
2 last_name = full_name.split()[1]
3 last_name
```

'Hudson'

```
1 # you could also use tuple assignment
2 # to extract the values
3 first_name, last_name = full_name.split()
4 print("First Name: ", first_name)
5 print("Last Name: ", last_name)
```

First Name: Marc

Last Name: Hudson

# String Splitting in Action

```
1  # use .split() to extract parts of  
2  # an email address  
3  email = "marc.hudson@du.edu"  
4  user_name_domain= email.split("@")  
5  user_name_domain
```

```
['marc.hudson', 'du.edu']
```

```
1  # extract user name from email  
2  user_name = email.split("@")[0]  
3  user_name
```

```
'marc.hudson'
```

```
1  # extract first name from user name  
2  print("First Name: ", user_name.split(".")[0])
```

```
First Name:  marc
```

# Join Strings

```
1  # we want to join user name
2  # and domain in the list into an email
3  user_name_domain
```

```
['marc.hudson', 'du.edu']
```

```
1  # we are joining the list items
2  # to an empty string
3  "".join(user_name_domain)
```

```
'marc.hudsondu.edu'
```

- `.join()` provides a way of concatenating the items of an iterable such as the items of a list into a string.

# Conversion of Strings and Numeric Types

- Integers can be converted to floats and floats to integers
- Numeric types can be converted to strings consisting of numbers surrounded by quotes can be converted to numeric type as well.

```
1 # conversion of strings and numeric types
```

```
1 # convert an integer to a float
2 float(1)
```

```
1.0
```

```
1 # convert a float to an integer
2 int(1.6)
```

```
1
```

```
1 # convert an integer to a string
2 str(2)
```

```
'2'
```

```
1 # convert a string to an integer
2 int("3")
```

```
3
```

# List

- A Python list is a collection of positionally ordered sequence of items.
- For example:
  - `number_list = [1, 2, 7, 20]`
- A list is surrounded by square brackets and its items are separated by commas
- A homogeneous list has items of the same type
- A heterogeneous list has items of different types.
- You could have a nested list of list, list of tuples, list of dictionaries and so on.



# List

- List are mutable. That means you can alter, change or modify a list. You can remove items of a list or add items to the list.
- A list can contain 0 or more objects.

## How to Create a List:

- A list can be using the list literal [ ] or square brackets → **[“A”, “B” “5”]**
- You could also create a list using the **list()** function → **list(“AB5”)**

# How to Create Lists

```
1 # Let's create a List
2 # using square brackets
3 my_list = ["A", "B", 5]
4 my_list
```

```
['A', 'B', 5]
```

```
1 # create a List using list()
2 your_list = list("AB5")
3 your_list
```

```
['A', 'B', '5']
```

```
1 # create an empty list
2 list()
```

```
[]
```

```
1 empty_list = []
2 empty_list
```

```
[]
```

```
1 # Let's create a List of fruits
2 fruits = ["apple", "mango", "lemon", "banana", "orange"]
3 fruits
```

```
['apple', 'mango', 'lemon', 'banana', 'orange']
```

# A Nested List

```
1  # you could have a nested list
2  # for example, a list of list
3  mylist1 = [2, 3, 4]
4  mylist2 = [5, 7, 1]
5  mylist3 = [9, 8, 7]
6  mylist = [mylist1, mylist2, mylist3]
7  mylist
```

```
[[2, 3, 4], [5, 7, 1], [9, 8, 7]]
```

---

# Slicing/Indexing a List

```
1 # extract the first item
2 fruits[0]
```

'apple'

```
1 # extract the second item
2 fruits[1]
```

'mango'

```
1 # extract the last item
2 fruits[-1]
```

'orange'

```
1 # make a copy of the list
2 fruit_copy = fruits[:]
3 fruit_copy
```

['apple', 'mango', 'lemon', 'banana', 'orange']

```
1 # slice from the 2nd to 4th item
2 # Python counts from zero
3 # start index is include
4 # end index is not included
5 fruits[1:4]
```

['mango', 'lemon', 'banana']

```
1 # slice from the 1st to the 3rd item
2 # use negative indexes
3 fruits[:-2]
```

['apple', 'mango', 'lemon']

# Slicing a Nested List

```
1  # slicing a list of list  
2  mylist
```

```
[[2, 3, 4], [5, 7, 1], [9, 8, 7]]
```

```
1  # extract the first list in mylist  
2  mylist[0]
```

```
[2, 3, 4]
```

```
1  # extract the first item in the first list  
2  # syntax: mylist[list_index][item_index]  
3  mylist[0][0]
```

```
2
```

```
1  # extract the first item in the third list  
2  mylist[2][0]
```

```
9
```

# Modify a List: Replace Items

```
1  # Let's replace the first item  
2  # of the list with "grape"  
3  fruits
```

```
['apple', 'mango', 'lemon', 'banana', 'orange']
```

```
1  fruits[0] = "grape"
```

```
1  fruits
```

```
['grape', 'mango', 'lemon', 'banana', 'orange']
```

# List Methods

```
1 # Let's see what methods
2 # belong to a list object
3 print(dir(fruits))
```

```
['__add__', '__class__', '__contains__', '__delattr__  
__', '__delitem__', '__dir__', '__doc__', '__eq__',  
 '__format__', '__ge__', '__getattribute__', '__geti  
tem__', '__gt__', '__hash__', '__iadd__', '__imul_  
__', '__init__', '__init_subclass__', '__iter__', '_  
_le__', '__len__', '__lt__', '__mul__', '__ne__',  
 '__new__', '__reduce__', '__reduce_ex__', '__repr_  
__', '__reversed__', '__rmul__', '__setattr__', '__s  
etitem__', '__sizeof__', '__str__', '__subclasshook  
__', 'append', 'clear', 'copy', 'count', 'extend',  
 'index', 'insert', 'pop', 'remove', 'reverse', 'sor  
t']
```



# Modify a List: Append & Extend

```
1 # Let's add a single item at the end  
2 # of the list: use .append(item)  
3 fruits
```

```
['grape', 'mango', 'lemon', 'banana', 'orange']
```

```
1 fruits.append("melon")  
2 fruits
```

```
['grape', 'mango', 'lemon', 'banana', 'orange', 'melon']
```

```
1 # add two or more items at the end of the list  
2 # use .extend([item1, item2,...])  
3 fruits.extend(["lime", "guava"])  
4 fruits
```

```
['grape', 'mango', 'lemon', 'banana', 'orange', 'melon', 'lime', 'guava']
```

# Modify a List: Delete and Remove Items

```
1 # delete elements using their index
2 fruits
```

```
['grape', 'mango', 'lemon', 'banana', 'orange', 'melon', 'lime', 'guava']
```

```
1 # let's delete the last item
2 del fruits[-1]
3 fruits
```

```
['grape', 'mango', 'lemon', 'banana', 'orange', 'melon', 'lime']
```

```
1 # remove an item using the item itself
2 # let's remove lemon
3 fruits.remove("lemon")
4 fruits
```

```
['grape', 'mango', 'banana', 'orange', 'melon', 'lime']
```

# Modify a List: Pop Items

```
1 # pop() removes and returns the last  
2 # element of the list by default  
3 fruits
```

```
['grape', 'mango', 'banana', 'orange', 'melon', 'lime']
```

```
1 fruits.pop()
```

```
'lime'
```

```
1 # view the list again  
2 fruits
```

```
['grape', 'mango', 'banana', 'orange', 'melon']
```

```
1 # you can specify which item you want  
2 # to pop using the item index  
3 # Let's pop the second item or mango  
4 mango_fruit = fruits.pop(1)  
5 mango_fruit
```

```
'mango'
```

```
1 # view the list of fruits  
2 fruits
```

```
['grape', 'banana', 'orange', 'melon']
```

# Modify a List: Insert Items

```
1 # view the list of fruits
2 fruits
```

```
['grape', 'banana', 'orange', 'melon']
```

```
1 # .insert(index, item) will insert
2 # an item in the specified position
3 # Let's insert "apple" in the first position
4 fruits.insert(0, "apple")
```

```
1 # view the list of fruits
2 fruits
```

```
['apple', 'grape', 'banana', 'orange', 'melon']
```

# Modify a List: Sort Items

```
1 fruits
```

```
['apple', 'grape', 'banana', 'orange', 'melon']
```

```
1 # sort items in a list in ascending order
2 fruits.sort()
3 fruits
```

```
['apple', 'banana', 'grape', 'melon', 'orange']
```

```
1 # sort items in reversed or descending order
2 fruits.sort(reverse=True)
3 fruits
```

```
['orange', 'melon', 'grape', 'banana', 'apple']
```

```
1 # sort items in ascending order again
2 # using the sorted(iterable) function
3 sorted(fruits) # this does not modify the list
```

```
['apple', 'banana', 'grape', 'melon', 'orange']
```

```
1 # you can see that the list is not
2 # modified by the sorted() function
3 fruits
```

```
['orange', 'melon', 'grape', 'banana', 'apple']
```

# List Concatenation

```
1  # concatenate two lists  
2  # this does not modify the list  
3  fruits + ["lime", "cucumber"]
```

```
['orange', 'melon', 'grape', 'banana', 'apple', 'lime', 'cucumber']
```

---

```
1  fruits
```

```
['orange', 'melon', 'grape', 'banana', 'apple']
```

---

# List Comprehension

- List comprehension provide an easy way of processing items in an iterable(such as a list).
- We will use list comprehensions in subsequent lessons but it is worth mentioning about it now.
- List comprehensions work like loops and are very powerful for repetitive tasks.

```
1 # Let's use list comprehensions to
2 # output all list items in upper case
3 [fruit.upper() for fruit in fruits]
```

```
['ORANGE', 'MELON', 'GRAPE', 'BANANA', 'APPLE']
```

```
1 # Let's square each number in a list
2 # using list comprehensions
3 numbers = [1, 2, 3, 4]
4 [num**2 for num in numbers]
```

```
[1, 4, 9, 16]
```



# Tuples

- A tuple is another object type or data structure similar to a list but cannot be modified (tuples are immutable).
- Tuples are positionally ordered sequence of items.
- Used in situations where a sequence of items don't need to be altered.
- For example, if you want to store names of books, authors, and, dates of publication, you can decide to store this data in tuples because the information is fixed and would not change.
- If you were tracking students' GPA's across different quarters, you could use a list since you would be collecting GPA's at different times.



# Tuple

- We can have nested tuples just like list.
- The items of a tuple could be homogeneous or heterogeneous.
- Tuples support indexing and slicing just like lists.
- A tuple can be created with parentheses, no parentheses or with the tuple() function.

```
1 # how to create a tuple
2 # with parentheses
3 my_tuple = (2, 4, 6, 8)
4 my_tuple
```

(2, 4, 6, 8)

```
1 # how to create a tuple
2 # with no parenthesis
3 my_tuple1 = 3, 4, 5, 6
4 my_tuple1
```

(3, 4, 5, 6)

```
1 # how to create a tuple
2 # using the tuple(iterable) function
3 your_tuple = tuple([8, 7, 6, 5])
4 your_tuple
```

(8, 7, 6, 5)

# Tuples Methods

```
1 # Let's see what methods
2 # belong to a tuple object
3 print(dir(my_tuple))
```

```
['__add__', '__class__', '__contains__', '__delattr__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getnewargs__', '__i
gt__', '__hash__', '__init__', '__init_subclass__', '__i
ter__', '__le__', '__len__', '__lt__', '__mul__', '__ne
__', '__new__', '__reduce__', '__reduce_ex__', '__repr
__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'count', 'index']
```

A tuple has fewer methods compared to a list.

# Tuple Methods in Action

```
1 my_tuple
```

```
(2, 4, 6, 8)
```

```
1 # tuple indexing
```

```
2 my_tuple[0]
```

```
2
```

```
1 # what is the index of  
2 # the item, 8  
3 my_tuple.index(8)
```

```
3
```

```
1 # how many 6 are in the tuple  
2 my_tuple.count(6)
```

```
1
```

# Sets

- A set is an unordered collection of unique items. It is neither a sequence nor a mapping.
- Sets are immutable.
- A set can be created using the set literals `{ }` or the **`set()`** function.

```
1 # how to create a set
2 # using set literals
3 my_set = {2, 4, 6, 7}
4 my_set
```

{2, 4, 6, 7}

```
1 # how to create a set
2 # using the set(iterable) function
3 my_set1 = set([1, 3, 4])
4 my_set1
```

{1, 3, 4}

```
1 my_set2 = set("abc")
2 my_set2
```

{ 'a', 'b', 'c' }

# Sets in Action: Finding Differences

```
1  # sets can be used to find the  
2  # difference between two collections  
3  num1 = [2, 3, 4, 5]  
4  num2 = [4, 5, 6, 7]  
5  
6  # what unique items are in num1  
7  # only and not in num2?  
8  diff1 = set(num1) - set(num2)  
9  diff1
```

{2, 3}

```
1  # what unique items are in num2  
2  # only and not in num1?  
3  diff2 = set(num2) - set(num1)  
4  diff2
```

{6, 7}

# Sets in Action: Filter Duplicates

```
1  # sets contain only unique values  
2  # you can use sets to filter duplicates  
3  my_set3 = set([2, 2, 2, 3, 5, 7, 8, 8, 8])  
4  my_set3
```

{2, 3, 5, 7, 8}

---

# Methods of Sets: Intersection and Union

```
1 my_set1
```

```
{1, 3, 4}
```

```
1 my_set3
```

```
{2, 3, 5, 7, 8}
```

```
1 # to find the intersection of two collections
```

```
2 my_set1.intersection(my_set3)
```

```
{3}
```

```
1 # to find the union of two collections
```

```
2 my_set1.union(my_set3)
```

```
{1, 2, 3, 4, 5, 7, 8}
```

## More set methods

```
'__xor__',  
'add',  
'clear',  
'copy',  
'difference',  
'difference_update',  
'discard',  
'intersection',  
'intersection_update',  
'isdisjoint',  
'issubset',  
'issuperset',  
'pop',  
'remove',  
'symmetric_difference',  
'symmetric_difference_update',  
'union',  
'update']
```



# Dictionaries

- We have already discovered that strings, lists and tuples are sequences.
- Sequences support some generic operations such as slicing by index.
- Dictionaries are not sequences but mappings.
- Dictionaries are collections but are different from sequences in that, they are mappings where objects or values are stored in keys.
- Dictionary values are therefore accessed through keys instead of index position as in sequences.



# Dictionaries

- Dictionaries are mutable. You can change the value stored in a key and you can update a dictionary or delete a key and its value.
- Dictionaries are key value pairs and act as look-up tables. They are called hash tables in some programming languages.
- A dictionary can be created using dictionary literals with curly brackets { } or the dict() function.



# How to Create Dictionaries

```
1 # how to create a dictionary
2 # using dictionary literals
3 my_dict = {"name": "John", "age": 25}
4 my_dict
```

```
{'name': 'John', 'age': 25}
```

```
1 # the dictionary values could be collections
2 my_dict1 = {"name": ["John", "Mary"], "age": [25, 23]}
3 my_dict1
```

```
{'name': ['John', 'Mary'], 'age': [25, 23]}
```

```
1 # a dictionary could be created
2 # using a dict(mapping) function
3 my_dict3 = dict(x=5, y=10)
4 my_dict3
```

```
{'x': 5, 'y': 10}
```

# How to Create Dictionaries

```
1 # a dictionary could be created from scratch
2 # using an empty dictionary
3 my_dict4 = {} # this is an empty dictionary
4 my_dict4
```

```
{}
```

```
1 # Let's now populate the dictionary with items
2 my_dict4["employee"] = ["McCarty", "Fischer", "Yang"]
3 my_dict4["emp_ID"] = ["101", "102", "102"]
4 my_dict4["room"] = ["08", "11", "20"]
5 my_dict4
```

```
{'employee': ['McCarty', 'Fischer', 'Yang'],
 'emp_ID': ['101', '102', '102'],
 'room': ['08', '11', '20']}
```

# Nested Dictionaries

- Data that is more complex could be stored in a nested dictionary.
- Suppose we want to record both the first name and last name of employees separately, we can have something like this.

```
1 # nested dictionaries
2 my_dict5 = {"employee_name":
3             {"first_name": ["Jim", "Pat", "Hung"],
4              "last_name": ["McCarty", "Fischer", "Yang"]},
5             "emp_ID": ["101", "102", "102"],
6             "room": ["08", "11", "20"]}
7
8 my_dict5
```

```
{'employee_name': {'first_name': ['Jim', 'Pat', 'Hung'],
                    'last_name': ['McCarty', 'Fischer', 'Yang']},
 'emp_ID': ['101', '102', '102'],
 'room': ['08', '11', '20']}
```

# A Dictionary is Mutable

```
1  # a dictionary can grow  
2  my_dict
```

```
{'name': 'John', 'age': 25}
```

```
1  # Let's add John's salary to my_dict  
2  my_dict["salary"] = 50000  
3  my_dict
```

```
{'name': 'John', 'age': 25, 'salary': 50000}
```

# How to Access Values in a Dictionary

```
1 # how to access the values in a dictionary  
2 my_dict["name"]
```

'John'

```
1 # extract salary  
2 my_dict["salary"]
```

50000

We will also take a look at how to loop through a dictionary and extract values, after we have covered loops

# How to Access a Nested Dictionary

```
1 my_dict5
```

```
{'employee_name': {'first_name': ['Jim', 'Pat', 'Hung'],  
  'last_name': ['McCarty', 'Fischer', 'Yang']},  
  'emp_ID': ['101', '102', '102'],  
  'room': ['08', '11', '20']}
```

```
1 # extract employee_name  
2 my_dict5["employee_name"]
```

```
{'first_name': ['Jim', 'Pat', 'Hung'],  
  'last_name': ['McCarty', 'Fischer', 'Yang']}
```

```
1 # extract list of first name in employee name  
2 my_dict5["employee_name"]["first_name"]
```

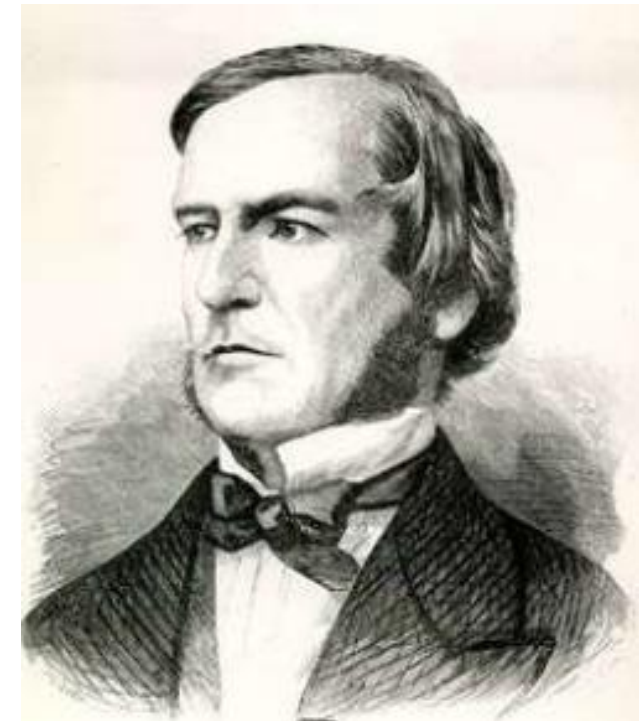
```
['Jim', 'Pat', 'Hung']
```

```
1 # extract first name in list of first name  
2 # in the employee_name key  
3 my_dict5["employee_name"]["first_name"][0]
```

# Boolean (bool) Type

- There is another data type in Python called the Boolean type (bool) named after the English Mathematician, George Boole.
- George Boole came up with the Boolean logic which involves three operators (**and**, **or**, **not**) and two values (**True**, **False**).

George Boole



<https://www.britannica.com/biography/George-Boole>





# Boolean Type

- The idea behind the Boolean logic is that every value can be reduced to either **True** or **False**.
- Therefore, Boolean type or the bool in Python is an object with a value, **True** or **False**.
- In Python, **True** or **False** are called Boolean values.
- Any expression that evaluates to True or False is called a **Boolean expression** or a conditional expression.
- Boolean expressions are very powerful in loops. They are used in if/elif/else statements.
- A while loop could also be initialized with a Boolean Flag.

# Boolean Values

Every object has a boolean value

```
1 # the inter value, zero is False
2 bool(0)
```

False

```
1 # every integer except zero is True
2 bool(1)
```

True

```
1 # empty list, sets, etc are False
2 bool([])
```

False

```
1 bool({})
```

False

```
1 # a collection with an item is True
2 bool(["orange"])
```

True

```
1 # what data type is "True"?
2 type(True)
```

bool

# Boolean Operators

- **and, or, not** are Boolean operators. Let's illustrate how the Boolean operators work with Boolean values.
- Practically, Boolean operators are not applied directly to **True** or **False**, but could be applied to Boolean variables (flag) and Boolean expressions.

How Boolean operators work

1	<b>not True</b>
---	-----------------

False

1	<b>not False</b>
---	------------------

True

1	<b>True and True</b>
---	----------------------

True

# Boolean Operators

1 True and False

False

1 True or True

True

1 True or False

True

Operation	Result
x and y	Returns True only if both x and y are True, otherwise, False.
x or y	Returns True if x or y
not x	Returns True if x is False and returns False If x is True

**not** has the highest precedence, followed by **and**, then **or**

# Apply Boolean Operators to Variables

```
1 # Let's initialize the variables
2 cold = True
3 windy = False
```

```
1 cold and windy
```

False

```
1 not (cold and windy)
```

True

```
1 cold or windy
```

True

```
1 not cold
```

False

```
1 not windy
```

True

```
1 not cold and windy
```

False

```
1 cold and not windy
```

True

# Boolean Expressions

- A Boolean expression is any expression that evaluates to True or False.
- For example: **5 > 0** is a Boolean expression and evaluates to True.
- Boolean expressions contain comparison or relational operators.

- Examples of comparison (relational) operators.

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

# Boolean Expressions

```
1 4 > 5
```

False

```
1 4.5 < 5
```

True

```
1 10 >= 8
```

True

```
1 4 != 5
```

True

```
1 5 == 5
```

True

```
1 math_score = 70
2 reading_score = 80
3 art_score = 80
4 music_score = 90
```

```
1 math_score > art_score
```

False

```
1 math_score < reading_score and reading_score == art_score
```

True

```
1 art_score < music_score and math_score != reading_score
```

True

```
1 art_score < music_score or math_score > reading_score
```

True

We can also apply Boolean operators (not, and, or) to Boolean expressions

# Let's Wrap it up with a Game!

```
1 name = input ("What is your name?: ")
2 age = input("How old are you? ")
3 age = eval(age)
4
5 if age <= 18:
6     print("Hi kido, sorry, you\n"
7         "are not old enough to create an account\n"
8         "on this website")
9 elif age > 18 and age < 80:
10    print("You are eligible to create an account\n"
11        "on this website")
12 else:
13    print("Hi Grani, you are very welcome to this site\n"
14        "You will have a 50 percent discount for any product\n"
15        "you buy from this site.")
```

```
What is your name?: Nash
How old are you? 15
Hi kido, sorry, you
are not old enough to create an account
on this website
```

```
1 name = input ("What is your name?: ")
2 age = input("How old are you? ")
3 age = eval(age)
4
5 if age <= 18:
6     print("Hi kido, sorry, you\n"
7         "are not old enough to create an account\n"
8         "on this website")
9 elif age > 18 and age < 80:
10    print("You are eligible to create an account\n"
11        "on this website")
12 else:
13    print("Hi Grani, you are very welcome to this site\n"
14        "You will have a 50 percent discount for any product\n"
15        "you buy from this site.")
```

```
What is your name?: Deborah
How old are you? 85
Hi Grani, you are very welcome to this site
You will have a 50 percent discount for any product
you buy from this site.
```



# Congratulations! You made it through lesson 2 of Python Fundamentals



Get ready to start “looping” in our next lesson! Happy Pythoning!

# End of Lesson

