Python (By Youssef Samir)

Variables and Data Types

- Variable can be thought of as a container of data with a specific label that allows easy access to the data.
    - This definition is half true, because variables in Python are labels that reference a data object with a unique ID found in memory.
- Python is a Dynamic and Strongly typed Language, but what do those terms mean?
    - **Dynamic**: Dynamically-typed means that a type of variable can change while the program is running.

```
x = 5 #x is integer
x = "python" #x is string
```

- **Strong**: Strongly-typed means that variables do not automatically change its own type to conform to an operation.

```
x = 5 #x is integer
x = x + "python" #error, cannot mix integers to strings
```

Defining Variables

- Variables in Python can be defined in multiple ways

```
x = 3 # Will create variable x and set it to 3
a = b = c = 5 # Will create variables a, b and c and set them to 5
i, j, k = 1, 2, 3 # Is a shorthand for i = 1 j = 2 k = 3
```

Naming Rules of Python Variables

- Variable names can only contain letters, digits and underscores (_).
- A variable name cannot start with a digit.

```
message_1 = "hello" #Valid variable name
1_message = "hello" #Error: Invalid variable name
```

- Variable names are case-sensitive, so “var”, “Var” and “vAr” are all different variables.
- Spaces are not allowed in variable names.

```
hello_message = "hello" #Valid variable name
hello message = "hello" #Error: Invalid variable name
```

- Avoid using python keywords as names for variables (such as `for`, `if`, `elif`, etc..).
- Use short, but descriptive names for your variables.

```
s_n = “Joe” #Short, but not descriptive                ✘
name_of_the_student = “Joe” #Descriptive, but long     ✘
student_name = “Joe” #Short and Descriptive            ✓
```

Data Types in Python

There are multiple data types in Python, some of them are:

Data Type	Example
int	`x = 10`
float	`x = 3.14`
bool (Boolean)	`x = True`
str (String)	`x = "Hello"`
list	`x = [1, 2, 3]`
Tuple	`x = (1, 2, 3)`
Set	`x = {1, 2, 3}`
dict (Dictionary)	`x = {1: "One", 2: "Two", 3: "Three}`
range	`x = range(10)`
NoneType	`x = None`

Strings in Python

- A String is a series of characters, that can be used to represent textual data.
- To define a string to hold a text, wrap them in either single quotes (‘’) or double quotes (“”).

```
x = "This is a string"
x = 'This is also a string'
```

- The ability to use either types of quotes, allows us to use the other type of quotes inside the string.

```
x = "That's a valid string"
x = 'I told him "Python is great!"'
x = 'That's an invalid string' #Error: Unterminated String
```

- What to do when we want to use both types of quotes inside a string? We use Escape Sequences!

Escape Sequence

- An escape sequence or escape character, is a special kind of character that starts with a backslash (`\`) then followed by a single character or a sequence of them, to specify illegal characters.
  - Even though, escape sequences can be 4 characters in length, yet they're interpreted as a single character.

Sequence	Result
`\n`	Newline
`\t`	Horizontal Tab
`\\`	Backslash
`\'`	Single Quotes
`\"`	Double Quotes
`\a`	Alarm
`\xhh` (Where `h` is a hexadecimal digit)	Hexadecimal Value

Multi-line String

- Text that is spread over multiple lines can only be represented in python using either of the following ways.
  - **Using Newline Escape Sequence (`\n`)**

```
x = "Hello, there\nGoodbye!"
```

- **Using Triple Quotes (`'''` or `"""`)**

```
x = """Hello, there
Goodbye!"""
```

- Both of those variables will print the string in the same multi-line format.

Formatting Strings

- As we already know, the `print()` function can be used to display information on the terminal (CLI), so we can use it to display strings that we write or generate.
- We can also use variables with the string inside `print()` to display a formatted info about that variable; that is called **String Concatenation.**

```
country = "Egypt"
print("I'm from " + country) #Will print "I'm from Egypt"
```

- But this method of concatenation will show its shortcomings as soon as we want to print multiple variables and strings. Also when we want to print variables of types other string, we will get errors (Due to being Strong-Typed) and we will have to convert every type to a string.

```
birth_year = 1997
print("I was born in " + birth_year) #Error: can only concatenate strings
print("I was born in " + str(birth_year))
```

- As of Python 3.6, a special kind of string, called F-string was introduced to allow string formatting easily.
- It easily enables us to embed variables inside strings, without the need of concatenation and type conversion!

```
country = "Egypt"
birth_year = 1997
print(f"I'm from {country} and I was born in {birth_year}")
#Will print "I'm from Egypt and I was born in 1997"
```

- If you have to work on a python version that doesn’t support F-strings, we can use the Format() function.

```
country = "Egypt"
birth_year = 1997
print("I'm from {} and I was born in {}".format(country, birth_year))
#Will print "I'm from Egypt and I was born in 1997"
```

String Methods

- A method is property of an object that define behavior specific to it alone (Will know more about it in OOP).
- Python strings has plenty of useful methods, such as:
    - `.title()`: Will change the string case into title case. (Capitalizes every first letter of the words).

    ```
    msg = "hello world"
    print(msg.title())
    #Will print "Hello World"
    ```

    - `.upper()`: Will change all the letters to uppercase.

    ```
    msg = "hello world"
    print(msg.upper())
    #Will print "HELLO WORLD"
    ```

    - `.lower()`: Will change all the letters to lowercase.

    ```
    msg = "HELLO WORLD"
    print(msg.lower())
    #Will print "hello world"
    ```

    - `.find(value)`: Will search the string for a certain substring and return its index.

    ```
    msg = "hello world"
    print(msg.find("ll"))
    #Will print 2, because "ll" starts at index 2
    ```

  - `.lstrip()` and `rstrip()`: Will remove leading and trailing whitespaces respectively.

    ```
    msg_1 = "hello world          "
    print(msg_1.rstrip())
    #Will print "Hello World"

    msg_2 = "            hello world"
    print(msg_2.rstrip())
    #Will print "Hello World"
    ```

  - `.count(value)`: Will count the number of occurrences of a certain value inside the string.

    ```
    msg = "hello world"
    print(msg.count("l"))
    #Will print 3, because "l" occurs 3 times in the string
    ```

  - `.join(iterable)`: Will join all the values in an iterable (such as a list) by a certain delimiter.

    ```
    names = ["Ahmed", "Mohamed", "Mahmoud"]
    print(" and ".join(names))
    #Will print "Ahmed and Mohamed and Mahmoud"
    ```

  - And many other useful methods.

- We can also use functions such as `len()`, `min()` and `max()` to measure the size of the string.
  - `len()`: Will measure the Length of the String.

    ```
    msg = "Python"
    print(len(msg))
    #Will print 6
    ```

  - `max()`: Will return the character with the highest ASCII order.

    ```
    msg = "Python"
    print(max(msg))
    #Will print y
    ```

- `min()`: Will return the character with the lowest ASCII order.

  ```
  msg = "Python"
  print(min(msg))
  #Will print P
  ```

Booleans in Python

- Python has a Boolean type that represents the Result from Logical Operations.
- It can only be either `True` or `False`.
- There is also, the concept of Truthy and Falsey Values
  - `0` will always evaluate to `False`, otherwise (Whether Positive or Negative) evaluate to `True`.
  - Empty Strings and Collections, such as `""`, `[]`, `{}` will evaluate to `False`, otherwise, evaluate to `True`.
  - `None` will always evaluate to `False`.
- Those rules are always be helpful to write shorter code

Numbers in Python

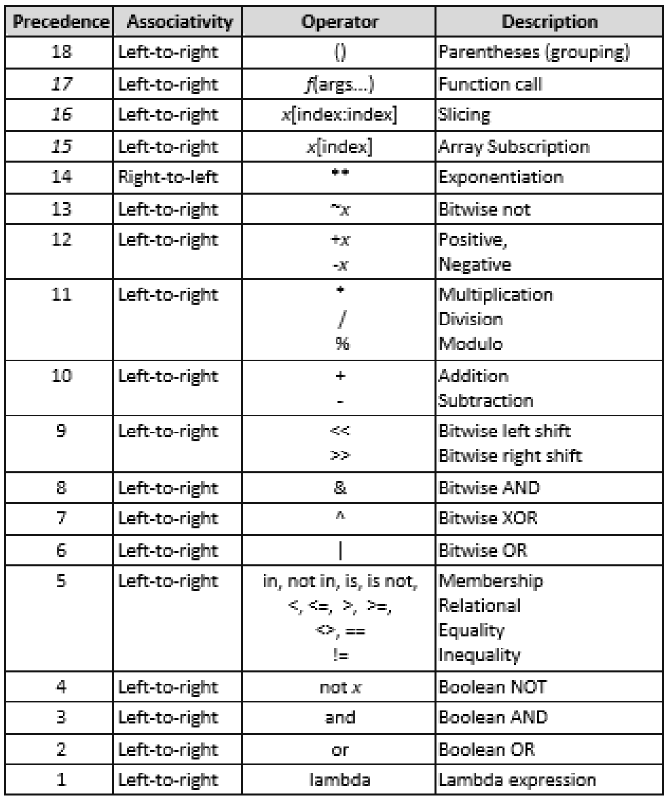
- Python has 3 main types of numbers; Integers, Floats and Complex Numbers.
- Integers represent whole numbers, those that don’t have fractional part with them. Integers can be defined as `x = 4`.
- Floats represent decimal numbers, those that consists of a whole and a fractional part, they can be defined as `x = 3.7`.
- Integers and floats can be mixed together in arithmetic operations, but be warry that the result type will always be a float, even if the result has no fractions.

  ```
  x = 4
  y = 2.0
  print(type(x / y))
  #Will print <class 'float'>
  ```

Operators and their Precedence and Associativity

Arithmetic
Comparison
Logical
Assignment
Bitwise
Identity
Membership

- Arithmetic and Logical operations can be performed through the use of Operators, Python has 7 Categories of operators, each of them has its own Precedence (Priority) and Order of Evaluation (Associativity).



Precedence	Associativity	Operator	Description
18	Left-to-right	()	Parentheses (grouping)
17	Left-to-right	*f*(args...)	Function call
16	Left-to-right	*x*[index:index]	Slicing
15	Left-to-right	*x*[index]	Array Subscription
14	Right-to-left	**	Exponentiation
13	Left-to-right	~*x*	Bitwise not
12	Left-to-right	+*x* -*x*	Positive, Negative
11	Left-to-right	* / %	Multiplication Division Modulo
10	Left-to-right	+ -	Addition Subtraction
9	Left-to-right	<< >>	Bitwise left shift Bitwise right shift
8	Left-to-right	&	Bitwise AND
7	Left-to-right	^	Bitwise XOR
6	Left-to-right	|	Bitwise OR
5	Left-to-right	in, not in, is, is not, <, <=, >, >=, <>, == !=	Membership Relational Equality Inequality
4	Left-to-right	not *x*	Boolean NOT
3	Left-to-right	and	Boolean AND
2	Left-to-right	or	Boolean OR
1	Left-to-right	lambda	Lambda expression

- **Precedence** refers to the Priority of an operator to get executed first when it is mixed with other operators of lower Precedence.

```
print(5 + 1 * 3) # Will print 8, because * has a higher precedence than +
print(5 ** 2 * 3) # Will print 75, because ** (Power) has a higher precedence than *
print((5 + 1) * 3) # Will print 18, because () will always execute first
```

- **Associativity** refers to the order of evaluation of operators that have the same precedence, whether it is evaluated from **Left-to-Right** or **Right-to-Left**.

```
print(5 * 4 / 3) # Will run 5 * 4 first then divide the result over 3 because
                 # the Associativity is from Left-to-Right
```

Comments

- Comments in Python are lines that are not processed by the Interpreter.
- Comments can be used as a description for a certain functionality or to prevent the Interpreter from executing a certain line of code.

```
#print("hello")
# Running this code will not print anything, because the above print command is commented-out
```

- We can also use the Triple quoted string as a multi-line comment, but do not assign it to a variable.

```
"""
This is a multi-line comment,
that is commonly used a description for a function
and its paramaters and return values
but remember to never assign it to a variable.
"""
```

Lists

- A list a mutable collection of data that are grouped together by the developer based on a common attribute.
    - Mutable means that it can be modified after its creation.
    - A list is also an iterable, because we can iterate over its elements as we will see in the loops section.
- To define a list, we use the Square Brackets `[]` where it can contain initial values or be kept empty.

```
l_empty = [] # Creates an empty list
l_defined = [1, 2, 5, 6] #Creates a list of 4 elements
```

- We can check the existence of a certain value inside a list, using the `in` and `not in` operators.

```
l = [1, 2, 3, 4, 5]
print(2 in l) # Will print True, because 2 exists in the List
print(6 not in l) # Will print True, because 6 does not exist in the List
```

- This is useful with Decision Making using if-conditions.

Accessing, Changing, Adding and Removing Elements

- List elements can be accessed using the Index-of operator `list[index]`.
- Elements in the List will always start at index `0` not `1`.

```
l = [1, 2, 3, 4, 5]
print(l[1]) # Will print 2, because index 1 refers to the second element
```

- We can also access the elements from the opposite direction, by using negative indices.

```
l = [1, 2, 3, 4, 5]
print(l[-1]) # Will print 5, because index -1 is actually the value of len(l) - 1
```

- By accessing an element of a list using the Index-of operator, we can also change it.

```
l = [1, 2, 3, 4, 5]
l[1] = 10
print(l) # Will print [1, 10, 3, 4, 5], because the second element has been modified
```

- Since Lists are mutable, we can also add new values at its end after defining it, using the `append()` method.

```
l = [1, 2, 3, 4, 5]
l.append(7)
print(l) # Will print [1, 2, 3, 4, 5, 7]
```

- We can also add values to it at any index, using the `insert()` method.

```
l = [1, 2, 3, 4, 5]
l.insert(0, 20)
print(l) # Will print [20, 1, 2, 3, 4, 5]
```

- We can add the values of a list to another list, using the `extend()` method.

```
l = [1, 2, 3, 4, 5]
l.extend([6, 7, 8, 9])
print(l) # Will print [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- To remove an element from a list, we can do that in two ways
  - `del list[index]`: Will delete the element referred to.

  ```
  l = [1, 2, 3, 4, 5]
  del l[1]
  print(l) # Will print [1, 3, 4, 5]
  ```

  - `pop(index)`: Will delete and return the element at that index.

  ```
  l = [1, 2, 3, 4, 5]
  l.pop(1)
  print(l) # Will print [1, 3, 4 ,5]
  ```

List Methods

- Python has multiple of useful methods for the List type.
  - `.clear()` Will remove all the elements in the List.

    ```
    l = [1, 2, 3, 4, 5]
    l.clear()
    print(l) # Will print []
    ```

  - `.remove(value)` Will remove the first occurrence of a value in the List.

    ```
    l = [1, 2, 3, 4, 5]
    l.remove(1)
    ```

```
print(l) # Will print [2, 3, 4, 5]
```

- `.count(value)` Will count the occurrences of a certain value in the List.

  ```
  l = [1, 2, 3, 4, 5, 1]
  print(l.count(1)) # Will print 2
  ```

- `.reverse()` Will reverse the List.

  ```
  l = [1, 2, 3, 4, 5]
  l.reverse()
  print(l) # Will print [5, 4, 3, 2, 1]
  ```

- `.index(value)` Will return the index of a certain value (-1 if not found).

  ```
  l = [1, 2, 3, 4, 5]
  print(l.index(3)) # Will print 2
  ```

- `.sort(reversed=False)` Will sort the List in an Ascending order by default, unless `reversed` is set to True.

  ```
  l = [3, 2, 4, 1, 5]
  l.sort()
  print(l) # Will print [1, 2, 3, 4, 5]
  -------------------------------------
  l = [3, 2, 4, 1, 5]
  l.sort(reversed=True)
  print(l) # Will print [5, 4, 3, 2, 1]
  ```

- We can also use functions with lists, such as `min()`, `max()`, `len()`, `sum()` and `sorted()`.
  - `min()` Will return the minimum value in the list

    ```
    l = [1, 2, 3, 4, 5]
    print(min(l)) # Will print 1
    ```

  - `max()` Will return the maximum value in the list

```
l = [1, 2, 3, 4, 5]
print(max(l)) # Will print 5
```

- `len()` Will return the length of the list

```
l = [1, 2, 3, 4, 5]
print(len(l)) # Will print 5
```

- `sum()` Will return the sum of all values in the list

```
l = [1, 2, 3, 4, 5]
print(sum(l)) # Will print 15
```

- `sorted()` Will return a sorted version of the list, without modifying its original order, as is the case with the `sort()` method.

```
l = [3, 2, 4, 1, 5]
print(sorted(l)) # Will print [1, 2, 3, 4, 5]
---------------------
l = [3, 2, 4, 1, 5]
print(sorted(l, reversed=True)) # Will print [5, 4, 3, 2, 1]
```

List Slicing

- We can obtain certain parts of a list, by slicing it using the Slicing Operator `list[start:end:step]`.
    - Remember that the resulting sub-list, will not include the end index, because it is exclusive.
- We don’t have to always use the Full form, defined above, to get slices of a list.
    - We can set the starting index only using `list[start:]`.
    - We can set the ending index only using `list[:end]`.
    - We can set the step value only using `list[::step]`.
    - We can set both the start and stop index using `list[start:end]`.
    - We can set both the start and step values using `list[start::step]`.

- From the above examples, we see the slicing operator is very flexible and can be used with freedom, without adhering to the full form.

```
l = [1, 2, 3, 4, 5]
print(l[2:]) # Will print [3, 4, 5]
print(l[:3]) # Will print [1, 2, 3]
print(l[::-1]) # Will print [5, 4, 3, 2, 1]
print(l[2:4]) # Will print [3, 4]
print(l[1:4:2]) # Will print [2, 4]
```

- We can also use slicing to copy lists by value, instead of copying a reference to it, as with the default behavior.

```
l_1 = [1, 2, 3, 4]
l_2 = l_1
l_2[0] = 5 # This change will also affect l_1, because both l_1 and l_2 reference the same object
-------------------
l_1 = [1, 2, 3, 4]
l_2 = l_1[:]
l_2[0] = 5 # This will not affect l_1, because we've actually made a different copy of l_1
```

Tuples

- Tuples are immutable collections, that cannot be modified or changed once defined.
- To define a tuple, we use the Parenthesis `()` with the values inside it.

```
t_1 = (1, 2)
t_1[0] = 3 #Error: Cannot modify values inside a tuple
t_1.append(3) #Error: append does not exist for tuples
```

- Tuples are used when a collection is not supposed to change after its definition.

Decision Making

- If-conditions can be used for decision-making based on a condition or multiples of them.

```
num = 7
if num > 5:
  print("The number is bigger than 5") # Will be printed because 7 > 5
```

- If-conditions can be accompanied by an `else` block, that gets executed if the condition failed.

```
num = 4
if num > 5:
  print("The number is bigger than 5")
else:
  print("The number is less than 5") # Will be printed because 4 < 5
```

- We can also check for other possible related conditions, to be checked if the condition in the if block failed, using the `elif` blocks.

```
num = 4
if num > 5:
  print("The number is bigger than 5")
elif num <= 4:
  print("The number is less than or equal 4") # Will be printed because 4 <= 4
else:
  print("The number is less than 5")
```

- We can use countless number of `elif` blocks to create an if-elif-else chain.

```
num = 6
if num == 5:
  print("The number is 5")
elif num == 4:
  print("The number is 4")
elif num == 3:
  print("The number is 3")
elif num == 6:
  print("The number is 6") # Will be printed because 6 == 6
else:
  print("The number is unknown")
```

- We can specify multiple conditions in a single `if` or `elif` block using the `and` and `or` operators.

```
num = 6
if num >= 5 and num <= 10:
  print("The number is between 5 and 10") # Will be printed because 5 <= 6 <= 10
else:
  print("The number is not in range")
```

- Or better yet, Python allows us to write ranges the same way we write it in math equations.

```
num = 6
if 5 <= num <= 10:
  print("The number is between 5 and 10") # Will be printed because 5 <= 6 <= 10
else:
  print("The number is not in range")
```

- We can check for the opposite of a condition by using the `not` operator.

```
is_it_true = False

if not is_it_true:
  print("No it is not true") # Will print because is_it_true is False
else:
  print("Yes it is true")
```

- We can also nest if-conditions inside each others.

```
num = 6
if num >= 5:
  if num <= 10:
    if num >= 6:
      print("That's the correct number")
    else:
      print("Number is less than 6")
  else:
    print("Number is greater than 10")
else:
  print("Number is less than 5")
```

Some notes about if-statements

- Always check if you have the correct amount of indentation (spaces and tabs) beneath each `if` `elif` and `else` statements, because any code written beneath it will only get executed if it has the correct indentation.

```
if True:
print("Hello") # This statement is not owned by the if-condition, also it will be Error
--------------
if True:
  print("Hello") # This statement is owned by the if-condition
```

- You can for equality and non-equality using `==` and `!=` respectively, but always be sure when you check for equality to use the `==` operator not the `=` assignment operator.

```
num = 7
if num = 7: # Error: You cannot use the assignment operator as a condition
  print("The number is 7")
```

- Try as much as possible to combine conditions using `and` and `or` operators instead of nesting and creating other if-elif-else chains, unless it is required.

```
name = "ahmed"
age = 15

if name == "ahmed" or age >= 18:
  print("Allowed") # Will print because at least one of the conditions is satisfied
else:
  print("Blocked")
```

For Loops

- A for loop can be used to iterate over an iterable (such as a list, tuple or a range object).
- It has this form.

```
for element_name in iterable:
  #statements
```

- For example, we can use it to print the square of every element in a list.
  - Using it this way, will not allow us to modify the list itself.

```
nums = [1, 2, 3, 4]
for i in nums:
  print(i ** 2)
```

- Or we can use it in conjunction with a `range` object, which generates numbers in a linear fashion.
  - It has multiple forms, such as.
    - `range(end)` Will generate numbers from 0 to end - 1, with a step of 1.

- `range(start, end)` Will generate numbers from start to end -1, with a step of 1.
- `range(start, end, step)` Will generate number from start to end - 1, with a step of `step`.

```
nums = [1, 2, 3, 4]
for i in range(len(nums)):
  nums[i] = nums[i] ** 2
print nums # Will print [1, 4, 9, 16]
```

Skipping iterations using `continue`

- We can skip an iteration if a certain condition meets, using the `continue` keyword.

```
for i in range(5):
  if i == 3:
    continue
  print(i)
# The Loop will print 0 1 2 4, skipping 3 because of the condition
```

Early Termination of the loop using `break`

- We can break out of the loop immediately, by using the `break` keyword.

```
for i in range(10):
  if i == 4:
    break
  print(i)
# The Loop will print 0 1 2 3, and stops at 4, even though it was supposed to go to 9
```

- Python, gives us the ability to run code if the loop did not break and it reached its end, by using the `else` block.

```
for i in range(5):
  if i == 6:
    break
  print(i)
else:
  print("The loop reached its end") # Will be printed, because the loop did not break
```

While Loops

- Another kind of loops is the `while` loop, which is used to execute code while a condition is satisfied.

```
num = 0
while num < 9:
  print(num)
  num += 1
# Will print every number from 0 to 8, because at 9, the (num < 9) is not satisfied
```

- Make sure that the loop terminates successfully, otherwise it can be an infinite loop (Unless that is what you want).

```
num = 0
while num < 9:
  print(num)
# Will print 0 an "infinite" number of times because the condition is always satisfied
```

- We can use the keywords we used with the for loop; `continue`, `break` and `else` with the while loop.

```
num = 0
while num < 9:
  if num == 3:
    continue
  if num == 7:
    break
  print(num)
else:
  print("Loop terminated successfully") # Will not print, because the loop is terminated at 7
```

Accepting user input using `input()`

- We can accept input from user on the Terminal (CMD), by using the `input()` function, which can be supplied with an optional message.

```
name = input("What is your name? ")
print(f"Hello, {name}")
```

- The function will always return a string, even if the user intended to write another type, hence we can use type conversions to convert the input to its correct type.

```
num_1 = int(input())
num_2 = int(input())

print(f"The sum of num_1 and num_2 is {num_1 + num_2}")
```

Dictionaries

- A dictionary is a collection of Key-Value pairs called “Items”.

```
nums = {"One": 1, "Two": 2, "Three": 3}
```

- To access a Value inside a dictionary, we use its key as an “index”.

```
nums = {"One": 1, "Two": 2, "Three": 3}
print(nums["Two"]) # Will print 2
```

- The keys can be of any type, as long as it is immutable (So we cannot use lists as keys, yet we can use tuples).
- The keys cannot be modified after it is created, but their values can be modified.

```
nums = {"One": 1, "Two": 2, "Three": 3}
nums["One"] = 5
print(nums["One"]) # Will print 5
```

- To add new Items, we can do it as follows.

```
nums = {"One": 1, "Two": 2, "Three": 3}
nums["Four"] = 4
print(nums) # Will print {'One': 1, 'Two': 2, 'Three': 3, 'Four': 4}
```

- If we tried to access a key that does not exist, it will give us an error, so a better way to access the items, is using the `get(key)` method.

```
nums = {"One": 1, "Two": 2, "Three": 3}
print(nums["Four"]) #Error: Key Error
-------------------
print(nums.get("Four")) # Will print None
```

- We can access the keys, values or tuples of the Key-Value pairs, by using the methods `keys()`, `values()`, `items()`.

```
nums = {"One": 1, "Two": 2, "Three": 3}
print(nums.keys()) # Will print dict_keys(['One', 'Two', 'Three'])
print(nums.values()) # Will print dict_values([1, 2, 3])
print(nums.items()) # Will print dict_items([('One', 1), ('Two', 2), ('Three', 3)])
```

- We can remove Key-Value pairs by either using `del dict[key]` or `dict.pop(key)`.

```
nums = {"One": 1, "Two": 2, "Three": 3}
del nums["One"]
nums.pop("Two")
print(nums) # Will print {'Three': 3}
```

- We can use the for loop to iterate over the dictionary as follows.

```
nums = {"One": 1, "Two": 2, "Three": 3}
for k in nums:
  print(f"{k} is {nums[k]}") # Will print One is 1 Two is 2 Three is 3
```

- Since, each Key-Value pair is, in fact, a tuple, we can do the following.

```
nums = {"One": 1, "Two": 2, "Three": 3}
for k, v in nums:
  print(f"{k} is {v}") # Will print One is 1 Two is 2 Three is 3
```

- We can use any type as a value, even dictionaries themselves!

```
students = {"id_0": {"name": "Ahmed", "age": 18},
            "id_1": {"name": "Yusuf", "age": 20},
            "id_2": {"name": "mohamed", "age": 21}}
print(students["id_2"]) # Will print {'name': 'mohamed', 'age': 21}
```