

# Python 1 - Overview

Bootcamp will cover Python fundamentals while making a music playlist program

- Evaluating primitive types in python: `type()`
- Declaring variables and variable declaration conventions: `=`
- Math Operators and string concatenation: `(+ , - , * , / , %)`
- IF and WHILE statements with conditional operators: `(== , > , >= , break)`
- User input: `input()`
- Data collections - Lists: `([ ], append(), insert(), del, pop(), len(), sort())`
- Data collections - Dictionaries: `({ }, [ ], insert(), del, clear(), keys(), values())`
- Declaring custom functions: `def, return`
- Classes and object oriented programming: `class(), __init__(), methods`
- Automating with FOR loops: `for, in`

## Jupyter Notebook

This is a web-based application (runs in the browser) that is used to interpret Python code.

- To add more code cells (or blocks) click on the '+' button in the top left corner
- There are 3 cell types in Jupyter:
  - Code: Used to write Python code
  - Markdown: Used to write texts (can be used to write explanations and other key information)
  - NBConvert: Used convert Jupyter (.ipynb) files to other formats (HTML, LaTeX, etc.)
- To run Python code in a specific cell, you can click on the '**Run**' button at the top or press **Shift + Enter**
- The number sign (#) is used to insert comments when coding to leave messages for yourself or others. These comments will not be interpreted as code and are overlooked by the program

## Data Types

- Four primitive types in Python
  1. Integers
  2. Booleans
  3. Floats
  4. Strings
- Types may be changed using `int(), str(), float(), and bool()` methods

```
In [1]: # The type() function will return the data type of the data passed to it  
type("Hello!")
```

Out[1]: str

```
In [2]: type(True)
```

Out[2]: bool

```
In [3]: type(3.14)
```

Out[3]: float

```
In [4]: type(3)
```

Out[4]: int

```
In [5]: # Casting - converting from one data type to another  
print(type(float(3)))  
print(int(3.55))  
print(bool('Hello'))
```

<class 'float'>

3

True

## Variables

- May consist of letters, numbers, and underscores, but not spaces.
  - **Cannot start with a number.**
- Avoid using Python keywords (for, if, and, or, etc.)
- Be careful when using 1s and lower case ls, as well as 0s and Os.
- Keep it short.
- Example: phone\_num = 647606

```
In [6]: # In the code below, the variable `hours_worked` has been assigned  
# an integer value of 10.  
hours_worked = 10
```

```
In [7]: print(hours_worked)
```

10

## Math Operators

- Addition, Subtraction, Multiplication and Division may be done using basic math operators (+, -, \*, /, %).
- Many built-in string methods (title, upper, lower, index, split).
- Python will also try to interpret your code with other data types
  - (+) may be used with strings!

```
In [8]: # Create two variables, price1 and price2 that have float values representing  
price1 = 3.40  
price2 = 2.51  
  
# Create a new variable whose value is the sum of the duration of both songs  
tot_price = price1 + price2  
# Python can perform all the typical mathematical operations  
diff_price = price1 - price2  
mult_price = price1 * price2  
div_price = price1 / price2  
print(tot_price)
```

5.91

```
In [9]: # Define string variables name, job, and tool  
name = "Peter"  
job = "works with"  
tool = "Python"
```

```
In [10]: # We can concatenate (combine) strings together using the addition (+) symbol
employment = name + " " + job + " " + tool
print(employment)
#A few of the methods strings come with! Check output to see how each works
print(employment.title())
print(employment.lower())

print(employment.index("works"))
print(employment.split(" "))
print(employment.replace("IT", "Finance"))
```

```
Peter works with Python
Peter Works With Python
peter works with python
6
['Peter', 'works', 'with', 'Python']
Peter works with Python
```

```
In [11]: # A few ways to combine strings and variables
# With F strings, variables go directly into a string! Even methods!
print(f"{name} works with {tool.upper()}")
```

```
Peter works with PYTHON
```

```
In [12]: # A boolean can only have one of two values. Either they are "True" or "False"
# Variables "yes" and "no" have been assigned boolean variables of "True" and "False"

yes = True
no = False
```

## IF and WHILE Statements

- Will only run indented code if condition is true
- Make use of **conditional operators** to create tests
  - (==) will return true if both variables are equal
  - (>) will return true if left variable is larger
  - (>=) will return if left variable is larger or equal to right variable
- IF will only run indented code once, WHILE will run indented code until condition is no longer true

```
In [13]: # Boolean variables are generally used for conditional statements such as a
# The below lines of code uses boolean variables to determine whether or no
if yes:
    print("True Statement!")

if no:
    print("Will not print")
```

True Statement!

```
In [14]: #New variable to keep track of total number of employees
dept_size = 10
```

```
In [15]: # if else statments can also be used with math or anything really (like str
# if dept_size is less than 14, display the number of employees in the depa
# Else, display a message saying the department size was exceeded

if dept_size >=0 and dept_size < 20:
    print(f"Small Department: {dept_size}")
elif dept_size < 50
    print(f"Medium Department: {dept_size}")
else:
    print(f"Large Department: {dept_size}")
```

File "<ipython-input-15-3dafd66fb8f8>", line 8

```
elif dept_size < 50
```

^

SyntaxError: invalid syntax

```
In [16]: # While loops will keep running a loop of code until the initial condition is met
# It is important to always have a breaking condition to stop the loop so it doesn't run forever
limit = 10
dept_size = 0
while dept_size < limit:
    print(dept_size)
    dept_size += 1

    if dept_size == 8:
        break # The 'break' statement in Python is used to close/end a loop
```

0  
1  
2  
3  
4  
5  
6  
7

## Lists

- Collection of items in a particular order
- They are used to store data and can be assigned to variables just like integers and strings
- Indexing (order) starts from **0**
- Accessing items in a list can be done with square brackets ([ ])
- Items can be easily added to lists using `append()` and `insert()` methods

```
In [17]: # Lists are a collection of data. List numberings always start from 0.
```

```
banks = ["RBC", "CIBC", "TD", "BMO"]
print(banks[0]) # Here the first item in the list is at index 0
print(banks[3]) # The third item in the list is at index 4

#Can use a colon to indicate range of indices
print(banks[0:3]) # From the first to third item
print(banks[:1])
print(banks[2:])

#Negative indexing goes from Right to Left, starting from -1
print(banks[-1])

#Reassign values with square brackets as well
banks[0] = "Scotiabank"
print(banks)

#Cannot do artists[4] = ""
```

```
RBC
BMO
['RBC', 'CIBC', 'TD']
['RBC']
['TD', 'BMO']
BMO
['Scotiabank', 'CIBC', 'TD', 'BMO']
```

```
In [18]: # add value to end of a list - Canadian Western Bank
# The .append() function can be used!
```

```
banks.append("CWB")
print(banks)
```

```
['Scotiabank', 'CIBC', 'TD', 'BMO', 'CWB']
```

```
In [19]: # add value to the start of a list - First Nations Bank of Canada
```

```
banks.insert(0, "FNBC")
print(banks)

# Return the length of the list
len(banks)
```

```
['FNBC', 'Scotiabank', 'CIBC', 'TD', 'BMO', 'CWB']
```

```
Out[19]: 6
```

```
In [20]: # Remove list entries
del banks[4]
print(banks)
```

```
['FNBC', 'Scotiabank', 'CIBC', 'TD', 'CWB']
```

```
In [21]: # lists can contain any type of data. A single list can be a mixture of dif

mix_list = ['Peter', 314425, True, "IT"]
print(mix_list)
print(mix_list[3])
```

```
['Peter', 314425, True, 'IT']
IT
```

```
In [22]: print(f"Name: {mix_list[0]}")
```

```
Name: Peter
```

## Dictionaries

- Collection of key-value pairs
- No positions as with lists, values stored at specific key
  - keys can be of any data type
- Accessing values in a dictionary can still be done with square brackets ([ ])
- Declared using braces ({ })

```
In [23]: # collection of "data" which is unordered, changeable, and not indexed. The
employee = { "name": "Peter", "employee_num": 314425, "department": "IT"}
# Here, 'name', 'employee_num', and 'department' are keys, and 'Peter', '31
print(employee)
```

```
{'name': 'Peter', 'employee_num': 314425, 'department': 'IT'}
```

```
In [24]: # Access key values using ['key_name']
employee["name"]
```

```
Out[24]: 'Peter'
```



```
In [25]: # Reassign a key value
employee["department"] = "Finance"
print(employee["department"])
```

Finance

```
In [26]: # Add a new key
employee["management"] = False
print(employee)
```

```
{'name': 'Peter', 'employee_num': 314425, 'department': 'Finance', 'management': False}
```

```
In [27]: # Can remove a key easily using del
# Other keys are unaffected when you use 'del' to remove a key
del employee["management"]
print(employee)
```

```
{'name': 'Peter', 'employee_num': 314425, 'department': 'Finance'}
```

```
In [28]: #Dictionary methods return iterables
print(employee.items())
print(employee.keys())
print(employee.values())

# Cannot do print(employee.keys[0]) because it is not a list
# Iterables are data objects that can be 'iterated' over, like in loops
# Iterables to be used with keyword IN ('IN' example is covered in the next
```

```
dict_items([('name', 'Peter'), ('employee_num', 314425), ('department', 'Finance')])
dict_keys(['name', 'employee_num', 'department'])
dict_values(['Peter', 314425, 'Finance'])
```

```
In [29]: # You can use dictionaries and lists in 'if' statements.

#Will look through keys by default
if "name" in employee:
    print("Yes, name is one of the keys in this dictionary")
else:
    print("no")
```

Yes, name is one of the keys in this dictionary

## For Loops

- Execute a block of code once for each item in collection (List/Dictionary)
- Declare temporary variable to iterate through collection
- Can be used in combination with IF statements

```
In [30]: #Loop through banks list
for bank in banks:
    print(bank)
```

```
FNBC
Scotiabank
CIBC
TD
CWB
```

```
In [31]: #Loop through pairs in employee dictionary
for key in employee:
    print(key)

for key, value in employee.items():
    print(f"{key}: {value}")
```

```
name
employee_num
department
name: Peter
employee_num: 314425
department: Finance
```

```
In [32]: # Use RANGE to specify a number of iterations
for i in range(len(banks)): # The len() function returns the length of the
    print(i)
```

```
0
1
2
3
4
```

## Functions

- Named blocks of code that do one specific job
- Functions are also referred to as methods
- Prevents rewriting of code that accomplishes the same task
- Keyword *def* used to declare functions
- Variables may be passed to functions

```
In [33]: # In this function 'name', 'employee_num', and 'department' are required va
def description(name, employee_num, department):
    print(f"{name} - Employee Number: {employee_num} - Dept: {department}")

description("Mike", 12210, "Marketing")
description(employee['name'], employee['employee_num'], employee['departmen
```

```
Mike - Employee Number: 12210 - Dept: Marketing
Peter - Employee Number: 314425 - Dept: Finance
```

## Classes

- Object-orientated programming approach popular and efficient
- Define classes of real-world things or situations (can be thought of as creating your own data type)
  - Attributes of various data types
  - Functions inside of a class are the same except called methods
  - Methods may be accessed using the dot operator
- Instantiate objects of your classes
- `__init__` method used to prefill attributes
- Capitalize class names

```
In [34]: class Employee():
        """A simple attempt to represent an employee."""
        def __init__(self, name, employee_num, department ):
            self.name = name
            self.employee_num = employee_num
            self.department = department

        def description(self): # Creating a function (a.k.a method) that can be
            print(f"{self.name} (employee number: {self.employee_num}) - Dept:
```

```
In [35]: employee1 = Employee("Mike", 12210, "Marketing")
employee2 = Employee("Peter", 31445, "IT")
employee1.description()
employee2.description()
```

```
Mike (employee number: 12210) - Dept: Marketing
Peter (employee number: 31445) - Dept: IT
```

## User Input

- Pauses your program and waits for the user to enter some text
- Variable used with `input()` will be a **string** even if user inputs an integer
  - Will need to make use of **type casting**.

```
In [37]: #Ask user for a name
my_age = input("Enter your age.\n")
print(f"Entered age is {my_age}")
print(f"You were born in {2020 - int(my_age)}")
```

```
Enter your age.
22
Entered age is 22
You were born in 1998
```

## Putting it all Together

- Let's take user input and create a new **Employee**
- We can then use our class methods easily!

```
In [38]: employee_input = input("Enter your name, employee number and department.\n")
name = employee_input.split(' ')[0]
employee_num = employee_input.split(' ')[1]
department = employee_input.split(' ')[2]
new_employee = Employee(name, employee_num, department)
new_employee.description()
```

```
Enter your name, employee number and department.
Peter 31445 Analyst
Peter (employee number: 31445) - Dept: Analyst
```

```
In [ ]:
```