

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]: print(type(3))
```

<class 'int'>

```
In [5]: print("This "+ str(3) +" is a string") # "3"  
        print(float(3))  
        print(bool(3))
```

```
This 3 is a string  
3.0  
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
```

```
In [8]: # Create variable `current_time` by assigning it a value of the current time. Replace the : with a .  
        current_time = 9.15
```

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 [9]: # Create two variables, price1 and price2 that have float values representing the respective price of two items  
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  
print(tot_price)
```

5.91

```
In [10]: # Define string variables name, job, and tool  
name = "Peter"  
job = "works with"  
tool = "Python"
```

```
In [11]: # We can concatenate (combine) strings together using the addition (+) symbol
employment = name + " " + job + " " + tool

#A few of the methods string come with! Check output to see how each works (definitions in handout)
print(employment.title())
print(employment.lower())
print(employment.upper())

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

```
Peter Works With Python
peter works with python
PETER WORKS WITH PYTHON
Peter works with Python
6
['Peter', 'works', 'with', 'Python']
Peter works with Python
```

```
In [12]: # 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 [13]: # 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", respectively.

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 [14]: # Boolean variables are generally used for conditional statements such as an if statement.  
# The below lines of code uses boolean variables to determine whether or not the following functions will run.  
if yes:  
    print("True Statement!")  
  
if no:  
    print("Will not print")
```

True Statement!

```
In [15]: # The below code is asking if 1 is smaller than 5, and if so, print "Employee added!"  
num_employees = 1  
  
if num_employees < 5:  
    print("Employee added!")
```

Employee added!

```
In [16]: #New variable to keep track of total number of employees  
dept_size = 10
```

```
In [17]: # if else statments can also be used with math or anything really (Like strings)!  
# if dept_size is less than 14, display the number of employees in the department. Else, display a message sayin  
# the department size was exceeded  
  
if dept_size < 14:  
    print(f"New hire. {dept_size} employees in department.")  
    dept_size += 1  
else:  
    print("Size exceeded, new offices needed!")
```

New hire. 10 employees in department.

```
In [18]: # While loops will keep running a loop of code until the intial condition is no longer true  
# It is important to always have a breaking condition to stop the loop so it does not run infinitely  
limit = 10  
  
while dept_size < limit:  
    print(dept_size)  
    dept_size += 1
```

```
In [19]: #Give dept_size a value of 0.
dept_size = 0

#WHILE Loop with condition of True will infinitely continue
while True:

    #IF dept_size reaches value of 8, break from WHILE loop
    if dept_size == 8:
        break # The 'break' statement in Python is used to close/end a loop

    #Print the dept_size and increment its value
    print(dept_size)
    dept_size += 1
```

```
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 [20]: *# 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 [21]: # add value to end of a list - Canadian Western Bank
banks.append("CWB") # The .append() function can be used to add an item to the end of a list
print(banks)

# 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)

del banks[4]

print(banks)
```

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

```
In [22]: #Remove and return last value of list
last_bank = banks.pop() # The .pop() function is used to remove and return the last item in a list
print(f"{last_bank} has been removed")
print(banks)
```

```
CWB has been removed
['FNBC', 'Scotiabank', 'CIBC', 'TD']
```

In [23]: *# lists can contain any type of data. A single list can be a mixture of different data types*

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

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

In [24]: `print(f"{mix_list[0]} - Employee Number:{mix_list[1]} - Dept: {mix_list[3]}")`

```
Peter - Employee Number:314425 - Dept: IT
```

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 [25]: *# collection of "data" which is unordered, changeable, and not indexed. They have keys and values.*
`employee = { "name": "Peter", "employee_num": 314425, "department": "IT"}`
Here, 'name', 'employee_num', and 'department' are keys, and 'Peter', '314425', and 'IT' are the corresponding
`print(employee)`

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

```
In [26]: # Access key values using ['key_name']  
employee["name"]
```

Out[26]: 'Peter'

```
In [27]: # Reassign a key value  
employee["department"] = "Finance"  
print(employee["department"])
```

Finance

```
In [28]: # Add a new key  
employee["management"] = False  
print(employee)
```

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

```
In [29]: # Can remove a key easily using del  
del employee["management"]  
print(employee)  
  
# Other keys are unaffected when you use 'del' to remove a key  
print(employee['name'])
```

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

```
In [30]: #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 cell blocks and section with 'For' L
```

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

```
In [31]: # 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

```
In [32]: # Use values() to check in values of dictionary
if "Peter" in employee.values():
    print("Yes, Peter is one of the values in this dictionary")
else:
    print("no")
```

Yes, Peter is one of the values in this dictionary

```
In [33]: # IN can be used with lists very easily too!
if "IT" in mix_list:
    print("You should try Python!")
```

You should try Python!

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

FNBC
Scotiabank
CIBC
TD

```
In [35]: #Loop through pairs in employee dictionary
for key, value in employee.items():
    print(f"{key}: {value}")
```

name: Peter
employee_num: 314425
department: Finance

```
In [36]: # Use RANGE to specify a number of iterations
for i in range(len(banks)): # The len() function returns the length of the previously defined 'banks' list
    print(i)
```

```
0
1
2
3
```

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 [37]: # A function is a block of organized, reusable code that is used to perform a single, related action.
def greeting(): # The bracket is empty as no values are required to be passed in this function
    print("Hi!")

greeting()
```

```
Hi!
```

```
In [38]: # In this function 'name', 'employee_num', and 'department' are required values that must be passed to the function
def description(name, employee_num, department):
    print(f"{name} - Employee Number: {employee_num} - Dept: {department}")

description("Mike", 12210, "Marketing")
```

Mike - Employee Number: 12210 - Dept: Marketing

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

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



```
In [40]: employee = Employee("Mike", 12210, "Marketing")
employee.description()
```

Marketing (employee number: Mike) - Dept: 12210

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 [41]: #Ask user for a name
my_name = input("Enter your age.\n")
print(f"Entered age is {my_name}")
```

Enter your age.

23

Entered age is 23

```
In [42]: #Will always be treated as a string
type(my_name)
```

Out[42]: str

```
In [ ]:
```

