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

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 [6]:
```
#The type function will display the type of whatever is given to it
type("Hello!")
```

Out[6]:
```
str
```

In [10]:
```
type(True)
```

Out[10]:
```
bool
```

In [11]:
```
type(3.14)
```

Out[11]:
```
float
```

In [5]:
```
print(type(3))
```

```
<class 'int'>
```

In [1]:
```
#The code to output the type of data for (3) has already been filled in for you as an

print("This "+ str(3) +" is a string") # "3"
print(float(3))
print(bool(3))
```

```
<class 'int'>
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.

In [2]:
```
# In the code below, the variable `hours_worked` has been assigned an integer value o
hours_worked = 10
```

In [4]:
```
print(hours_worked)
```

```
10
```

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

Math Operators

- Addition, Subtration, 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 [11]:
```
# Create two variables, price1 and price2 that have float values representing the res
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 [27]:
```
#Define string variables name and employed
name = "Peter"
job = "works with"
tool = "Python"
```

In [31]:
```
#We can concatenate strings together using +
employment = name + " " + job + " " + tool

#A few of the methods string come with! Check output to see how each works (definitio
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 [32]:
```
#A few ways to combine strings and variables
#With F strings, variables go directly into the string! Even methods!
print(f"{name} works with {tool.upper()}")
```

```
Peter works with PYTHON
```

In [35]:
```
#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 [36]:
```
#Boolean variables are generally used for conditional statements such as an if statem
#The below lines of code uses boolean variables to determine whether or not the follo
if yes:
    print("True Statement!")

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

```
True Statement!
```

In [48]:
```
# 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 [40]:
```
#New variable to keep track of total number of employees
dept_size = 10
```

In [45]:
```
# if else statments can also be used with math or anything really!

if dept_size < 14:
    print(f"New hire. {dept_size} employees in department.")
    dept_size += 1
else:
    print("Size exceeded, new offices needed!")
```

```
Size exceeded, new offices needed!
```

In [51]:
```
limit = 10

while dept_size < limit:
    print(dept_size)
    dept_size += 1
```

```
0
1
2
3
4
5
6
7
8
9
```

In [52]:
```
#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

    #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
- 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 [54]:
```
#Lists are a collection of data. The lists start at 0.

banks = ["RBC", "CIBC", "TD", "BMO"]
print(banks[0])
print(banks[3])

#Can use a colon to indicate range of indices
print(banks[0:3])
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 [55]:
```
# add value to end of a list - Canadian Western Bank
banks.append("CWB")
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 [56]:
```
#Remove and return last value of list
last_bank = banks.pop()
print(f"{last_bank} has been removed")
print(banks)
```

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

In [57]:
```
# lists can contain any type of data
mix_list = ['Peter', 314425, True, "IT"]
print(mix_list)
print(mix_list[3])
```

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

In [61]:
```
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 [62]:
```
# collection of "data" which is unordered, changeable and indexed. They have keys and
employee = { "name": "Peter", "employee_num": 314425, "department": "IT"}
print(employee)
```

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

In [64]:
```
#Access key values using ['key_name']
employee["name"]
```

Out[64]:
```
'Peter'
```

In [72]:
```
#Reassign a key value
employee["department"] = "Finance"
print(employee["department"])
```

```
Finance
```

In [75]:
```
#Add a key
employee["management"] = False
print(employee)
```

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

In [76]:
```
#Can remove a key eaasily using del
del employee["management"]
print(employee)

#Other keys unaffected
print(employee['name'])
```

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

In [77]:
```
#Dictionary methods return iterables
print(employee.items())
print(employee.keys())
print(employee.values())

# Cannot do print(person.keys[0]) because it is not a list

# Iterables to be used with keyword IN
```

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

In [84]:
```
# You can use dictionaries and lists in if statments.

#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 [85]:
```
# 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 [87]:
```
# 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 [90]:
```
#Loop through banks list
for bank in banks:
    print(bank)
```

```
FNBC
Scotiabank
CIBC
TD
```

In [91]:
```
#Loop through pairs in employee dictionary
for key, value in employee.items():
    print(f"{key}: {value}")
```

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

In [92]:
```
#Use RANGE to specify a number of iterations
for i in range(len(banks)):
    print(i)
```

```
0
1
2
3
```

Functions

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

In [93]:
```
# A function is a block of organized, reusable code that is used to perform a single,
def greeting():
    print("Hi!")

greeting()
```

```
Hi!
```

In [94]:
```
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
  - Attributes of various data types
  - Functions inside of a class are the same except called methods
  - Methods may be accessed using the dot operator
- Instanciate objects of your classes
- __init()__ method used to prefill attributes
- Capitalize class names

In [96]:
```
class Employee():
    """A simple attempt to represent am employee."""
    def __init__(self, employee_num, department, name):
        self.employee_num = employee_num
        self.department = department
        self.name = name

    def description(self):
        print(f"{self.name} (employee number: {self.employee_num}) - Dept: {self.depa
```

In [100]:
```
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 [102]:
```
#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 [103]:
```
#Will always be treated as a string
type(my_name)
```

Out[103]:
```
str
```