PythonTrack

Python Basics

Why Python?

Python - Why?

- It is easier to implement complex algorithms and data structures.
- To make sure everyone has equal understanding of programming.
- We use Python for lectures.

Syntax

- No semicolons, yay?
- White Spaces matter.
- Similar to the English language.

```java
// Java
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

```python
# Python
print('Hello, world!')
```

Africa To Silicon Valley

Syntax - Indentation

- In Python, unlike other programming languages, indentation serves a crucial purpose beyond just readability.
- Python uses indentation as a way to indicate and define blocks of code.

Variables

- Variables are used to store and manipulate data.
- Python has no command for declaring a variable.
- They are created by assigning a value to a name.
- Python has dynamic typing.

```
x = 4 # x is of type int
x = "A2SV"    # x is now of type
str
```

Variables- Names

- Must start with a letter or the underscore character
- Can not start with a number
- Can only contain alphanumeric characters and underscores (A-z, 0-9, and _)
- Case-sensitive (age, Age and AGE are three different variables)
- Can not be a keyword (if, while, and, for, …)
- snake_case

Data Types in Python

- Data types define the kind of data that can be stored and manipulated in a program.
- Common Built-in Data Types:
  - Boolean (bool)
  - Integer (int)
  - Float(float)
  - String (str)

Boolean

- In programming you often need to know if an expression is **True** or **False**.
- You can evaluate any expression in Python, and get one of two answers, **True** or **False**.
- When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
10 > 9  # True
10 == 9 # False
10 < 9  # False
```

Boolean- Evaluation

- The **bool()** function allows you to evaluate any value, and give you **True** or **False** in return,
- In Python values with content are True:
  - Any string is **True**, except empty strings.
  - Any number is **True**, except **0**.
  - Any list, tuple, set, and dictionary are **True** , except empty ones.

Numeric data types

- Integer:
  - Represent integer numbers without any decimal points
  - Can be positive or negative numbers, or zero.
  - Examples of integers are: **x = -5**, **x = 0**, **x = 10**, **x = 100**

Numeric data types

- Float:
  - Represent decimal numbers or numbers with a fractional part
  - They are written with a decimal point, even if the fractional part is zero
  - Examples of floating-point numbers are: **x =-2.5**, **x = 3.14**, **x = 1.0**

Operators

- Operators are used to perform operations on variables and values.
- In the example below, we use the **+** operator to add together two values:

```
print(10 + 5)
```

Operators

- Python divides the operators in the following groups:
  - Arithmetic operators
  - Assignment operators
  - Comparison operators
  - Logical operators
  - Identity operators
  - Membership operators

Operators- Arithmetic

- Arithmetic operators are used with numeric values to perform common mathematical operations.

Operators- Arithmetic

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Operators- Precedence

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x` `-x` `~x`	Unary plus, unary minus, and bitwise NOT
`*` `/` `//` `%`	Multiplication, division, floor division, and modulus
`+` `-`	Addition and subtraction

Operators- Comparison

- Comparison Operators are used to compare two values.

Operators- Comparison

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Practice Problems

- Arithmetic Operators
- Division
- Convert the Temperature
- Palindrome Number

Operators- Logical

- Logical operators are used to combine conditional statements.

Operators- Logical

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Operators- Identity

- Identity Operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

Operators- Identity

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Operators- Membership

- Membership Operators are used to test if a sequence is presented in an object.

Operators- Membership

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Strings

- Strings in python are surrounded by either single quotation marks, or double quotation marks.
- **'hello'** is the same as **"hello"**.
- You can display a string literal with the **print()** function:
- String in python are immutable.
- You can assign a multiline string to a variable by using three quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
```

Strings - Slicing Strings

- You can return a range of characters by using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string. We can also specify step as a third parameter (optional).

```
b = "Hello, World"
print(b[2])
print(b[-3])
print(b[2:5])
print(b[:5])
print(b[2:])
print(b[5:2:-1])
print(b[::-1])
print(b[::2])
```

Strings - String Concatenation

- To concatenate, or combine, two strings you can use the + operator.

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

Strings - Formatting

- To format strings in python we can use f-strings.

```
a = 1
b = "hello"
print(f"{b} {a} {a + 2}") # hello 1 3
```

Strings - Substring search

- In python we can use the **”in”** operator to check if a string occurs as a substring of another string

```
print("Hello" in "Hello world")
```

Variables - Casting

- Variable casting allows converting a value from one data type to another.
- Python provides built-in functions for explicit casting, such as **'str()'**, **'int()'**, and **'float()'**.

```
y = int(3.0)    # y will be 3
z = float(3)    # z will be 3.0
```

Check point

- What will be the output of the following statements?

```
s = "Hello, World!"

print(s[5]) # ?
print(s[-2]) # ?
print(s[1:]) # ?
print(s[-2:]) # ?
```

Practice Problems

- sWAP cASE
- String Split and Join
- What's Your Name?

Conditionals

If statement

- We use **if** statement to write a single alternative decision structure.
- Here is the general format of the if statement:

if condition:
    statement
    statement

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

Elif

- The **elif** keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

```
a = 33
b = 33
if b > 33:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

Else

- The **else** keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

Conditionals

- You can have **if** statements inside **if** statements, this is called nested **if** statements.
- We can use logical operators to combine conditional statements.

```
a = 200
b = 33
c = 500
if a > b and c > a:
    print("Both conditions are True")
```

Loops

While Loop

- With the **while** loop we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

For Loop

- A **for** loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

Nested Loops

- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop":

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
```

The range() Function

- The **range()** function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6): # init(d) 0 final 6 step(d) 1
  print(x)

for x in range(2, 6): # init 2 final 6 step(d) 1
  print(x)

for x in range(2, 10, 2): # init 2 final 10 step 2
  print(x)
```

Continue Statement

- With the **continue** statement we can stop the current iteration, and continue with the next.

```
i = 0
while i < 9:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
for i in range(9):
  if i == 3:
    continue
  print(i)
```

Break Statement

- With the **break** statement we can stop the loop even if the while condition is true:

```
for i in range(9):
  if i > 3:
    break
  print(i)
```

```
i = 1
while i < 9:
  print(i)
  if i == 3:
    break
  i += 1
```

For - Else

- With the **else** statement we can check if the loop finishes it's iteration or not.

```
fruits = ["apple", "banana", "mango",
"orange"]


for fruit in fruits:
    if fruit == "pineapple":
        print("Found pineapple!")
        break
else:
    print("Pineapple not found in the
list.")
```

Check point

- What is the output of the following nested Loop?

```
for num in range(10,14):
    for i in range(2, num):
        if num % i == 1:
            print(num)
            break
```

A) 10
11
12
13

B) 11
13

Functions

Functions

- A function is a reusable block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

```
def my_function():
  print("Hello from a function")

my_function()
```

Arguments

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses.
- You can add as many arguments as you want, just separate them with a comma.

```
def my_function(fname): fname[0]
    = “anna” print(fname[0] + "
    Refsnes")
data = ["Emil"]
my_function(data)
print(data[0])
```

Return Values

- To let a function **return** a value, use the return statement:

```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Lambda

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.
- Syntax:

  lambda arguments : expression

```
x = lambda a : a + 10
print(x(5))

x = lambda a, b : a * b
print(x(5, 6))
```

Lists

What are lists?

- Lists are fundamental data structures in Python used to store collections of data.
- They can hold items of any data type, including numbers, strings, and even other lists.
- Lists are ordered, changeable, and allow duplicate values.

Creating lists

- Lists can be created using square **brackets []** and separating items with commas.
- The **list()** constructor can also be used to create lists.

```
# Creating a list using square brackets
fruits = ["apple", "banana", "cherry"]

# Convert iterables to list using the list() constructor
numbers = list((1, 2, 3, 4, 5))
```

List data types

List items can be of any data type

- `list1 = ["apple", "banana", "cherry"]`
- `list2 = [1, 5, 7, 9, 3]`
- `list3 = [True, False, False]`
- `list4 = ["abc", 34, True, 40, "male"]`

Accessing List Items

- List items are accessed using their index number, starting from 0.
- Negative indexing can be used to access items from the end of the list.

```
nums = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
# Accessing the first item

nums[0]  10        #
#
   Accessing the last item
                   #
nums[-1] 19
```

Slicing Lists

- Slicing allows extracting a sublist from a list.
- Slicing uses the **colon (:)** to separate start and end indices (inclusive).

```
nums = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
#     Extracting a sublist from index 2 to index 4
                          #
nums[2:5]     [12, 13, 14]

nums[-4 : -1]??
```

Modifying Lists

- Lists are mutable, allowing you to change their contents.
- You can modify items using their index or extend the list using **append**() and **insert**().
- You can also remove items using **remove**() and **pop**().

Example

s

```
fruits = ["apple", "banana", "cherry"]
# Changing the first item
fruits[0] = "orange"  #   fruits = ["orange", "banana", "cherry"]
# Adding an item to the end
fruits.append("mango")   # fruits = ["orange", "banana", "cherry", "mango"]
# Removing an item by value
fruits.remove("cherry")  # fruits = ["orange", "banana", "mango"]
# Removing the last item
removed_item = fruits.pop() # removed_item = "mango", fruits =
["orange", "banana"]
```

Common List Operations

- Checking if an item exists: in keyword
- Sorting a list: **sort**() method
- **sorted** (nums , key = myFunction ( ), reverse = True/False)
- Reversing a list: **reverse**() method

Examples

```
fruits = ["orange", "banana"]
# Checking if "apple" exists in the list
if "apple" in fruits:
    print("Yes, apple is in the list")
# Sorting the list in ascending order
fruits.sort()  # fruits = ["banana", "orange"]
```

Reversing the sorted list

```
fruits.reverse()  # fruits = ["orange", "banana"]
```

Combining Lists

- Concatenating lists using the + operator or extend() method
- Adding items from one list to another individually

Examples

```
numbers = [1, 2, 3]

fruits = ["orange", "banana"]

# Concatenating lists using '+' operator

new_list = fruits + numbers   n#w_list = ["orange", "banana", 1, 2, 3]
   Extending a list using extend() method fruits.extend(numbers)
#
fruits = ["orange", "banana", 1, 2, 3]
                                    #
```

Traversing Lists

- Iterating through lists using for loops
- Accessing both index and value using enumerate() function

- ```
  for index in range(len(nums)):
  print(nums[index])
  ```

- ```
  for num in nums:
      print(num)
  ```

- ```
  for index, num in enumerate(nums):
  print(index, num)
  ```
  ```

List Comprehension

- Creating new lists based on existing lists
- Using expressions and conditions to filter and transform list elements

```
# Creating a list of even numbers from a list of numbers
numbers = [1, 2, 3, 4, 5]
even_numbers = [num for num in numbers if num % 2 == 0]
# even_numbers = [2, 4]
```

Why List Comprehension?

```
my_list = [[0]] * 5

my_list = ? # [[0], [0], [0], [0], [0]]
my_list[0][0] = 1

my_list = ?
```

Why List Comprehension?

```
my_list = [[0]] * 5
my_list[0][0] = 1
my_list = [[1], [1], [1], [1], [1]] #Why?
```

Other List Methods

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Tuples

What are Tuples?

- A tuple is a collection which is ordered, allows duplicates and is **unchangeable**. Tuples are also known as Immutable Lists.

- Tuples are written with round brackets.
  - fruits = ("apple", "banana", "cherry")
  - fruit = ("apple",)

Creating Tuples

- Tuples are written with round brackets **()**.
- This is called 'packing' a tuple.

```
fruits = ("apple", "banana", "cherry")
fruit = ("apple",)#or just () to create an empty one
```

- The **tuple()** constructor:

```
fruits = tuple(["apple", "banana", "cherry"])
numbers = tuple()
```

Unpacking tuples

- In Python, we are also allowed to extract the values back into variables. This is called "unpacking".

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

fruits = ("apple", "banana", "cherry", “oranges”, “pineapples”)

green, yellow, *red = fruits
```

Unpacking tuples

- In Python, we are also allowed to extract the values back into variables. This is called "unpacking".

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits


fruits = ("apple", "banana", "cherry", "oranges", "pineapples")
(green, yellow, *red) = fruits      # red = ["cherry","oranges", "pineapples"]
                                    #
```

Tuples

- Is it possible to
  - add an element to a Tuple? How?
  - delete an element?
  - join two tuples?

Tuple Similarities with List

- Similar data types
- Slicing and Indexing
- Similar Iteration

Tuple Methods

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Classes

Classes and Objects

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Class - Example

```
class MyClass:
    def __init__(self):
        self.x = 5


p1 = MyClass()
print(p1.x) # 5
```

The __init__() function

- Classes have a function called **__init__()**, which is always executed when the class is being initiated.

- Used to assign values to object properties

The __init__() function (Continued)

```
class Person:
  def __init__(self, name, age):

    self.name = name


    self.age = age

p1 = Person("John", 36)
```

- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc() # Hello my name is John
```

Collections & Containers

Collections and Containers

- The collections module in Python provides different types of containers.
- A container is an object that is used to store different objects and provide a way to access the contained objects and iterate over them.
- Some of the built-in containers are Tuple, List, Dictionary, etc.

Collections - Example

- Commonly used containers provided by the collections module:

Defaultdict

```
from collections import defaultdict
```

Counter

```
from collections import Counter
```

Deque

```
from collections import deque
```

defaultdict

- It is used to provide some default values for the key that does not exist and without raising a KeyError.

- The default value is specified when creating the defaultdict object using the default_factory parameter.

defaultdict - Example

```
from collections import defaultdict

d = defaultdict(int)

L = [1, 2, 3, 4, 2, 4, 1, 2]

for i in L:

    d[i] += 1

print(d)
```

Counter

- It is used to keep the count of the elements in an iterable in the form of an unordered dictionary
- The key represents the element in the iterable
- The value represents the count of that element in the iterable.

Counter - Example

```
from collections import Counter


print(Counter(['B','B','A','B','C','A','B','B','A','C']))

?
```

Counter - Example

```
from collections import Counter


print(Counter(['B','B','A','B','C','A','B','B','A','C']))

# {'B': 5, 'A' : 3, 'C' : 2}
```

Counter - Example

```
from collections import Counter


print(Counter("ABCDEFGABCDE"))

?
```

Counter - Example

```
from collections import Counter


print(Counter("ABCDEFGABCDE"))

# {'A': 2, 'B': 2, 'C': 2, 'D': 2, 'E': 2, 'F': 1, 'G': 1}
```

Deque (Doubly Ended Queue)

- Deque (Doubly Ended Queue) is the optimized list for quicker append and pop operations from both sides of the container.
- It provides O(1) time complexity for append and pop operations as compared to list with O(n) time complexity.

deque - Example

```
from collections import deque
de = deque([1,2,3])

de.append(4)

print(de)    # ?


de.popleft()

print(de) # ?
```

deque - Example

```
from collections import deque
de = deque([1,2,3])

de.append(4)


print(de)    # [1, 2, 3, 4]

de.popleft()

print(de) # [2, 3, 4]
```

Python Built-In Functions

Built-in Functions

Function	Use	Example
`all()`	Returns True if all items in an iterable object are true	`x = all([True, True, True])`
`any()`	Returns True if any item in an iterable object is true	`x = any([False, True, False])`
`chr()`	Returns a character from the specified Unicode code	`x = chr(97)`
`ord()`	Get the ASCII value of a character	`x = ord("h")`

Built-in Functions (Continued)

Function	Use	Example
`eval()`	Evaluates and executes an expression	`x = 'print(55)'` `eval(x)`
`type()`	Returns the type of an object	`b = "Hello World"` `y = type(b)`

Built-in Functions (Continued)

Function	Use	Example
`map()`	Apply a function to elements in an iterable	`func = lambda a: len(a)` `x = map(func,('a','ba','che'))`
`filter()`	Filter elements in an iterable based on a function	`func = lambda a: a%2` `x = filter(func, [1,2,3,4])`
`zip()`	Combine elements from multiple iterables into tuples	`x = zip(['a','b','c','d'],` `[1,2,3,4])`

Built-in Math Functions

Function	Use	Example
`min()`	Find the lowest in an iterator	`x = min(5, 10, 25)`
`max()`	Find the highest in an iterator	`y = max(5, 10, 25)`

`sum()`	Returns the sum of all elements in a list or iterable	`total = sum(5, 10, 25)`
		`nums = [5, 10, 25]`
		`total = sum(nums)`

`abs()`	Returns the absolute (positive) value of the specified number	`x = abs(-7.25)`

Built-in Math Functions (Continued)

Function	Use	Example
`sqrt()`	Returns the square root of a number:	`import math` `x = math.sqrt(64)`
`ceil()`	Rounds a number upwards to its nearest integer	`import math` `x = math.ceil(1.4)`
`floor()`	Rounds a number downwards to its nearest integer	`import math` `x = math.floor(1.4)`

Built-in String Functions

Function	Use	Example
`len()`	Returns the length of a string	`s = "Hello world!"` `s_length = len(s)`
`split()`	Splits a string into a list based on a delimiter	`s = "a,b, c"` `result = s.split(",")`
`join()`	Concatenates elements of a list into a single string	`words = ['Hello','world']` `result = " ".join(words)`
`strip()`	Removes leading and trailing whitespaces	`s = "   Hello, World!\t\n "` `s.strip()`
`replace()`	Replaces occurrences of a substring with another	`s = "Hello, a2sv! How is everyone in a2sv?"` `s.replace("a2sv", "A2SV")`

Common Pitfalls

- Iteratively concatenating strings using the + operator. It can be inefficient due to string immutability. Use "".join() for efficient string concatenation.

- Forgetting to include self as the first parameter in instance methods of a class.

Best Practices

- Explore the collections module for specialized data structures (e.g., Counter, defaultdict) instead of reinventing the wheel.
- Look for opportunities to use built-in functions like map, filter, and list comprehensions to avoid unnecessary loops.
- When iterating over multiple iterables in parallel, use zip for clean and efficient code.
- Initialize all necessary attributes in the `__init__` method to ensure a well-defined object state.

Practice Problems

Two Sum
Contains duplicate
Majority Element
Majority Element ii
List Comprehension
Runner-up Score
Nested Lists
Lists

Smallest even multiple
Weird
Powers
Mod Power
Longest common prefix
More exercise

Quote of the day

"The difference between ordinary and extraordinary is that little extra. Add a touch of **class** to everything you do."

– Jimmy Johnson

Africa To Silicon Valley