Introduction To Python Programming

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- **Data types**
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Data types

Name	Type	Description
Integers	int	Whole numbers, such as: **3 300 200**
Floating point	float	Numbers with a decimal point: **2.3 4.6 100.0**
Strings	str	Ordered sequence of characters: **"hello" 'Sammy' "2000" "楽しい"**
Lists	list	Ordered sequence of objects: **[10,"hello",200.3]**
Dictionaries	dict	Unordered Key:Value pairs: **{"mykey" : "value" , "name" : "Frankie"}**
Tuples	tup	Ordered immutable sequence of objects: **(10,"hello",200.3)**
Sets	set	Unordered collection of unique objects: **{"a","b"}**
Booleans	bool	Logical value indicating **True** or **False**

Data types

Python has a lot of built-in data types
Each variable has a data type based on the value assigned to it
You can check a variable's type using type() function

```python
# use type() function to check what is the type of a variable
x = 5
y = "python is awesome"
z = [1, 2, 3]

print(type(x)) # int
print(type(y)) # str
print(type(z)) # list
```

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- **Numbers & Math**
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Numbers & Math

There are two types of numbers in Python: integers (int) and floating point (float)

Scientific notation is used to describe very large numbers
2.5e2 = 2.5 x 10 ^ 2 = 250

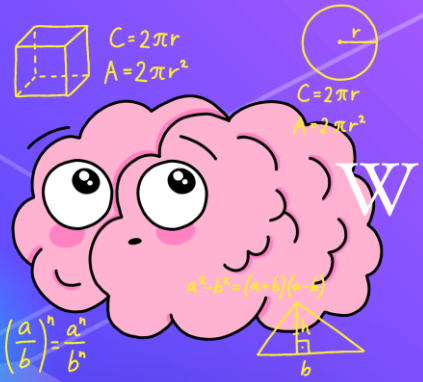
```
# this is integer values (int)
5
1000
-5000
0


# this is float values (float)
5.25
1000.75
-5000.3
5.0
2.5e2   # 2.5*(10**2)
2.5e+2  # 2.5*(10**2)
2.5e-2  # 2.5*(10**-2)
```

Numbers & **Math**

Python supports a number of arithmetic operations

Operator	Description	Syntax
+	Addition: adds two operands	x + y
-	Subtraction: subtracts two operands	x - y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when first operand is divided by the second	x % y
**	Power : Returns first raised to power second	x ** y



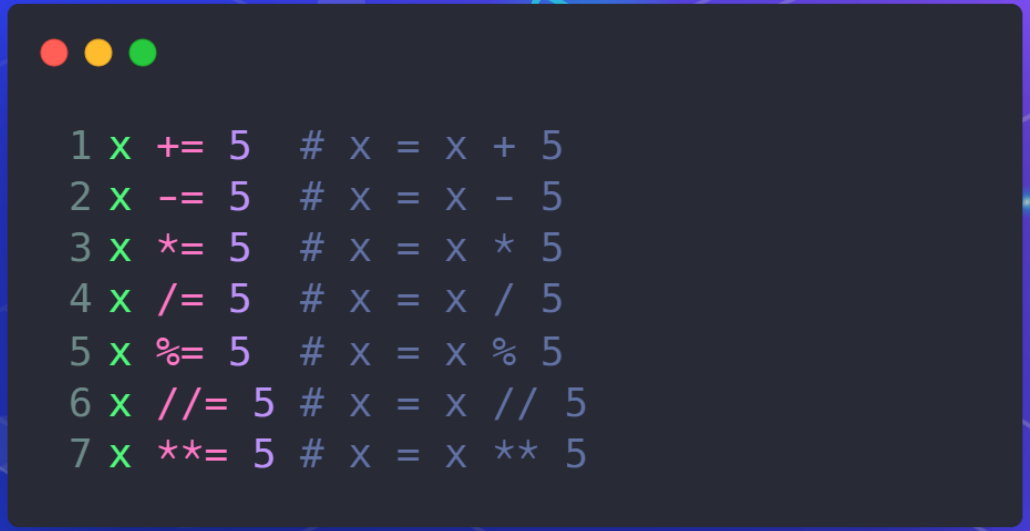

```
1 3 + 5       # result is 8
2 10 - 7      # result is 3
3 2 * 5       # result is 10
4 15 / 5      # result is 3
5 3 / 2       # result is 1.5
6 3 // 2      # result is 1
7 32 % 3      # result is 2
8 2 ** 3      # result is 8
9 4 ** 0.5    # result is 2
```

Numbers & **Math**

You can combine an operator with the assignment expression (=) to update a variable's value

For example, '+=' increments the variable on the left hand side by the value on the right hand side

And '*=' multiplies the variable on the left hand side by the value on the right hand side



```
x += 5   # x = x + 5
x -= 5   # x = x - 5
x *= 5   # x = x * 5
x /= 5   # x = x / 5
x %= 5   # x = x % 5
x //= 5  # x = x // 5
x **= 5  # x = x ** 5
```

Quiz Time!

Q1. 3 + 3 * 3 + 3

- A. 36
- B. 15
- C. 27

Q2. (3 + 3) * (3 + 3)

- A. 36
- B. 15
- C. 27

Q3. 4 // 2 + 5 * (1+2)

- A. 21
- B. 9
- C. 17

Q4. 3 + 3 / 3 - 3

- A. 1
- B. ZeroDivisionError
- C. 0

Q5. (3 + 3) / (3 – 3)

- A. 1
- B. ZeroDivisionError
- C. 0

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- **Boolean & Comparison and Logic**
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Boolean & Comparison and Logic

Boolean algebra is the type of algebra performed on Boolean values only. Those are, True and False (0 and 1)

```
1 is_online = True
2
3 has_dog = False
```

Boolean & **Comparison** and Logic

Comparisons yield a Boolean value (Assume a = 10 & b = 20)

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

```
1 5 == 5      # result is True
2 5 != 5      # result is False
3 10 > 7      # result is True
4 2 >= 5      # result is False
5 15 < 5      # result is False
6 3 <= 3      # result is True
```

Boolean & Comparison and **Logic**

OPERATOR	DESCRIPTION	SYNTAX
and	Like multiplication: 1 x 0 = 0 Logical AND: True if both the operands are true	x and y
or	Like addition: 1 + 0 = 1 Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x

```
1 # AND
2 1 < 2 and 2 < 3          # Result is True
3 1 != 1 and 2 < 3         # Result is False
4 1 != 1 and 2 > 3         # Result is False
5
6
7 # OR
8 1 < 2 or 2 < 3           # Result is True
9 1 != 1 or 2 < 3          # Result is True
10 1 != 1 or 2 > 3         # Result is False
11
12
13 # NOT
14 not 1 == 1              # Result is False
15 not 1 > 10              # Result is True
```

Quiz Time!

Q1. 5 > 10 and 3 > 2

- A. True
- B. False

Q2. 5 > 10 or 3 > 2

- A. True
- B. False

Q3. -10 < 3 and 0 < 2

- A. True
- B. False

Q4. (not 1 == 10) and 2 >= 2

- A. True
- B. False

Q5. 0 > -1 and (1 == 2 and (not 1 != 2))

- A. True
- B. False

Introduction to Python programming Course Outline

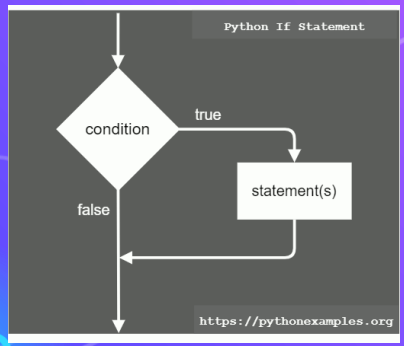
- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- **If Conditions**
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

If Conditions

Previously, when we run our code, it would execute all statements in order

It's time to apply flow control

If statements allow us to control the flow of the code based on a certain condition



```
person = 'George'

if person == 'Sammy':
    print('Welcome Sammy!')
elif person =='George':
    print('Welcome George!')
else:
    print("Welcome, what's your name?")

# Welcome George!
```

Quiz Time!

What will be the output of the following if statements:

Q1

```
number1 = 5
number2 = 1
if (number1 + number2) < 3:
  print("Sloths")
else:
  print("Cats")
```

- A. Sloths
- B. Cats
- C. No print

Q2

```
num = 15
if num / 7 == 7:
  print("It divides by 7")
elif num / 3 == 5:
  print("It divides by 3")
else:
  print("Doesn't divide")
```

- A. It divides by 7
- B. It divides by 3
- C. Doesn’t divide

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- **For & While Loops**
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

For Loops

Loops are used to repeat a certain block of code

For loops can repeat the code for a known number of times

They should be used when we know how many times we need the code to repeat

```python
for i in range(10):
  print("i =", i)
  # i = 0
  # i = 1
  # i = 2
  # i = 3
  # i = 4
  # i = 5
  # i = 6
  # i = 7
  # i = 8
  # i = 9
```

```python
# range(start, stop, step)
for i in range(2, 10, 2):
  print("i =", i)
  # i = 2
  # i = 4
  # i = 6
  # i = 8
```

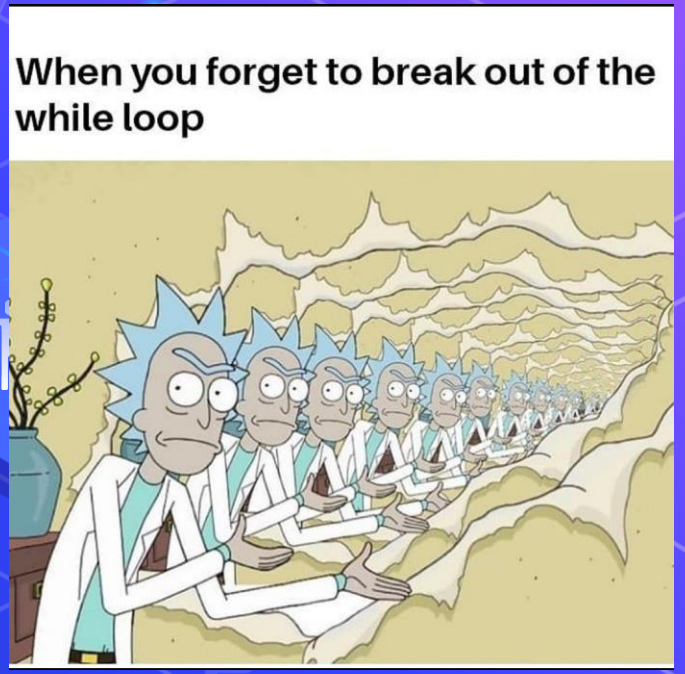
While Loops

While loops keep repeating the code while a given condition is True

It will break out of the loop once the condition turns to False

```
x = 20
while x > 0:
  print("x =", x)
  x -= 5 # Update

  # x = 20
  # x = 15
  # x = 10
  # x = 5
```



Make sure your condition will turn False after a while, or you're getting stuck with an infinite loop!

Quiz Time!

What will be the output of the following statements:

Q1

```
for num in range(2,-5,-1):
    print(num)
```

- A. 2, 1, 0
- B. 2, 1, 0, -1, -2, -3, -4, -5
- C. 2, 1, 0, -1, -2, -3, -4

Q2

```
counter = 1
sum = 0
while counter ⩽ 6:
    sum = sum + counter
    counter = counter + 2
print(sum)
```

- A. 12
- B. 9
- C. 7

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- **Strings**
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Strings

Strings are ordered sequences of characters (alphabets, numbers, etc.)
Individual characters can be accessed using indexing

```
greeting = "Hello World"
greeting = 'Hello World'

print(greeting[0])  # H
print(greeting[2])  # l
print(greeting[-1]) # d
```

String Formatting

A way to inject a variable into a string for convenience

Add an ‘f’ before the string to add formatting,
then add variables using braces {}


```
x = 10
y = x / 2

print(f"Value of x = {x} and value of y = {y}")
# Value of x = 10 and value of y = 5.0
```

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- **Lists**
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Lists

Lists are the most common data structure in Python

You can store multiple values (elements) inside a single variable

Unlike other programming languages, Python lists can have elements of different types

```
my_list = ['A string', 23, 100.232 , 'p', True]

print(my_list[0])   # 'A string'
print(my_list[1])   # 23
print(my_list[2])   # 100.232
print(my_list[3])   # 'p'
print(my_list[4])   # True
```

Lists

List elements can be lists too!

```
my_list = [[1,2,3], [4,5,6], [7,[8,9]]]

print(my_list[0])        # [1,2,3]
print(my_list[0][1])     # 2
print(my_list[2][1][1]) # 9
```

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- **Tuples**
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Tuples (faster and immutable lists)

Used when you have immutable values and need faster processing on them

```python
my_tuple = ('A string', 23, 100.232 , 'p', True)

print(my_tuple[0])  # 'A string'
print(my_tuple[1])  # 23
print(my_tuple[2])  # 100.232
print(my_tuple[3])  # 'p'
print(my_tuple[4])  # True
```

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- **Sets**
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Sets (unique lists)

Used for intersections & union operations

```python
my_list = [1,1,2,2,3,4,5,6,1,1]

my_set = set(my_list)
print(my_set)   # {1, 2, 3, 4, 5, 6}
```

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- **Dictionaries**
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Dictionaries

Just like a human dictionary, Python dictionary are data structures that store data in key – value pairs

```
store = {'apples': 10, 'oranges': 20}

print(store['apples'])  # result is 10
print(store['oranges']) # result is 20
```

Dictionaries

Some of the useful dictionary functions:

dict.get(‘key’) – looks for the key in the dictionary and returns value if found, returns default value if not found

dict.keys() – returns dictionary keys
dict.values() – returns dictionary values

```
store = {'apples':10, 'oranges':20}

print(store['grapes'])      # KeyError
print(store.get('grapes'))  # '' - default value
print(list(store.keys()))   # ['apples', 'oranges']
print(list(store.values())) # [10, 20]
```

More functions:
https://www.w3schools.com/python/python_ref_dictionary.asp

Quiz Time!

Q1. list1 = ['physics', 'chemistry', 1997, 2000]
print(list1[1][-1])

- A. p
- B. c
- C. y
- D. Error

Q2. list1 = [1, 2, 3, [1, 2], (1, 2, 3)]
print(len(list1))

- A. 8
- B. 5
- C. 6

Q3. list1 = [1998, 2002]
list2 = [2014, 2016]
print(list2 + list1)

- A. [4012, 4018]
- B. [2014, 2016, 1998, 2002]
- C. [1998, 2002, 2014, 2016]

Q4. name = "Data Science"
print(name[:4] + "Analysis")

- A. "Data Analysis"
- B. "Data Snalysis"
- C. "DataAnalysis"

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- **Advanced If and Loops**
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Advanced if conditions

Again??

We can add if conditions inside if conditions, creating a nested if condition!

```
first_name = 'Omar'
last_name = 'Hammad'

if first_name == 'Omar':
  if last_name == 'Hassan':
    print("Hello, Omar Hassan")
  else:
    print("Hello, stranger Omar")   # This will be printed
else:
  print("Hello, kind stranger")
```

Advanced if conditions

We can write if conditions in one line: inline if statement

```
x = 5

# inline if
y = x * 2 if x > 0 else x * -1
print(y)    # y = 10

# This is the same as:
# if x > 0:
#   y = x * 2
# else:
#   y = x * -1
# print(y)
```

```
if x > 0:
    y = x * 2
else:
    y = x * -1
```

```
y = x * 2 if x > 0 else x * -1
```

Advanced for loop

For loop can be used to iterate over any iterable

Lists, tuples, strings, sets and dictionaries are all examples of Python iterables

```python
name = "Omar"

for ch in name:
  print(ch.upper())

  # O
  # M
  # A
  # R
```

```python
my_list = ['Apple', 'Orange', 'Banana']

for fruit in my_list:
  print(f"Fruit: {fruit}")

  # Fruit: Apple
  # Fruit: Orange
  # Fruit: Banana
```

Advanced for loop

Continue: skip the current iteration and go to the next one
Break: break out of the loop and end the loop

```python
# Print only odd numbers
for x in range(10):
  if x % 2 == 0:
    continue
  print("X:", x)

  # X: 1
  # X: 3
  # X: 5
  # X: 7
  # X: 9
```

```python
# Print numbers up to six
for x in range(10):
  if x == 6:
    break
  print("X:", x)

  # X: 0
  # X: 1
  # X: 2
  # X: 3
  # X: 4
  # X: 5
```

Advanced for loop

We can use ‘else’ with for loops, just like ‘if’

The code block in ‘else’ will only be executed if the loop finishes running normally. If a break happens, the ‘else’ block will not be executed

```
names = ['Omar', 'Mohamed', 'Karim']

for name in names:
  if name == 'Hussien':
    print("Hello Hussien")
    break
else:
  # This will be executed
  print("I'm finished")

# I'm finished
```

```
names = ['Omar', 'Mohamed', 'Karim']

for name in names:
  if name == 'Omar':
    # This will be executed
    print("Hello Omar")
    break
else:
  print("I'm finished")

# Hello Omar
```

Advanced while loop

Just like for loops, we can use break and continue using while loops too

```python
x = 0

#Print only even numbers
while x < 10:
  x += 1
  if x % 2 != 0:
    continue
  print(f'X: {x}')

# X: 2
# X: 4
# X: 6
# X: 8
# X: 10
```

```python
x = 0

#Print numbers up to 6
while x < 10:
  x += 1
  if x == 6:
    break
  print(f'X: {x}')

# X: 1
# X: 2
# X: 3
# X: 4
# X: 5
```

Advanced while loop

‘else’ statement works on while too!

```
x = 0

#Print only even numbers
while x < 10:
  x += 1
  if x % 2 != 0:
    continue
  print(f'X: {x}')
else:
  print("I'm finished")

# X: 2
# X: 4
# X: 6
# X: 8
# X: 10
# I'm finished
```

Quiz Time!

What will be the output of the following statements:

Q1

```
x = 0
a = 0
b = -5
if a > 0:
    if b < 0:
        x = x + 5
    elif a > 5:
        x = x + 4
    else:
        x = x + 3
else:
    x = x + 2
print(x)
```

- A. 2
- B. 5
- C. 3

Q2

```
for l in 'Jhon':
   if l == 'o':
      continue
   print(l)
else:
  print("It's John!")
```

- A. J, h, o, n, It’s John!
- B. J, h, n, It’s John!
- C. J, h, n

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- **List Comprehensions**
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

List Comprehensions

List comprehensions offer a shorter way to create a new list based on the values of an existing list

It can be used instead of typing a full for loop

```python
lst = [1, 2, 3, 5, 7, 11]

new_list = [x ** 2 for x in lst]
print(new_list)

# [1, 4, 9, 25, 49, 121]
```

Same as

```python
lst = [1, 2, 3, 5, 7, 11]
new_list = []

for x in lst:
  new_list.append(x ** 2)
print(new_list)

# [1, 4, 9, 25, 49, 121]
```

Quiz Time!

What will be the output of the following statement:

Q1.

```
lst = [int(x*x) for x in range(3,12,4)]

print(lst[-2])
```

- A. 121
- B. 9
- C. 49

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- **Dictionary Comprehensions**
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Dictionary Comprehensions

Dictionary comprehension is a method for transforming one dictionary into another dictionary. During this transformation, items within the original dictionary can be conditionally included in the new dictionary and each item can be transformed as needed.

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
double_dict1 = {k:v*2 for (k,v) in dict1.items()}
print(double_dict1)

# {'a': 2, 'b': 4, 'c': 6, 'd': 8, 'e': 10}
```

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
dict1_keys = {k*2:v for (k,v) in dict1.items()}
print(dict1_keys)

#{'dd': 4, 'ee': 5, 'aa': 1, 'bb': 2, 'cc': 3}
```

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- **Exceptions**
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Exceptions

When an error occurs in Python, the whole program crashes and stops execution

Exception handling is a way of handling errors so that the program can overcome them and continue running normally

```python
x = 5
y = 0

try:
  print(x+y)
  print(x-y)
  print(x*y)
  print(x/y) # This will yield a ZeroDivisonError
  print(x**y)
except:
  print("An error occured")

# 5
# 5
# 0
# An error occured
```

Exceptions

We can make Python check for specific errors

```python
x = 5
y = 0

try:
  print(x+y)
  print(x-y)
  print(x*y)
  print(x/y) # This will yield a ZeroDivisonError
  print(x**y)
except ZeroDivisionError:
  # Error is handled here
  print("Can't divide by zero")
except ValueError:
  print("Encountered value error")

# 5
# 5
# 0
# Can't divide by zero
```

Check Python Error Types

https://docs.python.org/3/library/exceptions.html

Exceptions

Try statements have two extra features:

- ‘else’ : will execute if no errors were caught
- ‘finally’ : will execute whether there were errors caught or not (always execute)

```
x = 5
y = 2

try:
  print(x+y)
  print(x-y)
  print(x*y)
  print(x/y)
  print(x**y)
except ZeroDivisionError:
  print("Can't divide by zero")
except ValueError:
  print("Encountered value error")
else:
  print("No errors encountered, yay!")
finally:
  print("I will always be executed")

# 7
# 3
# 10
# 2.5
# 25
# No errors encountered, yay!
# I will always be executed
```

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- **File Handling**
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

File Handling

Python supports handling of various file types, one example is text files

Python can open text files in three modes:

- Read mode (r)
- Write mode (w)
- Append mode (a)

Handling files with Python is very important since most of our data is stored in files of different types

```python
# Read from file
my_file = open('test.txt', 'r')
print(my_file.read())   # or use readlines()
my_file.close()


# Write to a file
my_file = open('test.txt', 'w')  # or w+ for read & write
print(my_file.write('Hello Python'))
my_file.close()


# Append to a file
my_file = open('test.txt', 'a')  # or a+ for read & append
print(my_file.write('Hello Python'))
my_file.close()

```

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- **Functions**
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- Lambda Expressions

Functions

Functions are used to store a block of code to run later when needed

They become very handy when code needs to be used frequently, and it helps in encapsulation

In Python, we define a function and give it a name. When we need to use it, we ‘call’ it using it’s name

```
def say_hello():
    print('hello')


say_hello()

# hello
```

Function Arguments

Functions can have parameters that would be passed when the function is called

Those are called input arguments

```python
# x is an input argument
def even_or_odd(x):
  if x % 2 == 0:
    print('even')
  else:
    print('odd')

even_or_odd(7) # → x = 7
# odd
```

```python
def greet(name):
  print(f'Hello, {name}!')

greet('Omar')
# Hello, Omar!
```

Function Return

In many cases, functions can be used to perform a certain operation to calculate a value

We usually need this value for further use

We can use functions' 'return' to return a value back to our program

```python
def circle_area(radius):
  area = 3.14 * radius ** 2
  return area

result = circle_area(10)
print(result)
# 314.0
```

Quiz Time!

What will be the output of the following statement:

Q1.

```
def repeat(message, num = 1):
    print(message * num)

repeat('Welcome')
repeat('Viewers', 3)
```

- A. Welcome
  Viewers
- B. Welcome
  ViewersViewersViewers
- C. Welcome
  Viewers,Viewers,Viewers
- D. Welcome

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- **Built-in functions & Operators (zip, enumerate, range, …)**
- Map, Filter, Reduce
- Lambda Expressions

Built-in functions & Operators

Python supports a lot of functions and operators that are built inside it

Examples of built-in functions:

- Range
- Enumerate
- Zip
- In operator

And much more!

Check out more functions here
https://docs.python.org/3/library/functions.html

Range

- range(start, stop, step) – returns a list of value starting with ‘start’ up to ‘stop’, taking a step size of ‘step’

  It stops **before** the stop, meaning it doesn’t include the ‘stop’ value

```
numbers = list(range(1, 10, 2))
print(numbers)
# [1, 3, 5, 7, 9]
```

Enumerate

- enumerate(list) – returns a list of tuples containing an index associated to each value in the original list

This is very useful when we need to iterate through a list and use the index at the same time

```
fruits = ['apple', 'orange', 'banana', 'grapes']

result = list(enumerate(fruits))
print(result)
# [(0,'apple'), (1,'orange'), (2,'banana'), (3,'grapes')]
```

Zip

- zip(list1, list2, …) – concatenates two or more lists together, element wise

It’s very useful when we need to iterate over multiple lists at the same time (e.g. we need to iterate over student name and grade)

```
names = ['George', 'Benjamin', 'Abraham']
grades = [80, 75, 100]

result = list(zip(names, grades))
print(result)
# [('George', 80), ('Benjamin', 75), ('Abraham', 100)]
```

in

We used the in operator earlier when dealing with for loops

‘in’ operator can also be used for logical operations, to check if a value exists in some container

```
names = ['George', 'Benjamin', 'Abraham']

print('George' in names)
# True

text = 'Hello World'
print('x' in text)
# False
```

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- **Map, Filter, Reduce**
- Lambda Expressions

Map, Filter, Reduce

There are a bunch of other useful functions in Python like map, filter & reduce

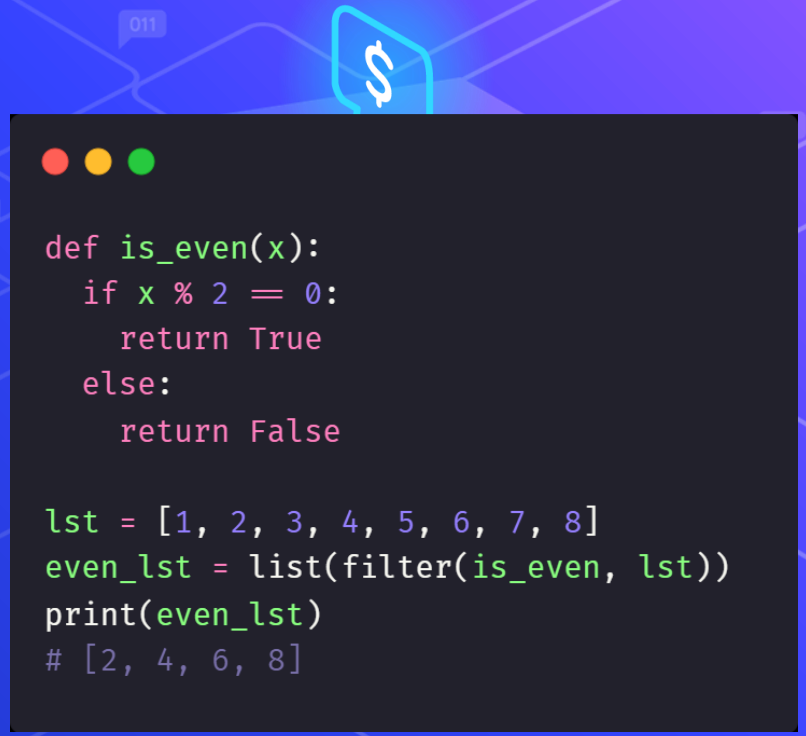
- map(function, list) – applies a certain function on each element in the list and returns a new list with those values

```
def power(x):
  return x ** 2

lst = [1, 2, 3, 4, 5]
new_lst = list(map(power, lst))
print(new_lst)
# [1, 4, 9, 16, 25]
```

Map, Filter, Reduce

- filter(function, list) – applies a filter function on the list and returns the values that have a True value on the filter function



```
def is_even(x):
  if x % 2 == 0:
    return True
  else:
    return False

lst = [1, 2, 3, 4, 5, 6, 7, 8]
even_lst = list(filter(is_even, lst))
print(even_lst)
# [2, 4, 6, 8]
```

Map, Filter, Reduce

- reduce(function, list) – applies a function on the list that reduces all elements into a single value (like a sum) and returns that value

```
from functools import reduce

def add(a, b):
  return a + b

lst = [1, 2, 3, 4, 5]
reduce_lst = reduce(add, lst)
print(reduce_lst)
# 15
```

Quiz Time!

What will be the output of the following statements:

Q1

```
words = ["bay", "cat", "boy", "fan"]
b_words = list(filter(lambda word: word.startswith("b"), words))
print(b_words)
```

- A. [“bay”, “cat”]
- B. [“bay”, “boy”]
- C. ["bay", "cat", "boy", "fan"]

Q2

```
multiply_by_two = lambda x: x * 2
numbers = [1, 2, 3]
doubled = map(multiply_by_two, numbers)
doubled_list = list(doubled)
print(doubled_list)
```

- A. [1, 2, 3]
- B. [1, 4, 9]
- C. [2, 4, 6]

Introduction to Python programming Course Outline

- Intro to Computer Science
- Environment Setup (Anaconda)
- Command Line
- Conda & pip package managers
- Jupyter Notebook
- Input & Output
- Variables
- Data types
- Numbers & Math
- Boolean & Comparison and Logic
- If Conditions
- For & While Loops
- Strings
- Lists
- Tuples
- Sets
- Dictionaries
- Advanced If and Loops
- List Comprehensions
- Dictionary Comprehensions
- Exceptions
- File Handling
- Functions
- Built-in functions & Operators (zip, enumerate, range, …)
- Map, Filter, Reduce
- **Lambda Expressions**

Lambda Expressions

In Python, Lambda expressions are used to define an anonymous function - a function with no name that can't be called

```
def power(x):
  return x ** 2

lst = [1, 2, 3, 4, 5]
new_lst = list(map(power, lst))
print(new_lst)
# [1, 4, 9, 16, 25]
```

Why would we need this?

You may have noticed in the previous examples that we defined a function for each operation we had to do

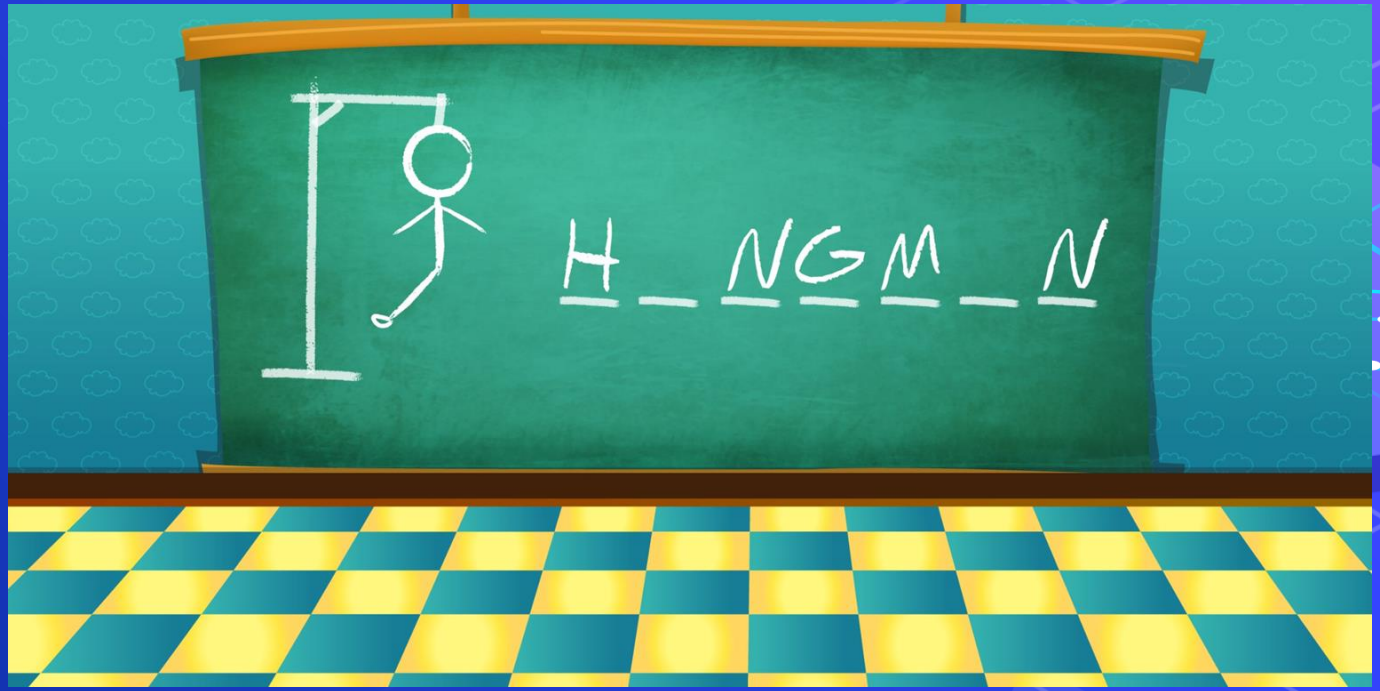
We can use lambda functions to define functions on the fly without having to define them earlier separately

```
lst = [1, 2, 3, 4, 5]
new_lst = list(map(lambda x: x ** 2, lst))

print(new_lst)
# [1, 4, 9, 16, 25]
```

Project #1 Rock Paper Scissors

Project #2 HangMan



Questions ?!

Thanks!

>_ Live long and prosper