Python

Cheat Sheet

Andrei Dumitrescu

This cheat sheet provides you with all the Python fundamentals in one place.

If you want to **MASTER** all the Python key concepts starting from Scratch, check out my Python Complete Bootcamp course. No prior Python knowledge is required.

V1.0

TABLE OF CONTENTS

Variables

Defining variables

```
miles = 10              # type integer
first_name = 'John'     # type string (the value of the variable is between single quotes)
last_name = "Wick"      # type string (use single or double quotes)
a, b, c = 1.5, 3, 'x'   # defining 3 variables on the same line (float, an integer and a string)

# PEP 8 recommends using snake _case for variable names
max_permitted_value = 500    # snake-case notation
maxPermittedValue = 500      # camel-case notation

# Invalid or not recommended names
4you = 10             # not permitted, name starts with a digit
valu!es = 100         # not permitted, name contains special characters
str = 'Python'        # not recommended, name str is a Python language keyword
_location = 'Europe' # not recommended name.
# Avoid names that start with underscores (they have special meaning)
```

Comments in Python

Comments in Python start with the hash character # and extend to the end of the physical line. If you want to comment out more lines, insert a hash character at the beginning of each line.

```
# This line is a comment.

The following line is commented out and will be ignored by the Python Interpreter
# x = 1

a = 7    # defines a variable that stores an integer
my_str = 'A hash character # within a string literal is just a hash character'
```

Data Types

```
age = 31      # type int
miles = 3.4  # type float
finished = True                  # type bool
name = 'Andrei'                  # type str (string)
years = [2018, 2019, 2020]    # type list
week_days = ('Monday', 'Tuesday', 'Wednesday')  # type tuple
vowels = {'a', 'e', 'o', 'u'}               # type set
fs = frozenset((1, 2, 'abc', 'xyz'))   # type frozenset

# type dictionary
countries = {'de':'Germany', 'au':'Australia', 'us':'United States of America', 'gr':'Greece'}
```

Python Operators

Arithmetic Operators

```
a = 9
b = 4
a + b         # addition operator => 13
a - b         # subtraction operator => 5
a * b         # multiplication operator => 36
a / b         # division operator => 2.25
a // b        # floor division operator => 2
5.0 // 3.0    # => 1.0 -> works on floats too
a ** b        # exponentiation operator (a to the power of b) => 6561
a % b         # modulus operator => 1
```

Assignment Operators

```
a = 5
a += 2    # shorthand for a = a + 2 => a = 7
a -= 3    # shorthand for a = a - 3 => a = 4
a /= 2    # shorthand for a = a / 2 => a = 2
a *= 3    # shorthand for a = a * 3 => a = 6
a **=2    # shorthand for a = a ** 2 => a = 36
```

Arithmetic Built-in Function

```
divmod(9, 4)    # returns the quotient and the remainder using integer division => (2, 1)
sum([1,2,4])    # returns the sum of an iterable => 7
min(1,-2,3)     # returns the minimum value => -2
max(1,2,4)      # returns the maximum value => 4
a = 10/3        # a = 3.3333333333
round(a, 4)     # returns a number rounded with 4 decimals => 3.3333
pow(2, 4)       # 2 ** 4 = 16
```

Comparison and Identity Operators

```
# Assignment operator is =
a = 2
b = 3

# Equality operator is ==
# It compares the values stored in variables
a == b    # => False
b == b    # => True

# Inequality operator is !=
a != b    # => True

# Other comparisons
a > b     # => False
5 >= 5    # => True
b <= a    # => False

'Python' == 'python'    # => False, case matters
"Python" == 'Python'    # => True, double and single quotes are equivalent

id(a)    # => returns the address where the value referenced by a is stored. Ex: 140464475242000

# is operator checks if two variables refer to the same object (saved at the same memory
address)
a is b  # => False = compares the address of a to the address of b
# equivalent to:
id(a) == id(b)
```

Boolean Variables

```
# True is 1 and False is 0
True == 1     # => True
bool(True)    # => 1

False == 0     # => True
bool(False)    # => 0

1 is True     # => False
0 is False    # => False

True > False                # => True
a = (True + True ) * 10     # => 20

id(True)    # => 10714848 (you'll get another value)
id(4 > 2)   # => 10714848  - the address of True and False is constant during program execution

# The next 2 expressions are equivalent
(4 > 2) == True     # => True
(4 > 2) is True     # => True
```

Truthiness of objects

```
bool(0)       # => False
bool(0.0)     # => False
bool(10)      # => True
bool(-1.5)    # => True

bool('')       # => False (empty string)
bool('py')    # => True

bool([])       # => False (empty list)
bool([1,2])    # => True

bool(())       # => False (empty tuple)
bool((3,4))    # => True

bool({})                      # => False (empty dictionary)
bool({1:'abc',2:55,'a':5})    # => True
```

Boolean Operators

```
# expression1 and expression2   => True when both expressions are True and False otherwise
# expression1 or expression2    => True when any expression is True

a, b = 3, 5
a < 10 and b < 10   # => True
a < 10 and b > 10   # => False

a < 10 or b < 10    # => True
a < 10 or b > 10    # => True

# The next 2 expressions are equivalent
2 < a < 6
a < 2 and a < 6   # more readable

a != 7 or b > 100          # => True
not a == a                 # => False
a == 3 and not b == 7      # => True

not a > 10 and b < 10      # => True

a < 10 or b > 10           # => True
not a < 10 or b > 10       # => False
not (a < 10 or b > 10)     # => False

# !!
# Python considers 4 > 2 and 2 == True
4 > 2 == True      # => False
(4 > 2) == True    # => True
```

Python Strings

Introduction to Strings

Strings (str objects) are enclosed in single or double quotes (' or "). Just be consistent within your code and don't mix up " with '

```
message1 = 'I love Python Programming!'
message2 = "I love Python Programming!"
```

```
# The print() function displays the content  of the variable at the terminal
print(message1)

# hello1 = 'Hi there! I\'m Andrei'    # => error, cannot use ' inside ' ' or " inside " "
hello1 = 'Hi there! I\'m Andrei'      # => correct. ' inside ' ' must be escaped  using \
hello2 = "Hi there! I'm Andrei"       # you can use ' inside " " or " inside ' '

# Instructions between triple quotes (""" or ''') are treated as comments, but they are not comments
but documentation.. It's recommended to use only # for commenting individual lines

'''
This is a documentation (treated as a comment)
a = 5
print(a)
Comment ends here.
'''

# Defining a multiline string
languages = """ I learn Python,
Java,
Go,
PHP. Let's get started!
"""
print(languages)

# \n is a new line
print('Hello \nWorld!')    # => it displays Hello World! on 2 lines

# \ is used to escape characters that have a special meaning
info = '\\ character is used to escape characters that have a special meaning'
print(info)     # => \ character is used to escape characters that have a special meaning
```

User Input and Type Casting

```
# Getting user input
user_input = input("Enter some data: ")    # returns the data as a string

# To perform arithmetic operations you must convert data to int or float
a = input('Enter a:')          # => a is type string
b = int(input('Enter b:'))     # => b is type int
c = float(a) * b               # => multiplying float by int
```

```
# Type casting
a = 3       # => type int
b = 4.5     # => type float
c = '1.2'   # => type str

print('a is ' + str(a))   # => str(a) returns a string from an int
print('b is ' + str(b))   # => str(b) returns a string from a float
d = b * float(c)          # => here I multiply two floats, d is type float.

str1 = '12 a'
# float(str1)    # => error
```

String Indexing, Operations and Slicing

```
# A string is an ordered sequence of UTF-8 characters
# Indexing starts from 0
language = 'Python 3'
language[0]             # => 'P'
language[1]             # => 'y'
language[-1]            # => '3'
language[-2]            # => ' '
"This is a string"[0]   # => 'T'

# language[100]        # => IndexError: string index out of range

# Cannot modify a string, it's immutable
# language[0] = 'J'     # => TypeError: 'str' object does not support item assignment

# len() returns the length of the string
len("This is a string")  # => 16

# Strings can be concatenated with + and repeated with *
print('Hello ' + 'world!')     # => 'Hello world!'
print('Hello ' * 3)            # => 'Hello Hello Hello '
print('PI:' + str(3.1415))    # => Can concatenate only strings

# Slicing returns a new string
# Slicing format [start:end:step ] where start is included, end is excluded, step is by default 1
movie = 'Star Wars'
movie[0:4]       # => 'Star' -> from index 0 included to index 4 excluded
movie[:4]        # => 'Star' -> start index defaults to zero
movie[2:]        # => 'ar Wars' -> end index defaults to the index of the last char of the string
movie[::]        # => 'Star Wars'
```

```
movie[2:100]   # => 'ar Wars -> slicing doesn't return error when using index out of range
movie[1:6:2]   # => 'trW' -> from index 1 included to 6 excluded in steps of 2
movie[6:1:-1]  # => 'aW ra' -> from index 6 included to index 1 excluded in steps of -1 (backwards)
movie[::-1]    # => 'sraW ratS -> reverses the string
```

Formatting Strings

```
price = 1.33
quantity = 5

# f-string literals (Python 3.6+) - PYTHONIC and recommended!
f'The price is {price}'                  # => 'The price is 1.33'
f'Total value is {price * quantity}'     # => 'Total value is 6.65'
f'Total value is {price * quantity:.4f}' # => 'Total value is 6.6500' -> displaying with 4 decimals

# Classical method (old, not recommended)
'The price is {}'.format(price)                       # => 'The price is 1.33'
'The total value is {}'.format(price * quantity)      # => 'The total value is 6.65'
'The total value is {:.4f}'.format(price * quantity)  # => 'The total value is 6.6500' -> displaying with
4 decimals

'The price is {} and the total value is {}'.format(price, price * quantity)     # => 'The price is 1.33 and
the total value is 6.65'
'The price is {0} and the total value is {1}'.format(price, price * quantity)   # => 'The price is 1.33
and the total value is 6.65'
'The total value is {1} and the price is {0}'.format(price, price * quantity)   # => 'The total value is
6.65 and the price is 1.33'

print('The total value is ', price * quantity)     # => 'The total value is 6.65'
```

String Methods

```
dir(str)          # => lists all string methods
help(str.find)    # => displays the help of a method

# All string methods return a new string but don't modify the original one
my_str = 'I learn Python Programming'

# str.upper() returns a copy of the string converted to uppercase.
my_str.upper()    # => 'I LEARN PYTHON PROGRAMMING'
```

```
# str.lower() returns a copy of the string converted to uppercase.
my_str.lower()    # => 'i learn python programming'

# str.strip() removes leading and trailing whitespace
my_ip = '    10.0.0.1 '
my_ip.strip()    # => '10.0.0.1'

my_ip = '$$$10.0.0.1$$'
my_ip.strip('$')    # => '10.0.0.1'

# str.lstrip() remove all leading whitespace in string
my_ip.lstrip()    # => '10.0.0.1 '

# str.rstrip() Remove all trailing whitespace of string
my_ip.rstrip()    # => '    10.0.0.1'

my_str = 'I learn Python'
my_str.replace('Python', 'Go')    # => 'I learn Go'

# str.split() returns a list from a string using a delimiter
my_ip.split('.')    # => ['10', '0', '0', '1']

# By default the delimiter is one or  more whitespaces
my_list = my_str.split()     # => my_list = ['I', 'learn', 'Python', 'Programming']

# str.join() concatenates the elements of a list into a string
':'.join( ['10', '0', '0', '1'])    # => '10:0:0:1'

# in and not in operators test membership
my_ip = '    10.0.0.1 '
'10' in my_ip         # => returns True
'10' not in my_ip     # => returns False
'20' in my_ip         # => returns False

# Other string methods
my_str = 'this is a string. this is a string'

# str.find() returns the first index in my_str where substring 'is' is found or -1 if it didn't find the
substring
my_str.find('is')     # => 2
my_str.find('xx')     # => -1

# str.capitalize() returns a capitalized copy of the string
my_str.capitalize()    # => 'This is a string. this is a string'
```

```
# str.isupper() returns True if my_str is an uppercase string, False otherwise
my_str.isupper()    # => False

# str.islower() returns True if my_str is a lowercase string, False otherwise
my_str.lower().islower()    # => True

# str.count(s) returns the number of occurrences of substring 's' in my_str
my_str.count('s')    # => 6

'0123123'.isdigit()        # => True
'0123123x'.isdigit()       # => False
'abc'.isalpha()            # => True
'0123123x'.isalnum()    # => True

# str.swapcase() inverts case for all letters in string
my_str1 = 'Hi everyone!'
my_str1.swapcase()      # => 'hI EVERYONE!'
```

if ... elif ... else Statements

Execute a specific block of code if a test condition is evaluated to True

Syntax

```
a, b = 3, 5
# if a is less that b execute the indented block of code under the if clause, otherwise go and test
the elif condition

if a < b:
   print('a is less than b')
elif a == b:
   print('a is equal to b')
else:
   print('a is greater than b')
```

or / and operators

```
your_age = 14
```

```
#if ANY expression is True execute the indented block of code under the if clause
if your_age < 0 or your_age > 99:
   print('Invalid age!')
elif your_age <= 2:
   print('You are an infant')
elif your_age < 18:
   print('You are a child')
else:
   print('You are an adult')

a = 3
if 1 < a <= 9:
   print('a is greater than 1 and less than or equal to 9')

# equivalent to:
if a > 1 and a <= 9:
   print('a is greater than 1 and less than or equal to 9')

# The following 3 examples test if number a is divisible by 6
a = 12

# 1st example - nested if
if a % 2 == 0:
   if a % 3 ==0:
      print('Example 1: a is divisible by 2 and 3 (or by 6)')

# 2nd example - and operator. It returns True if both expressions are True, False otherwise
if a % 2 == 0 and a % 3 == 0:
   print('Example 2: a is divisible by 2 and 3 (or by 6)')

# 3rd example
if not (a % 2  and a % 3 ):
   print('Example 2: a is divisible by 2 and 3 (or by 6)')
```

The truthiness of a variable

```
b = 0
if b:   # it tests the truthiness of b or bool(b)
   print('The truthiness of b is True')
else:
   print('The truthiness of b is False')

my_str = 'some string'
```

```
if my_str:   # it tests the truthiness of my_str or bool(my_str)
   print('The truthiness of my_str is True')
else:
   print('The truthiness of my_str is False')

name = 'Andrei'

# Pythonic version
print('Hello Andrei') if name == 'Andrei' else print('You are not Andrei!')

# equivalent to:
if name == 'Andrei':
   print('Hello Andrei')
else:
   print('You are not Andrei')
```

For Loops

It iterates over a sequence and executes the code indented under the for clause for each element in the sequence

```
movies = ['Star Wars', 'The Godfather', 'Harry Potter ', 'Lord of the Rings']

for m in movies:
   print(f'One of my favorites movie is {m}')
else: #the code below gets executed when "for" has finished looping over the entire list
   print('This is the end of the list')
```

range()

```
for i in range(100):
   pass    # => empty instruction or "do nothing"

for i in range (10):  # => from 0 (default, included) to 10 excluded
   print(i, end=' ')
# it prints:  0 1 2 3 4 5 6 7 8 9

for i in range (3, 9):  # => from 3 included to 9 excluded
   print(i, end=' ')
# it prints: 3 4 5 6 7 8
```

```
for i in range (3, 20, 3):  # => from 3 included to 20 excluded in steps of 3
   print(i, end=' ')
# it prints: 3 6 9 12 15 18

for i in range (8, -4, -2):  # => from 8 included to -4 excluded in steps of -2
   print(i, end=' ')
# it prints: 8 6 4 2 0 -2
```

for and continue

```
# for ... continue -> it prints out all letters of the string without 'o'
for letter in 'Python Go and Java Cobol':
   if letter == 'o':
      continue    # go to the beginning of the for loop and do the next iteration
   print(letter, end='')
```

for and break

```
sequence = [1, 5, 19, 3, 31, 100, 55, 34]
for item in sequence:
   if item % 17 == 0:
      print('A number divisible by 17 was found! Breaking the loop...')
      break   # breaks out the loop
else: # belongs to for, not to if
   print('There is no number divisible by 17 in the sequence')

# it prints out the numbers from 0 to 4
for number in range(10):
   if number == 5:
      break
   print(number)

# it prints out the letters Pytho
for letter in 'Python':
   print(letter)
   if letter == 'o':
      break
```

While Loops

```
a = 10

# Infinite Loop, it prints out 10 indefinitely
while a:  # it tests the truthiness of a or bool(a) which is always True
   print(a)

# Printing out the numbers from 10 to 1
while a:    # => "while a:" is equivalent to "while a > 0:"
   print(a)
   a -= 1
else:   # => executes the block of code below after finishing the while loop (and if no "break"
statement was executed)
   print('Finishing printing numbers. a is now 0')

# Printing out only odd numbers between 1 and 20
a = 0
while a < 20:
   a += 1
   if a % 2 == 0:
      continue    # go the the beginning of the while loop
   print(f'Odd number {a}')   #it reaches this line only if the continue statement wasn't executed


# printing out numbers greater than 2
a = 7
while a > 0:
   a -= 1
   if a == 2:
      break   # => it breaks out the while loop and executes the next instruction after while
   print(a)

print('Loop ended.')
```

Python Lists

Introduction to lists

```
# Creating lists
list1 = []                  # empty list
list2 = list()              # empty list
list3 = list('Python')    # => ['P', 'y', 't', 'h', 'o', 'n'] -> creates a list from a string
list4 = ['Python', 'Go', 2018, 4.5, [1,2.3, 'abc']]    # a list stores any type of object
len(list4)                   # => 5 -> returns the number of elements in the list

# Lists are indexed like strings
list4 = ['Python', 'Go', 2018, 4.5, [1,2.3, 'abc']]
list4[0]       # => 'Python'
list4[-1]      # => [1, 2.3, 'abc']
list4[4][1]    # => 2.3
#list4[10]    # Raises an IndexError (out of bounds index)

# A list is a mutable object and can be modified
list4[0] = 'Rust'    # =>['Rust', 'Go', 2018, 4.5, [1, 2.3, 'abc']]

# Lists are sliced like strings. Slicing returns a new list
# General syntax: list[start:stop:step]
# start is included, stop is excluded and step is by default 1
numbers = [1, 2, 3, 4, 5]
numbers[1:4]                  # => [2, 3, 4]
numbers[1:40]                 # => [2, 3, 4, 5]  -> out of bound slicing doesn't return error
numbers[:3]                   # => [1, 2, 3]  -> by default start is zero
numbers[2:]                   # => [3, 4, 5]  -> by default stop is the end of the list
numbers[::]                   # => [1, 2, 3, 4, 5]  -> returns the entire list
numbers[1:5:3]                # => [2, 5]  -> from 2 included to 5 excluded in steps of 3
numbers[4:1:-2]               # => [5, 3]
numbers[0:2] = ['a', 'b']     # => ['a', 'b', 3, 4, 5]    -> slicing modifies a list
numbers[0:2] = ['x', 'y', 'z']    # => ['x', 'y', 'z', 3, 4, 5]

l1 = [1, 2, 3]
l2 = l1              # l1 and l2 reference the same object, l2 IS NOT a copy of l1
l1 is l2             # => True
l1 == l2             # => True
l1.append(4)    # here I've modified both l1 and l2, they are still the same list
l1 is l2             # => True
l1 == l2            # => True

l3 = l1.copy()    # l2 is a copy of l1, they don't reference the same object
l1 == l2             # => True
l1 is l2             # => False
l3.remove(1)
```

```
l1 == l3          # => False
l1 is l3          # => False

l1 = [1, 2]
id(l1)             # => 139875652516360 (you'll get another value)
l1 += [3, 4]       # => [1, 2, 3, 4]  -> concatenating a new list to l1 - equivalent to using extend()
id(l1)             # => 139875652516360  -> it's the same list

l1 = l1 + [5, 6]   # => [1, 2, 3, 4, 5, 6] -> concatenating a new list to l1
id(l1)             # => 139875654318792    -> l1 is a new list
```

Iterating over a list

```
ip_list = ['192.168.0.1', '192.168.0.2', '10.0.0.1']
for ip in ip_list:
    print(f'Connecting to {ip} ...')
```

List Membership

in and not in operators test list membership

```
'10.0.0.1' in ip_list      # => returns True
'192' not in ip_list       # => returns True
'192' in ip_list          # => returns False
```

List Methods

```
dir(list)      # returns a list will all list methods

# list.clear() removes all items from list
l1 = ['a', 'b', 'c']
l1.clear()

list1 = [1, 2.2, 'abc']
len(list1 )     # => 3

# list.append() adds a single element to the end of the list
list1.append(5)          # => [1, 2.2, 'abc', 5]
# list1.append(6, 7)    # TypeError: append() takes exactly one argument (2 given)
list1.append([6, 7])     # => [1, 2.2, 'abc', 5, [6, 7]]

# list.extend() extends the list with elements of an iterable object
list1.extend([5.2])          # => [1, 2.2, 'abc', 5, [6, 7], 5.2]
#list1.extend(5.2)          # TypeError: 'float' object is not iterable
```

```
list1.extend(['x', 'y'])      # => [1, 2.2, 'abc', 5, [6, 7], 5.2, 'x', 'y']

# list.insert() Inserts an item at a given index
list1.insert(2, 'T')              # => [1, 2.2, 'T', 'abc', 5, [6, 7], 5.2, 'x', 'y']

# Insert on the last position
list1.insert(len(list1), 'Q')    # => [1, 2.2, 'T', 'abc', 5, [6, 7], 5.2, 'x', 'y', 'Q']

# list.pop() removes and returns an element of the list
list1 = [1, 2.2, 'T', 'abc', 5, [6, 7], 5.2, 'x', 'y', 'Q']
print(list1)          # => [1, 2.2, 'T', 'abc', 5, [6, 7], 5.2, 'x', 'y', 'Q']
list1.pop()          # => 'Q'
list1.pop(2)         #=> 'T'
print(list1)         # => [1, 2.2, 'abc', 5, [6, 7], 5.2, 'x', 'y']
#list1.pop(50)    # IndexError: pop index out of range

# list.remove() removes the first occurance and doesn't return an item of the list
print(list1)              #[1, 2.2, 'abc', 5, [6, 7], 5.2, 'x', 'y']
list1.remove('abc')    # => [1, 2.2, 5, [6, 7], 5.2, 'x', 'y']
#list1.remove('a')      # ValueError: list.remove(x): x not in list

# list.index() returns the index of an item
letters = list('abcabcabc')
letters.index('b')         # => 1
letters.index('b', 3)      # => 4 -> it starts from index 3
letters.index('b', 3, 6)   # => 4 -> it searches from index 3 to index 6

# list.count() returns the no. of occurrences of an item in a list
letters.count('a')    # => 3

# Sort a list
# list.sort() and sorted(list)
nums = [6, -1, 55, 2.3]
sorted(nums)    # => [-1, 2.3, 6, 55] -> returns a NEW sorted list
print(nums) # => [6, -1, 55, 2.3]

nums.sort()    # sorts the list in-place
print(nums)    # => [-1, 2.3, 6, 55]
max(nums)    # => 55
min(nums)    # => -1
sum(nums)   # => 62.3

# These methods return an error if the list is not sortable
nums.append('5.5')
```

```
#nums.sort()    TypeError: '<' not supported between instances of 'str' and 'int'
#nums.max()     AttributeError: 'list' object has no attribute 'max'

# Converting a list into a string and a string into a list
ip_list = ['192.168.0.1', '192.168.0.2', '10.0.0.1']
# str.join() returns a string from a list
ip_str = ':'.join(ip_list)       # => ip_str is equal to '192.168.0.1:192.168.0.2:10.0.0.1'

# str.split() returns a list from a string
ip_list = ip_str.split(':')    # => ip_list is equal to ['192.168.0.1', '192.168.0.2', '10.0.0.1']
```

List Comprehension

Syntax: list = [expression for item in iterable if condition]

```
s = [x for x in range(10)]
print(s)    # => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

evens = [x for x in s if x % 2 == 0]
print(evens)    # => [0, 2, 4, 6, 8]


word_list = 'I learn Python programming!'.split()
info = [[w.upper(), w.lower(), len(w)] for w in word_list]
print(info)

# Celsius to Fahrenheit
celsius = [7.12, 10.1, 14.15, 22.5, 29.4, 32.9]
fahrenheit = [ 1.8 * x + 32 for x in celsius ]
print(fahrenheit)    # => [44.816, 50.18, 57.47, 72.5, 84.92, 91.22]

miles = [12, 10, 26, 80]
# 1 mile = 1.609 km
km = [m * 1.609 for m in miles]
print(km)    # => [19.308, 16.09, 41.834, 128.72]
```

Python Tuples

A tuple is an immutable ordered sequence of objects of any type.

Introduction to tuples

```
# Creating tuples
t0 = ()                         # empty tuple
t1 = tuple()                    # empty tuple
t = (1.2)                       # this isn't a tuple, it's a float!
type(t)                         # => float
t2 = (1.2,)                     # creating a tuple with a single element (comma is mandatory)
t3 = tuple('abc')               # creating a tuple from an iterable (string)
t4 = tuple([1, 3.2, 'abc'])     # creating a tuple from an iterable (list)
t5 = (1, 3.2, 'abc')

# Tuples are indexed like strings and lists
t5[0]       # => 1
t5[2]       # => 'abc'
t5[-1]      # => 'abc'
#t5[10]    # => IndexError: tuple index out of range

# Tuples are  immutable objects. Can't be changed.
# t5[0] = 4    # => TypeError: 'tuple' object does not support item assignment

# Tuples are sliced like strings and lists.
# The start is included and the stop is excluded
print(t5)    # => (1, 3.2, 'abc')
t5[0:2]      # => (1, 3.2)
t5[:2]       # => (1, 3.2)
t5[::]       # => (1, 3.2, 'abc')
t5[::2]        # => (1, 'abc') -> in steps of 2
t5[-1:0:-1]    # => ('abc', 3.2)
```

Iterating over a tuple

```
movies = ('The Wizard of Oz', 'The Legend', 'Casablanca')
for movie in movies:
    print(f'We are watching {movie}')
```

Tuple membership

in and not in operators test tuple membership

```
'The Legend' in movies      # => True
'The Legend' not in movies  # => False
```

Tuple Methods

```
dir(tuple)    # returns a list will all tuple methods

my_tuple = (1, 2.2, 'abc', 1)
len(my_tuple)    # => 4

# tuple.index() returns the index of an item
my_tuple.index(1)     # => 0 -> the index of the first element with value 1
my_tuple.index(10)    # => ValueError: tuple.index(x): x not in tuple

# tuple.count() returns the no. of occurrences of the item in tuple
my_tuple.count(1)    # => 2

# Sorting tuples
# tuple.sort() and sorted(tuple)
nums = (6, -1, 55, 2.3)
sorted(nums)    # => (-1, 2.3, 6, 55) -> returns a new sorted list
max(nums)       # => 55
min(nums)       # => -1
sum(nums)       # => 62.3
```

Sets and Frozensets in Python

A set is an unordered collection of immutable unique objects.

Introduction to sets

```
# Creating sets
set1 = set()       # empty set
#x = {}            # x is a dictionary, not a set

set2 = {'a', 1, 2, 1, 'a', 2.3, 'a'}                # => {1, 2, 2.3, 'a'} -> unique unordered collection
set3 = set('hellooo python')                        # =>{'n', 'e', 'p', 't', 'o', 'h', 'l', ' ', 'y'}
set4 = set([1, 2.3, 1, 'a', 'a', 2.3, 'b', 5])    # => {1, 2.3, 5, 'a', 'b'}
#set4[0]                                           # TypeError: 'set' object does not support indexing
set5 = {(1, 2), 'a'}     # a set can contain immutable objects like tuples
#set6 = {[1, 2], 'a'}    # TypeError: unhashable type: 'list' -> list is mutable, not allowed in set

s1 = {1, 2, 3}
s2 = {3, 1, 2}
s1 == s2    # => True - order does not matter
```

```
s1 is s2     # => False

# The assignment operator (=) creates a reference to the same object
s3 = s1
s3 is s1       # => True
s3 == s1       # => True
s3.add('x')    # adds to the set
print(s1)      # => {1, 2, 3, 'x'}
s3 == s1       # => True
s3 is s1       # => True
```

Iterating over a set

```
some_letters = set('abcabc')
for letter in some_letters:     # prints: c a b
   print(letter, end=' ')
```

Set Membership

```
# in and not in operators test set membership
'a' in some_letters          # => True
'aa' in some_letters         # => False
'bb' not in some_letters     # => True
```

Set Methods

```
# set.copy() creates a copy of the set (not a reference to the same object)
s4 = s1.copy()
s4 is s1     # => False
s4 == s1     # => True
s4.add('z')
s4 == s1     # => False

s1 = {1, 2, 3, 'x'}
# set.pop() removes and returns an arbitrary set element
item = s1.pop()
print(f'item:{item}, s1:{s1}')  # => item:1, s1:{2, 3, 'x'}

# set.discard()  removes an element from a set if it is a member.
# If the element is not a member, do nothing.
s1.discard(2)      # discards element from the set, s1 is {3, 'x'}
```

```
s1.discard(22)    # no error if the element doesn't exist

# set.remove() removes an element from a set; it must be a member.
# If the element is not a member, raise a KeyError.
#s1.remove(100)    # KeyError if element doesn't exist
s1.clear()              # Removes all elements from this set
```

Set and Frozenset Operations

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# set.difference() returns the set of elements that exist only in set1 but not in set2
set1.difference(set2)    # => {1, 2}
set1 - set2                  # => {1, 2}

# set.symetric_difference() returns the set of elements which are in either of the sets but not in
both
set1.symmetric_difference(set2)    # => {1, 2, 4, 5}
set1 ^ set2                                # => {1, 2, 4, 5}

# set.union() returns the set of all unique elements present in all the sets
set1.union(set2)       # => {1, 2, 3, 4, 5}
set1 | set2                # => {1, 2, 3, 4, 5}

# set.intersection() returns the set that contains the elements that exist in both sets
set1.intersection(set2)    # => {3}
set1 & set2                   # => {3}

set1.isdisjoint(set2)        # => False
set1.issubset(set2)          # => False
set1 > set2                    # => False
set1 <= set2                   # => False
{1, 2} <= {1, 2, 3}            # => True

# A frozenset is an immutable set
fs1 = frozenset(set1)
print(fs1)    # => frozenset({1, 2, 3})

# All set methods that don't modify the set are available to frozensets
fs1 & set2    # => frozenset({3})
```

Set Comprehension

General Syntax: set = {expression for item in iterable if condition}

```
set1 = {item for item in [1, 2, 1, 2, 1, 2, 3, 4, 3]}
print(set1)     # => {1, 2, 3, 4}

set2 = {item ** 2 for item in set1 if item %2 == 0}
print(set2)     # => {16, 4}

# Lists with duplicates
cities = ['Paris', 'NYC', 'BERLIN', 'Liverpool', 'Osaka', 'Barcelona']
capitals = ['Paris', 'BERLIN', 'Madrid', 'Paris', 'BERLIN']

# Set comprehension returns a set with capitalized cities in both lists
capitals_unique = {word.capitalize() for word in set(cities) & set(capitals)}
print(capitals_unique)      # => {'Paris', 'Berlin'}
```

Dictionaries in Python

A dictionary is an unordered collection of key: value pairs

Introduction to dictionaries

```
# Creating dictionaries
dict1 = {}          # empty dictionary
dict2 = dict()    # empty dictionary

# Keys must be immutable types, values can any type
# invalid_dict = {[1,2,3]: "123"}    # => Raises a TypeError: unhashable type: 'list'

# key of type tuple is permitted (immutable). Values can be of any type
valid_dict = {(1,2,3):[1,2,3], 3: 'abc', 'abc':{14,'a'}, 4.5:True}

product = {'id':1234, 'name':'Laptop'}
product['seller'] = 'Amazon'           # adds a new key:value pair
product.update({'price':888.99})    # another way to add to a dictionary

p = product['price']              # getting the value of a key,  p is 888.00

del product['seller']           # removing a key:value pair
print(product)                    # => {'id': 1234, 'name': 'Laptop', 'price': 888.99}
```

```
#seller = product['seller']    # Looking up a non-existing key is a KeyError

# dict.get() retrieves the value of a key
product.get("id")          # => 1234
product.get("review")    # => None -> it doesn't return KeyError

# dict.get() supports a default argument which is returned when the key is missing
product.get("id", 4)              # => 1234
product.get("review", 'N/A')    # => 'N/A'

# dict.pop() removes the specified key and returns the corresponding value.
# If key is not found, a default value is given, otherwise KeyError is raised
name = product.pop('name')                    # name is 'Laptop'
print(product)                                # => {'id': 1234, 'price': 888.99}
# name = product.pop('name')                  # => KeyError: 'name', key 'name' doesn't exist anymore
name = product.pop('name', 'No such key')   # => name is 'No such key'
```

Dictionary Operations and Methods

```
product = {'id':1234, 'price':888.99}

# in and not in operators test dictionary key membership
'price' in product  # => True
5 in product       # => False

# Getting dictionary views: dict.keys(), dict.values() and dict.items()
keys = product.keys()    # getting all keys as an iterable
keys = list(keys)          # it can be converted to a list
print(keys)                # => ['id', 'price']

values = product.values()    # getting all values as an iterable
values = list(values)          # it can be converted to a list
print(values)                  # => [1234, 888.99]

key_values = product.items()    # getting all key:value pairs as tuples
key_values = list(key_values)    # it can be converted to a list of tuples
print(key_values)                 # => [('id', 1234), ('price', 888.99)]

# dict.copy() creates a copy of a dictionary
prod = product.copy()

# Return the number of key:value pairs in the dictionary
len(prod)
```

```
#  dict.clear() removes all items from dictionary
product.clear()
```

Iterating over a dictionary

```
product = {'id':1234, 'price':888.99}

# Iterating over the keys
for k in product:
   print(f'key:{k}')

#equivalent to:
for k in product.keys():
   print(f'key:{k}')

# Iterating over the values
for v in product.values():
   print(f'value:{v}')

# Iterating over both the keys and the values
for k,v in product.items():
   print(f'key:{k}, value:{v}')
```

zip() Built-in Function

Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables.

The iterator stops when the shortest input iterable is exhausted.

```
years = [2015, 2016, 2017, 2018]
sales = [20000, 30000, 40000, 45000]

# Zipping in a list of tuples
sales_list = list(zip(years, sales))
print(sales_list)   # => [(2015, 20000), (2016, 30000), (2017, 40000), (2018, 45000)]

# Zipping in a dictionary
sales_dict = dict(zip(years, sales))
print(sales_dict)   # => {2015: 20000, 2016: 30000, 2017: 40000, 2018: 45000}
```

Dictionary Comprehension

```
d1 = {'a':1, 'b':2, 'c':3}

## Doubled values
d2 = {k: v * 2 for k, v in d1.items()}
print(d2)    # => {'a': 2, 'b': 4, 'c': 6}

## Doubled keys, squared values
d3 = {k * 2: v * 3 for k, v in d1.items() }
print(d3)    # => {'aa': 3, 'bb': 6, 'cc': 9}

sales = {2015: 20000, 2016: 30000, 2017: 40000, 2018: 45000}
# Creating a dictionary called vat considering that the value added tax is 20%
vat = {k: v * 0.2 for k, v in sales.items()}
print(vat)    # => {2015: 4000.0, 2016: 6000.0, 2017: 8000.0, 2018: 9000.0}
```

Python Functions

Introduction to Functions

```
# Use def to define new functions
def my_function1():
    """
     This is the function's docstring.
    """
    print('This is the function's body!')
    # this function returns None implicitly

# Calling the function
my_function1()              # => This is a function!
my_function1.__doc__     # => This is the function's docstring.

# The return statement exits the function
def my_function2():
    x = 1
    return x                                # the function ends here
    print('Never reaches this line!')    # it will never reach this line

# Calling the function
my_function2()     # returns 1
```

```
# A function can return more values as a tuple
def add_multiply_power(x, y):
    return x + y, x * y, x ** y

# Calling the function
a, b, c = add_multiply_power(2, 3)    # returns (2 + 3, 2 * 3, 2 ** 3)
print(a, b, c)                        # => 5 6 8
```

Function's Arguments

```
# 1. Function with positional arguments
def add(x, y):
    print(f"x is {x} and y is {y}")
    return x + y    # returns the result of x + y

# Calling function with positional arguments
s = add(5, 6)    # => prints out "x is 5 and y is 6" and returns 11, s is 11

# Calling function with keyword arguments
s = add(y=1, x=8)    # => prints out "x is 8 and y is 1" and returns 9, s is 9


# 2. Function with default arguments
def add(x=1, y=0):
    print(f"x is {x} and y is {y}")
    return x + y    # returns the result of x + y

# Calling function with default arguments
s = add()       # => prints out "x is 1 and y is 0" and returns 1, s is 1
s = add(5)      # => prints out "x is 5 and y is 0" and returns 5, s is 5
s = add(5,3)    # => prints out "x is 5 and y is 3" and returns 8, s is 8

# wrong way to define a function => SyntaxError: non-default argument follows default argument
# def my_function(a, b=5, c):
#     print(a, b, c)

#3.  Function that takes a variable number of positional arguments
def concatenate(*args):
    result = ''
    for tmp in args:
        result = result + tmp
```

```
    return result

# Calling the function
result = concatenate()
print(result)          # => '' -> empty string

result = concatenate('Python', '!')
print(result)          # => Python!

result = concatenate('I', 'love ', 'programming')
print(result)         # => Ilove programming


#4.  Function that takes a variable number of keyword arguments
def device_info(**kwargs):  #kwargs is a dictionary
   for k, v in kwargs.items():
      print(f'{k}: {v}')

# Calling the function
device_info(name='Cisco Router', ip='10.0.0.1', username='u1', password='secretpass')
# or:
d1 = {name='HP', ip='192.168.0.1', username='root', password='secret123'
device_info(**d1)
```

Scopes and Namespaces

```
x = 3    # this is a global scoped variable

def my_func1():
    print(f'x is {x}')  # this is "x" from the global namespace

# Calling the function
my_func1()      # => x is 3


def my_func2():
    x = 6                 # this is a local scoped variable
    print(f'x is {x}')    # this is NOT "x" from the global namespace

# Calling the function
my_func2()    # => x is 6
print(x)          # => 3 -> "x" variable was not modified inside the function
```

```
def my_func3():
    global x      # importing "x" from the global namespace
    x = x * 10    # this is "x" from the global namespace
    print(f'x is {x}')

# Calling the function
my_func3()    # => x is 30
print(x)          # => 30 -> global "x" variable was modified inside the function


def my_func4():
    print(f'x is {x}')
    x += 7      # this is an error, we used local x before assignment

## Calling the function
my_func4()      # => UnboundLocalError: local variable 'x' referenced before assignment
```

Lambda Expressions

```
# "x" and "y" are lambdas arguments.
add = lambda x, y: x + y    # this creates an anonymous function
type(add)                   # => function


# Assigning lambda expression to a variable
result = add(2, 3)    # => 5

# You can use default arguments
add = lambda x=1, y=0: x + y
result = add()    # => 1

# You can even use *args and **kwargs
my_function = lambda x, *args, **kwargs: (x, *args, {**kwargs})
# x is 2.3, args is (a, b, c) and kwargs is {arg1='abc', arg2='def', arg3='geh'}
my_function(2.3, 'a', 'b', 'c', arg1='abc', arg2='def', arg3='geh')


# Passing lambda as an argument to a function
# Lambdas are functions and can therefore be passed to any other function as an argument (or
returned from another function)
def my_func(x, fn):
    return fn(x)
```

```
result = my_func(2, lambda x: x**2)
print(result)    # => 4

result = my_func(2, lambda x: x**3)
print(result)     # => 8

result = my_func('a', lambda x: x * 3)
print(result)    # => 'aaa'

result = my_func('a:b:c', lambda x: x.split(':'))
print(result)    # => ['a', 'b', 'c'] -> this is a list

result = my_func(('p', 'y', 't', 'h', 'o', 'n'), lambda x: '-'.join(x))
print(result)    # => p-y-t-h-o-n > this is a string
```

Working with Files in Python

```
# Open a file named a.txt and return a file object called f
# a.txt it's in the same directory with the python script
f = open('a.txt', 'r')    # it opens the file in read-only mode

content = f.read()    # reads the entire file as a string
print(content)

f.closed       # => False, file is not closed

# Close the file
f.close()

## Open the file in read-only mode and reads its contents as a list
## the file object will be automatically closed
with open('a.txt', 'r') as my_file:
   content = my_file.readlines()    # content is a list

my_file.closed    # => True, my_file has been closed automatically

# file object is an iterable object
with open('a.txt', 'r') as my_file:
   for line in my_file:    #iterating over the lines within the file
      print(line, end='')
```

```
# Open the file in write-only mode.
# Create the file if it doesn't exist or overwrite the file if it already exists
with open('my_file.txt', 'w') as file:
    file.write('This file will be overwritten!')

# Open the file in append-mode.
# Create the file if it doesn't exist or append to it if it exists
with open('another_file.txt', 'a') as file:
    file.write('Appending to the end!')

# Open the file for both read and write
# Do not create the file if it doesn't exist
with open('my_file.txt', 'r+') as file:
    file.seek(0)    # the cursor is positioned at the beginning of the file
    file.write('Writing at the beginning')    # writing at the beginning

    file.seek(5)    # moving the cursor at position 5
    content = file.read(10)    # reading 10 characters starting from position 5
    print(content)
```

Exceptions Handling

Exceptions are run-time errors.

```
a = 2

# for ZeroDivisionError
b = 0

# for TypeError
b = '0'

# for another Exception
# del b

try:
  # Try to execute this block of code!
  c = a / b
  print(c)
except ZeroDivisionError as e:
  # This block of code is executed when a ZeroDivisionError occurs
```

```
  print(f'There was Division by zero: {e}')
except TypeError as e:
  # This block of code is executed when a TypeError occurs
  print(f'There was a Type Error: {e}')
except Exception as e:
  # This block of code is executed when other exception occurs
  print(Another exception occurred: {e}')
else:
  # This block of code is executed when no exception occurs
  print(f'No exception was raised. c is {c}')
finally:
  print('This block of code is always executed!')

print('Continue script execution...')
# Other instructions and statements
x = 2
```

Object Oriented Programming (OOP)

```
# Defining a new class
class Robot:
  """
   This class implements a Robot.
  """
  population = 0  # class attribute

  # this is the constructor
  # it gets executed automatically each time an object is created
  def __init__(self, name=None, year=None):
     self.name = name          # instance attribute, default None
     self.year = year          # instance attribute, default None
     Robot.population += 1    # incrementing the class attribute when creating a new object

  # this is de destructor, it's automatically called then the object gets out of scope
  # MOST of the time it's not recommended to implement it (Python has a garbage collector)
  def __del__(self):
     # print('Robot died')
     pass

  def set_energy(self, energy):
     self.energy = energy    # instance attribute, set in methods
```

```
  def __str__(self):    # magic method, called automatically when printing the object
    return f'Name: {self.name}, Built Year: {self.year}'

  def __add__(self, other):    # magic method, called automatically when adding 2 objects
    return self.energy + other.energy


r0 = Robot()    # creating an instance with attributes set to default None
print(r0)         # => Name: None, Built Year: None

r1 = Robot('R2D2', 2030)    # creating an instance (object)
print(r1.__doc__)              # => This class implements a Robot. -> class docstring

print('Robot name:', r1.name)   # => R2D2 -> accessing an instance attribute
print(r1.__dict__)            # => {'name': 'R2D2', 'year': 2030} -> dictionary with instance attributes

r1.set_energy(555)        # creating the "energy" attribute
print(r1.energy)            # => 555

print(Robot.population)    # => 2
print(r1.population)         # => 2

r1.population += 10          # creating an instance attribute (doesn't modify the class attribute)
print(Robot.population)    # => 2
print(r1.population)         # => 12
```