Branch: master

Find file Copy path

vivadata-student-003 / curriculum / 01-Python-Programming / 01-Data-Types-and-Data-Structures / 00-Lectures / 01-Data-Types-and-Data-Structures.ipynb

nmviva Add 0101

3858fb3 on 1 Apr

1 contributor

Raw Blame History

1320 lines (1319 sloc) 26.9 KB

I - Python Programming

I.1. Data Types and Data Structures

Photo by Jessica Ruscello (https://unsplash.com/photos/-GUyf8ZCTHM)

I. Data types and operators

Very important pythonic rules :

- Python is case sensitive : always be careful with capital letters
- Spacing matters : use 4 whitespaces to indent your code

**Data types : `int`, `float`, `str`, `bool`

Operators :

Operator	Category
+ - * ** / // %	Arithmetic
+= -= *= \/=	Incrementation and decrementation
+ +=	String concatenation
== != < > <= >=	Comparison
is is not	Identity
and or ! (xor)	Logical
.	Object method or attribute
[ ]	Access specific elements in a list

I.1. Numbers and arithmetic operators

There are two types of numbers : `int` (ex: 2) and `float` (ex: 3.5)

You can use the following arithmetic operators on numbers :

You can use the following arithmetic operators on numbers :

Operation	Symbol
sum	+
substraction	-
multiplication	*
power	**
division	/
integer division	//
modulo	%

- Do not forget to put a whitespace between each number and operator sign or parenthesis
- Be careful : Python follows operator precedence conventions

In [ ]:
```
10 / 2 + 3
```

In [ ]:
```
10 / (2 + 3)
```

In [ ]:
```
# Powers of 10
3e6
```

I.2. Strings

str : sequence of characters between quotes (double or simple as you want).

In [ ]:
```
'Hello world'
```

In [ ]:
```
# If you need to use quotes inside the string, you can use the type of quotes you didn't use first.
"He says "Hello world""
```

In [ ]:
```
"He says 'Hello world'"
```

In [ ]:
```
#  Or a '\' before single quote to indicate it's part of the string.
'This isn't what I said'
```

In [ ]:
```
'This isn\'t what I said'
```

Operations on strings

In [ ]:
```
# `+` for concatenation.
'Hello' + 'world'
```

In [ ]:
```
# Be careful, it's litteral: if you need to put a whitespace between the strings, you have to write it.
'Hello' + ' ' + 'word'
```

In [ ]:
```
# '* ' multiplies the string
'Hello' * 3
```

In [ ]:
```
# The function len returns the number of characters in a given string
len('Hello world')
```

Useful methods

In [ ]:
```
# .lower(): writes everything in lowercase
'HELLO WorlD'.lower()
```

In [ ]:
```
# .upper(): writes everything in uppercase
'heLLo WorlD'.upper()
```

In [ ]:
```
# .capitalize(): 1st letter uppercase and the rest is lowercase.
'heLLo WOrld'.capitalize()
```

In [ ]:
```
# .count(word): counts the occurancies of a given word
'hello world, hello vivadata'.count('hello')
```

In [ ]:
```
# .find('x'): returns the lowest index where a given substring is found (-1 if not found)
'hello world'.find('l')
```

In [ ]:
```
'hello world'.find('i')
```

In [ ]:
```
# .index('x'): same idea but returns Error if not found
'hello world'.index('l')
```

In [ ]:
```
'hello world'.index('a')
```

In [ ]:
```
# in : returns thue if a string contains another string
"world" in "hello world"
```

In [ ]:
```
# .format() : replace the {} in the string by what you want. Useful to write strings with variable argumen
ts.
'{} loves {}'.format('Mary', 'python')
```

More ressources on string formatting https://pyformat.info/ (https://pyformat.info/).

I.3. Booleans and comparison/logical operators

`bool` : represents values of True or False. It is useful with comparison and logical operators.

Comparison operators : `==` `!=` `<` `>` `<=` `>=`

In [ ]:
```
# >, more than
# >=, more or equal to
6 > 5
```

In [ ]:
```
# <, less than
# <=, less or equal to
6 <= 5
```

In [ ]:
```
# ==, equal to (be careful, '=' is used to assign variable you can't use it for equality)
# !=, not equal to
10 != 2
```

Logical operators : `and` `or` `!=` (xor) `!`(not)

A	B	A and B	A or B	A != B	!A
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

In [ ]:
```
x = 10
y = 5
z = 20
```

In [ ]:
```
# and: both sides are True
x > y and x > z
```

In [ ]:
```
# or: at least one side is true
x > y or x > z
```

In [ ]:
```
# not: inverse of a boolean type
x > y
```

In [ ]:
```
not x > y
```

In [ ]:
```
# Some booleans can be used with string also, == compare the identity.
'hello' == 'world'
```

In [ ]:
```
'hello' == 'hello'
```

In [ ]:
```
"10" == 10
```

Idendity operators : `is` and `is not`

== is an equality test. It checks whether the right hand side and the left hand side are **equal** objects (but different references in memory).

is an equality test. It checks whether the right hand side and the left hand side are **equal** objects (but different references in memory).

`is` is an identity test. It checks whether the right hand side and the left hand side are **the very same** object (same reference in memory).

In [ ]:
```
0 == False
```

In [ ]:
```
0 is False
```

In [ ]:
```
_ = 0  # or ""
if _:
    print("It's not working")
if _ is not None:
    print("It's working")
```

I.4. Casting types

Sometimes you need to change data to another type. For example if your number are stored as strings (ex: '34'), you can't make any calculation with them.

Use **`type()`** to get the type of an element

In [ ]:
```
type(34)
```

In [ ]:
```
type(34.0)
```

In [ ]:
```
type('34')
```

In [ ]:
```
type(2 == 2)
```

Use **`isinstance()`** to check the type of an object

In [ ]:
```
isinstance(12, int)
```

In [ ]:
```
isinstance(12.5, int)
```

In [ ]:
```
isinstance("12", int)
```

You can **recast** an element to another type : just pass the element as an argument of the type you want.

In [ ]:
```
type(int(34.0))
```

In [ ]:
```
# Be careful with booleans
bool(0)
```

In [ ]:
```
bool(1)
```

In [ ]:
```
bool(-127)
```

I.5. Variables

When you want to store data, you can assign it to variables with **=**

In [ ]:
```
# Simple assignment
x = 200
```

In [ ]:
```
# Once your variable is assigned, you can reuse it
100 + x
```

In [ ]:
```
# You can also reassign it using '=' again.
# You need to re-run the calculation after changing the variable if you want this change to take account.
x = x - 100
```

In [ ]:
```
# You can define variables and use .format() to pass them as string arguments
watermelon = 5
weight = 5 * 2
'A melon weights {}kg and you buy two, so the total weight is {}kg'.format(watermelon, weight)
# New method : fstrings
f'A melon weights {watermelon}kg and you buy two, so the total weight is {weight}kg'
```

I/O : `print()` and `input()`

In [ ]:
```
# input
first_name = input("What's your first name ?")
```

In [ ]:
```
# output
print(first_name)
```

II. Data structures

Data structures are containers with multiple elements. There are 4 data structures in Python.

	List	Tuple	Set	Dict
Symbol	[,]	(,)	{,}	{k: v,}
Type	ordered	ordered	unordered	unordered
Access	index	index	value	key
Property	mutable	immuable	unique values immuable	keys/values

Everything is detailed in the doc : https://docs.python.org/3/tutorial/datastructures.html (https://docs.python.org/3/tutorial/datastructures.html)

II.1. Ordered sequences

II.1.1. `list` are mutable ordered sequences of elements

- a **`list`** is defined with square brackets [ ] and elements are separated by coma.
- each element is identified by its index (be careful : the first index is 0 not 1)
- a **`list`** can contain any other data type (including lists)
- a **`list`** can mix data types (but it is **not recommanded**)
- **`string`** are sequences

In [1]:
```
# Examples
numbers = [1, 2, 3, 5, 4, 6, 5, 5, 6, 7]
strings = ['a', 'b', 'd', 'c']
mixed = ['a', 1, 3, 4.5, True]
```

In [ ]:
```
# Concatenation
numbers + strings
```

In [ ]:
```
# Contains
# in: evaluates if an object is in the list, returns True or False
8 in numbers
# not in: same in negative
8 not in numbers
# it doesn't work with subsets
[3, 5] in numbers
```

Subsetting

In [ ]:
```
# Indexing : extract a specific element with its index
numbers[1]
# First element
numbers[0]
# Last element
numbers[-1]
```

In [ ]:
```
# Slicing : extract a sequence
# (be careful : the upper bound is exclusive so you need to write the index after the element you want).
numbers[3:6]
# From the beginning
numbers[:3]
# To the end
numbers[3:]
```

Mutability

In [ ]:
```
strings[1] = 'd'
```

Useful methods (you can find a lot more in the doc)

In [ ]:
```
# len: counts the number of elements in the list
len(numbers)
```

In [ ]:
```
# max(): indicates the greatest element of the list
max(strings)
```

In [ ]:
```
# min(): indicates the smallest element
min(numbers)
```

In [ ]:
```
# .index(): retuns the index of a givent element
numbers.index(3)
# you can also find the index of element you have indication about
numbers.index(max(numbers))
```

In [ ]:
```
# sorted(): returns a copy of the list sorted from smallest to greatest element
# reverse=True : sorts in a reverse order.
sorted_numbers = sorted(numbers, reverse=True)
```

In [ ]:
```
# .sort(): modifies the order of the list from smallest to greatest element
numbers.sort()
```

In [ ]:
```
# separator.join(): returns the list with the elements joined by the given separator
alphabet = ['a', 'b', 'c', 'd']
' '.join(alphabet)
```

In [ ]:
```
# .append(): adds elements to the end of the list
numbers.append(8)
# Be careful : if you use it with another list, i will insert a list in a list
numbers.append([9, 10, 11])
```

In [ ]:
```
# .extend(): incorporates elements from another list to the list
numbers.extend([15, 16, 17])
```

In [ ]:
```
# use str.split() to create a list from a string
"this is an example".split()
```

II.1.2. `tuple`are immuable ordered sequences of elements

- a **`tuple`** is defined between parenthesis and elements are separeted by comas
- you can use it with 2 or more elements
- you can extract an element or slice it like a list
- you cannot change the values of elements

In [ ]:
```
# Example
my_tuple = (1, 3 , 4, 'a', 'c', 2)
```

In [ ]:
```
# Tuple unpacking
length, width = 100, 200
```

In [ ]:
```
# List of tuples
students = [
    ("kim", 10),
    ("john", 12),
    ("mark", 8),
    ("nina", 19)
]
```

II.2. Unordered sequences

II.2.1. `set` are mutable unordered collections of unique elements

- **`set`** are defined with curly brackets
- you can create a set from a list: it will remove duplicates
- be careful : elements can be displayed in a different order each time
- **`set`** have a faster access time than **`list`**

In [ ]:
```
countries_ = {'France', 'UK', 'USA', 'China', 'India'}
```

In [ ]:
```
# .add(): to add element.
countries_.add('Russia')
```

In [ ]:
```
# .pop(): to remove randomly an element.
countries_.pop()
```

In [ ]:
```
# Using a set to remove duplicates from a string
set("something is happening")
```

Operations

In [ ]:
```
# Arithmetic
{1, 5, 7} + {3, 4}
{10, 20, 30} - {30, 40, 50}
```

In [ ]:
```
# Union and intersection
{2, 4, 6} | {1, 3, 5}
{1, 2, 3, 4} &  {2, 4, 6, 8}
```

In [ ]:
```
# Is subset and is superset
{5, 10} <= {5, 10, 15, 20}
{20, 40, 60} >= {20, 40}
```

II.2.2 `dict` store mappings of unique keys to values

- **`dict`** are defined by curly brackets, the key/value are associated with colon and there is coma between each pair
- keys can be of any types and can have different types
- **`dict`** also have a faster access time than **`list`**

In [ ]:
```
# Example
grades = {
    'ann': 15,
    'mary': 16,
    'george': 12,
    'william': 11
}
```

In [ ]:
```
# Access the value for a given key
grades['ann']
# Insert a new value
grades['tom'] = 12
```

In [ ]:
```
# Check if it contains a given value
'mary' in grades
# You can also use the method .get() which will returns None if nothing is found.
grades.get('george')
```

Useful methods