# Intro to Python — SMM692

## Python Objects

Simone Santoni

Bayes Business School

MSc Pre-Course Series

# Outline

# Scope

In Chapter 3, the attention revolves around the following topics:

- The concept of Python object
- The types of Python objects
- The characteristics of each Python object

# Why Shall I Learn About Python Objects?

- Built-in objects make coding efficient and easy
  - For example, using the string object, we can represent and manipulate a piece of text — e.g., a newspaper article — without loading any module
- Built-in objects are flexible
  - For example, we can deploy built-in objects to create a class
- Built-in objects have been created and refined over time by a large community of expert developers. Hence, they are often more efficient than ad-hoc objects (unless the creator of the ad-hoc object knows her business!)

# Learning Goals

At the end of the chapter, you will be able to evaluate the various types of Python objects regarding:

- Key features
- Use cases/roles
- Available methods

# What is a Python Object?

In essence, Python objects are pieces of data. Mark Lutz, the author of the popular book Learning Python, points out

> *"... in Python, we do things with stuff. "Things" take the form of operations like addition and concatenation, and "stuff" refers to the objects on which we perform those operations"*

# What Are the Main Families of Python Objects?

In Python, there are two families of objects:

- Built-in objects provided by the Python language itself
- Ad-hoc objects — called classes — we can create to accomplish specific goals

# What Are the Main Types of Built-In Python Objects?

- Strings
- Numbers
- Data containers
  - Lists
  - Dictionaries
  - Tuples
  - Sets
- Files
- Python statements, syntax, and control flow
- Iterators

# Outline

# Integers and Floating Points

The most common number types are integers and floating-point numbers:

- Integers are whole numbers such as 0, 4, or -12
- Floating-point numbers represent real numbers such as 0.5, 3.1415, or -1.6e-19
  - However, floating points in Python do not have — in general — the same value as the real number they represent
  - It is worth noticing that any single number with a period '.' is considered a floating point in Python

Snippet 4.1 — doing 'stuff' with numbers

```
# integer addition
>>> 1 + 1
2

# floating-point multiplication
>>> 10 * 0.5
5.0

# 3 to the power 100
>>> 3 ** 100
515377520732011331036461129765621272702107522001
```

# Number Type Objects in Python

Number Type Objects in Python

| Literal | Interpretation |
|---|---|
| 1234, -24, 0, 99999999999999 | Integers (unlimited size) |
| 1.23, 1., 3.14e-10, 4E210, 4.0e+210 | Floating-point numbers |
| 0o177, 0x9ff, 0b101010 | Octal, hex, and binary literals in 3.X |
| 0177, 0o177, 0x9ff, 0b101010 | Octal, octal, hex, and binary literals in 2.X |
| 3+4j, 3.0+4.0j, 3J | Complex number literals |
| set('spam'), {1, 2, 3, 4} | Sets: 2.X and 3.X construction forms |
| Decimal('1.0'), Fraction(1, 3) | Decimal and fraction extension types |
| bool(X), True, False | Boolean type and constants |

# String Type Fundamentals

**What:** A Python string is a positionally ordered collection of other objects. Strictly speaking, strings are *immutable sequences* of one-character strings; other, more general sequence types include lists and tuples, covered later.

**How and why:** Strings are used to record words, contents of text files loaded into memory, Internet addresses, Python source code, and so on.

Snippet 4.6 — Python strings as sequences

```
# the string
>>> S = "Python 3.X"
# check the length of S
>>> len(S)
6
# access the first unitary string in the sequence behind S
>>> S[0]
"P"
# access the last unitary string in the sequence behind S
>>> S[len(S)-1]
"X"
# or, equivalently
>>> S[-1]
"X"
# access the unitary strings between the i-th and j-th positions
# sequence behind S
>>> S[2:5]
"tho"
```

# Sting Literals and Operators

| Literal/operation | Interpretation |
|---|---|
| `S = ""` | Empty string |
| `S = ‘’` | Single quotes, same as double quotes |
| `S = "spam’s"` | Single quote as a string |
| `S = ‘spam\’s’` | Escape symbol |
| `length(S)` | Length |
| `S[i]` | Index |
| `S[i:j]` | Slice |
| `S1 + S2` | Concatenate |
| `S * 3` | Repeat S *n* times (e.g., three times) |
| `"text".join(strlist)` | Join multiple strings on a character (e.g., “text”) |
| `"{}".format()` | String formatting expression |

# String Literals and Operators (cont'd)

| Literal/operation | Interpretation |
|---|---|
| `S.strip()` | Remove white spaces |
| `S.replace("pa", "xx")` | Replacement |
| `S.split(",")` | Split on a character (e.g., ",") |
| `S.lower()` | Case conversion — to lower case |
| `S.upper()` | Case conversion — to upper case |
| `S.find("text")` | Search substring (e.g., "text") |
| `S.isdigit()` | Test if the string is a digit |
| `S.endswith("spam")` | End test |
| `S.startswith("spam")` | Start test |
| `S = """...multiline..."""` | Triple-quoted block strings |

# Outline

# List Type Fundamentals

**What:** A Python `list` is an *ordered*, *mutable* array of objects. A list is constructed by specifying the objects, separated by commas, between square brackets, []

**How and why:** Lists are just places to collect other objects — being numbers, strings, or even other lists — so you can treat them as groups.

Snippet 4.11 — list indexing and slicing

```
# the list
>>> L = [4, ["abc", 8.98]]
# get the first item of L
>>> L[0]
4
# get the second element of L
>>> L[1]
["abc", 8.98]
# get the first item of L's second item
>>> L[1][0]
"abc"
```

# Popular List Methods

| Method | Synopsis |
|---|---|
| `L.append(X)` | Append an item to an existing list |
| `L.insert(i, X)` | Append an item to an existing list in position *i* |
| `L.extend([X0, X1, X2])` | Extend an existing list with the items from another list |
| `L.index(X)` | Get the index of the first instance of the argument in an existing list |
| `L.count(X)` | Get the cardinality of an item in an existing list |
| `L.sort()` | Sort the items in an existing list |
| `L.reverse()` | Reverse the order of the items in an existing list |
| `L.copy()` | Get a copy of an existing list |
| `L.pop(i)` | Remove the item at the given position in the list, and return it |
| `L.remove(X)` | Remove the first instance of an item in an existing list |
| `L.clear()` | Remove all items in an existing list |

# Sample List Methods in Action

Snippet 4.13 — methods for expanding an existing list

```
# create two lists
>>> L1 = ["Leonard", "Penny", "Sheldon"]
>>> L2 = ["Howard", "Raj", "Amy", "Bernadette"]
# expand an existing list with .append()
>>> L2.append("Priya")
>>> print(L2)
["Howard", "Raj", "Amy", "Bernadette", "Priya"]
# concatenate L1 and L2 with .extend()
>>> L1.extend(L2)
>>> print(L1)
["Leonard", "Penny", "Sheldon", "Howard", "Raj", "Amy", "Bernadette", "Priya"]
```

# Dictionary Type Fundamentals

**What:** If you think of lists as ordered collections of objects, you can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are stored and fetched by *key* instead of by *positional offset*.

**How and why:** Dictionaries take the place of records, search tables, and any other sort of aggregation where item names are more meaningful than item positions.

Snippet 4.15 — initializing a new dictionary object

```
# method 1
>>> D = {
    "Captain Marvel": 3,
    "Living Tribunal": 2,
    "One-Above-All": 1
    }
# method 2
>>> CHARACTERS = [
    "Captain Marvel",
    "Living Tribunal",
    "One-Above-All"
    ]
>>> RANK = [3, 2, 1]
>>> D = dict(zip(CHARACTERS, RANK))
>>> print(D)
{"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
```

# Popular Dictionary Methods

| Method | Synopsis |
|---|---|
| `D.keys()` | Get a list of all keys in a dictionary |
| `D.values()` | Get a list of all values in a dictionary |
| `D.items()` | Get a list of all key-value pairs in a dictionary |
| `D.get(key, default)` | Get the value of a key in a dictionary, or a default value if the key is not present |
| `D.update(D2)` | Update a dictionary with the key-value pairs from another dictionary |
| `D.copy()` | Get a copy of a dictionary |
| `D.clear()` | Remove all key-value pairs from a dictionary |

# Sample Dictionary Methods in Action

Snippet 4.18 — accessing the informaton included in a dictionary

```
# the dictionary
>>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
# let us change the power rank for Captain Marvel
>>> D["Captain Marvel"] = 12
>>> print(D)
{"Captain Marvel": 12, "Living Tribunal": 2, "One-Above-All": 1}
# let us eliminate the character Living Tribunal
>>> del D["Living Tribunal"]
>>> print(D)
{"Captain Marvel": 12, "One-Above-All": 1}
# let us add a further character
>>> D["Wanda Maximoff"] = 4
>>> print(D)
{"Captain Marvel": 12, "One-Above-All": 1, "Wanda Maximoff": 4}
```

# Tuple Type Fundamentals

**What:** Tuples are sequences of immutable Python objects. They are similar to lists, but they are immutable. Tuples are created by enclosing a comma-separated list of values in parentheses. **How and why:** Tuples are useful for storing data that is not to be changed, such as the coordinates of a point in a two-dimensional space. In general, we use tuples any time information integrity is a concern — in other words when we want to ensure the information included in an object will not change because of another reference in our program.

Snippet 4.19 — creating and accessing a table

```
# the tuple
>>> T = ("Captain Marvel", 3)
# access a tuple element
>>> T[0]
"Captain Marvel"
# access a tuple element
>>> T[1]
3
```

# Set Type Fundamentals

**What:** A `set` is an *unordered* collection of *unique* and *immutable* objects.

**How and why:** Sets made this way support common mathematical set operations. Hence, they have a variety of applications, especially in numeric and database-focused work.

Snippet 4.23 — set operations

```
# create two sets
>>> X = set(["a", "b", "c"])
>>> Y = set(["c", "d", "e"])
# set difference
>>> X - X
set()
>>> X - Y
{"a", "b"}
# union
>>> X | Y
{"a", "b", "c", "d", "e"}
# intersection
>>> X & Y
{"c"}
# superset
>>> X > Y
False
# subset
>>> X < Y
False
```

# Outline

# Reading and Writing Data through Pipes

**What:** Your Python program may read data from a file stored in your machine and/or write the outcome of your analysis to a file.

**How and why:** You open a pipe to a file using the built-in function `open`. The output of the function is a `file` object.

Snippet 4.24 — data input with open

```
# create a pipe to a file
>>> file = open(file="my_file.txt", mode="r")

# calling "file" yields the attributes of the file object
>>> file
<_io.TextIOWrapper name="my_file.txt" mode="r" encoding="UTF-8">

# let us source the data
>>> data = file.read()
>>> print(data)
Hi there

# close the pipe
>>> file.close()
```

# File Methods

| Method | Description |
|---|---|
| `file.close()` | Closes the file |
| `file.detach()` | Returns the separated raw stream from the buffer |
| `file.fileno()` | Returns a number that represents the stream as per the OS' perspective |
| `file.flush()` | Flushes the internal buffer |
| `file.isatty()` | Returns whether the file stream is interactive or not |
| `file.read()` | Returns the file content |
| `file.readable()` | Returns whether the file stream can be read or not |
| `file.readline()` | Returns one line from the file |
| `file.readlines()` | Returns a list of lines from the file |
| `file.seek()` | Change the file position |
| `file.seekable()` | Returns whether the file allows us to change the file position |
| `file.tell()` | Returns the current file position |
| `file.truncate()` | Resizes the file to a specified size |
| `file.writable()` | Returns whether the file can be written to or not |
| `file.write()` | Writes the specified string to the file |
| `file.writelines()` | Writes a list of strings to the file |

# Outline

# Python Statement Fundamentals

In simple terms, Python statements are *"the things you write to tell Python what your program should do"*. (Lutz, 2013)

# Control Flow

**What:** Many Python statements we write are compound statements: there is one statement nested inside another. The outer statement is called the 'if' statement, and the inner statement is called the 'then' statement.

**How and why:** The 'if' statement determines whether to execute the 'then' statement. Specifically, the 'then' statement is executed insofar as the 'if' statement evaluates to 'True.'

Snippet 4.28 — an example of control flow

```
# a set with a customer's past purchases
>>> S = set(["a", "x", "u"])

# a rule-based product recommender
>>> if "x" in S:
...     print(
...          "Customers who bought x also bought Air" \
...          "Jordan 7 Retro Miro"
...          )
... else:
...     pass
Customers who bought x also bought Air Jordan 7 Retro Miro
```

# Outline

# While and For Loops

**What:** Oftentimes, we write Python statements that repeat the same task — i,e. they loop a certain number of times or over multiple items.

**How and why:** The `while` statement provides a way to code general loops. The `for` statement is designed for stepping through the items in a sequence or other iterable object and running a block of code for each.

Snippets 4.31/32 — while and for loop examples

```
# loop until reaching a numeric threshold
>>> i = 0
>>> while i <= 3:
...     print(i)
...     i = i + 1
0
1
2
3
# run a mathematical operation on a list of items and
# append the outcome to a second list
>>> input = [2, 8, 1]
>>> output = []
>>> for item in input:
...     output.append(item + 1)
>>> print(output)
[3, 9, 2]
```

# Iterations and Comprhensions

**What:** As we know from the previous section, `while` and `for` loops can handle most repetitive tasks programs need to perform. One of the most prominent tools is `list comprehension`.

**How and why:** List comprehnsions make loops *easier to write/read* and *more efficient*. In practice, we include a Python statement containing a for loop among brackets.

Snippets 4.38 — nested for loops Vs. list comprhension

```
# the for loop way
# --+ create an empty list
L = []
# --+ create a for loop appending the square of some items
>>> for i in range(3):
...     L.append(i ** 2)
# --+ print the list
>>> print(L)
[0, 1, 4]

# the list comprehension way
>>> L = [i ** 2 for i in range(3)]
>>> print(L)
[0, 1, 4]
```

# At the End of the Chapter, You Will Be Able to...