A Crash Course in Python

Alessandro Leite

October 11th, 2019

Goal

- Learn the essential concepts of the Python language ecosystem.

- **Key topics**:
  1. Installing Python/Jupyter on Windows, Linux and macOS
  2. Python basics:
     - Identifiers, expressions, and statements
     - Control flow
     - Loop
     - Data structures: lists, tuples, and dictionaries
     - Manipulating files
     - Modules
     - Randomness
     - Manipulating relational databases
     - Working with JSON files

Outline

1. **Identifiers, expressions, and statements**
   - Identifiers and keywords
   - Operators
2. **Control flows**
   - Iterators
   - Defining functions
3. **Data Structures**
   - Lists
   - Tuples
   - Dictionaries
4. **Working with files**
5. **Modules**
6. **Randomness**
7. **Relational Databases and SQLite**
8. **JavaScript Object Notation**

What is Python?

- Python is an interpreted, high-level, general-purpose programming language
- It was Created by Guido van Rossum and first released in 1991
- Python's design philosophy emphasizes code readability
  - See *The Zen of Python* (*python.org/dev/peps/pep-0020*)
  - Or execute in Python

```
import this
```

- Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects

Identifiers

- **Identifiers** are used to name `variables`, `functions`, classes in Python
- Must start with a letter or underscore
- Must consist of letters, numbers, and underscores
- Are case sensitive

Valid:	spam	_speed	
Invalid:	23spam	var.name	
Different:	spam	Spam	SPAM

Python's keywords

- Keywords cannot be used as identifiers

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Arithmetic operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder
//	Exact division (i.e., floor division)

- Expressions are evaluated from left to right
- **Precedences rules**
  - Parenthesis
  - Exponentiation
  - Multiplication, division, and remainder
  - Addition and subtraction
- In Python 3, **integer division** results in a **floating point** result

Comparison operators

Operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
>	Greater than
>=	Greater than or equal to
!=	Not equal

Logical operators

Operator	Description
and	Returns **True** if both statements are **true**
or	Returns **True** if one of the statements are **true**
not	Negate a statement; i.e., reverse the result of a statement

Identity operators

Operator	Description
is	Returns **True** if two variables point out to the same object
is not	Returns **True** if two variables do not point out to the same object

Membership operators

Operator	Description
in	Returns **True** if a value is present in the a specific set (e.g., list)
not in	Returns **True** if a value is **absent** of a specific set

Bitwise operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of them is 1
^	XOR	Sets each bit to 1 if only one of them if 1
$\sim$	NOT	Inverts all the bits
$\ll$	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
$\gg$	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Constants

- Constants are fixed values (i.e., immutable values) such as numbers and strings
- Strings can be defined using `single quote` ('...') or `double quotes` ("...")
- Variables, literals, and constants have a **type**
- Python knowns the difference between an **integer** and a **string**
- For example, **+** means addition if one of the operands is a number and a **concatenation** if it is a **string**
- We can ask Python what type something is by using the **type()** function
- Numbers can be **integers** or **floating point** numbers

String conversions

- We can use **int()** and **float()** to convert between string, integer and floating point values
- Python **raises** an **exception** if the string does not be converted to a number

User input

- We can read data from the user using the **input()** function
- The **input()** function returns a **string**

```
name = input("What is your name?")
print("Welcome ", name)
```

If statements

```
if <expression> :
   if body
elif <expression> :
   elif body
else:
   else body
```

- There can be zero or more **elif** parts
- The **else** part is optional
- The keyword **elif** is short for *else if*, and is useful to avoid excessive indentation
- An **if** . . . **elif** . . . **elif** . . . sequence is a substitute for the **switch** or **case** statements found in other languages (e.g., Java, C, C++)

Indentation

- Python uses indentation to delimit blocks of code
- This makes Python code readable
- We need to be very careful with our code formatting
- While spaces are ignores inside parentheses and brackets

```
x = float(input("Please enter a number "))
if x < 0:
  print("Value is negative")
elif x == 0:
  print("Value is zero")
else:
  print("Value is positive")
```

The while statement

- The **while** statement comprises a form to iterate in Python

```
while <expression>:
    body
```

- Approach
  1. Evaluate the expression, yield **True** or **False**
  2. If the expression if **False**, exit the while statement and continue the execution after **while** statement
  3. If the expression is **True**, execute the **body** and then go back to step 1
- A body can comprise one or more statements
- We can stop a **while** statement through the **break** expression
- We can “skip” some execution using the **continue** expression

For statements

- Sometimes we want to iterate through a set of things, such as list of words, the lines of a file, or a list of numbers
- When we have a list of things to iterate through, we usually use a **for** statement
- a **while** statement is known as indefinite loop because it simply loops until some condition becomes **False**, whereas the **for** loop iterates through a known set of items

```
names = ['Sophie', 'Bia', 'Alice']
for name in names:
    print('Hello ', name)
```

Number sequence iteration

- To iterate over a sequence of numbers, we use the function **range()**
- The **range()** function generates arithmetic progressions
- For example, **range(5)** generates 5 values (i.e., 0, 1, 2, 3, 4)
- The given end point is never part of the generated sequence
- It is possible to let the range start at another number, or to specify a different increment value. For instance

```
for x in range(0, 10, 2):
    print(x)
```

- To iterate over the indices of a sequence, we can combine the use of the functions **range()** and **len()**

```
words = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(words)):
  print(i, words[i])
```

- In some context, it is convenient to use the **enumerate** function

```
list(enumerate(words))
```

Functions

- There are two kinds of functions in Python:
  1. **built-in functions** – functions provided as part of core of the language such as `print()`, `input()`, `type()`, `float()`, `int()`, among others.
  2. **user-defined functions** – functions defined by the developers

Function definition

Function

- It is reusable code that can take one or more arguments, does some computation, and then returns a result
- We define a function using the keyword **def**
- It must be followed by the function name and the list of arguments inside the parenthesis.

```
def square(x):
  """Computes and returns the square of x"""
  return x ** 2
```

- The first statement of the function body can optionally be a string literal
- It comprises the function's documentation, or *docstring*

Functions can have optional arguments

```python
def ask_ok(prompt, retries=4, reminder='Please try again!'):
  while True:
    ok = input(prompt)
    if ok in ('y', 'ye', 'yes'):
      return True
    if ok in ('n', 'no', 'nop', 'nope'):
      return False
    retries = retries - 1
    if retries < 0:
      raise ValueError('Invalid user response!')
    print(reminder)
```

- It can be called in different ways:
  - giving only the mandatory argument

    ```python
    ask_ok('Do you really want to quit?')
    ```

  - giving one of the optional arguments

    ```python
    ask_ok('OK to overwrite the file?', 2)
    ```

  - giving all the arguments

    ```python
    ask_ok('OK to overwrite the file?', 2, 'Come on, only
        yes or no!')
    ```

  - giving only the required argument and the last one

    ```python
    ask_ok('OK to overwrite the file?', reminder = 'Please,
        only yes or no!')
    ```

Lambda Expressions

- Lambda functions are small anonymous functions created through the keyword **lambda**
- Lambda functions can be used wherever function objects are required
- They are syntactically restricted to a single expression

```
def make_incrementor(n):
    return lambda x: x + n
```

- This function returns another function

```
f = make_incrementor(10)
f(1)
```

Void and fruitful functions

- When a function does not return a value, we call it a `void` function
- Functions that return a value are called `fruitful` functions
- Void functions are "`non-fruitful`" functions

The are some advantages to create functions

- Functions enable us to organize our code into paragraphs
- Each paragraph (i.e., function) captures a complete through
- Allows us to avoid repetitions
- Promotes reuse – make it work once and the reuse it
- To break long or complex concepts into logical chunks and to encapsulate them across different functions

Working with strings

- Strings can be indexed with the first character having index equals to 0
- Indices may also be negative numbers, to start counting from the right
- In addition to indexing, slicing is also supported
- Indexing is used to obtain individual characters, whereas slicing allows you to obtain substring

```
word = "Mary"
word[0] # M
word[-1] # y
word[0:3] # Mar
```

- The start index is always included, and the end is always excluded

Common string functions

Function	Description
len	returns the number of characters in a string
strip	removes while spaces from the beginning and end of a string
format	performs a string formatting operation. The string can contain literal text or replacement field delimited by curly braces {}
find	returns the lowest index in the string where a substring is found

- We can get a list of all available built-in functions of string, by executing

```
dir(str)
```

A list is the most fundamental data structure in python

- Like a string, a **list** comprises a sequence of values
- In a string, the values are characters; whereas in a list, they can be any type
- The values in list are called **elements** or sometimes **items**
- The simplest way to create a list is to enclose the elements in square brackets (`[` and `]`)
- We can create an empty list with empty brackets, `[]`
- Lists can contain another lists

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
names = ["Alice", "Sophie", "Mary"]
empty = []
values = [cheeses, names]
```

Lists are mutable

- The syntax for accessing the elements of a list is the same as for accessing the characters of a string —the bracket operator
- The expression inside the brackets specifies the index
- Unlike strings, lists are mutable because we can change the order of items in a list or reassign an item in a list

```
names[1] = "Elsa"
```

- Any integer expression can be used as an index
- If we try to read or write an element that does not exist, we get an `IndexError`
- The `in` operator also works on lists

```
names = ["Alice", "Elsa", "Mary"]
name = input('Give a name ')
if name in names:
  print("The name {} was found".format(name))
else:
  print("Unknown name {}".format(name))
```

Traversing a list

- The common way to traverse a list is with a `for` loop

```
for name in names:
  print(name)

numbers = range(0, 10)
for i in range(len(numbers)):
   numbers[i] = numbers[i] * 2
```

- Although a list can contain another list, the nested list still counts as a single element

```
names = ["Alice", "Sophie", "Mary"]
ages = [9, 12, 20]
people = [cheeses, names]

print(len(people))
```

- The length of the list *people* is 2

Concatenating lists

- We can use either use the + operator to concatenate two lists or the method `extend` of lists

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
cheeses.extend(['Roquefort', 'Brie'])
print(cheeses)

cheeses += ['Feta', 'Mozzarella', 'Burrata']
print(cheeses)
```

- The slice operator also works on lists

```
cheeses[:2]
```

- A slice operator on the left side of an assignment can update multiple elements of a list

```
options = ['a', 'b', 'c', 'd', 'e', 'f']
options[1:3] = ['x', 'y']

print(options)
```

List methods

- The method `append` adds a new element to the end of a list

```
cheeses = ['Cheddar', 'Edam']
cheeses.append('Mozzarella')
print(cheeses)
```

- `sort` arranges the elements of the list from low to high

```
cheeses = ['Cheddar', 'Mozzarella', 'Burrata', 'Edam', '
    Feta']
cheeses.sort()

print(cheeses)
```

- If we don't want to change our list, we can use the `sorted` function, which returns a new list

```
cheeses = ['Cheddar', 'Mozzarella', 'Burrata', 'Edam', '
    Feta']
cheeses_sorted = sorted(cheeses)

print(cheeses)
print(cheeses_sorted)
```

Deleting elements

- There are several ways to delete elements from a list
- If we know the index of the element you want to delete, we can use `pop` method

```
numbers = [1, 5, 8]
number = numbers.pop(1)

print(numbers) # [1, 8]
print (number) # 5
```

- `pop` modifies the list and returns the element that was removed
- If we don't provide an index, it deletes and returns the **last** element

Deleting elements (cont.)

- If we don't need the removed value, we can use the operator `del`

```
numbers = [1, 5, 8]
del numbers[1]
print(numbers) # [1, 8]
```

- If we know the element to remove, but not the index, we can use method `remove`

```
numbers = [1, 5, 8]
numbers.remove(5)
print(numbers) # [1, 8]
```

- We can remove more than one element, you can use `del` with a slice index

```
numbers = [1, 5, 8]
del numbers[0:2]
print(numbers) # [8]
```

List comprehensions

- Frequently, we need to transform a list into another one, by choosing only certain elements
- A way to do this in Python is through list comprehensions

```
even_numbers = [x for x in range(5) if x % 2 == 0]
     squares = [x * x for x in range(5)]
even_squares = [x * x for x in even_numbers]
```

- A list comprehensions can include multiple loops:

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]
```

Working with tuples

- Tuples are another kind of sequence that functions much like a list
- Tuples have elements which are indexed starting at 0
- Pretty much anything we can do to a list that doesn't change its state, we can do to a tuple
- We can specify a tuple by using parentheses or nothing instead of square brackets

```
cheeses = ('Cheddar', 'Mozzarella', 'Burrata')
names = ('Alice', 'Elsa', 'Mary')
```

- Tuples are a convenient way to return multiple values from a function

```
def sum_and_product(x, y):
  return (x + y), (x * y)
```

- Unlike a list, once we created a tuple, we cannot alter its contents

Tuples are more memory efficient

- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- Thus, in our program when we are making “temporary variables”, we prefer tuples over lists

Tuples are comparable

- The comparison operators work with tuples and other sequences (e.g., lists)
- If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ

```
numbers = (1, 5, 8)
print (numbers < (6, 10, 15))
```

Working with dictionaries

- A dictionary is another fundamental data structure in Python
- It associates values with keys
- It enables us to quickly retrieve the value corresponding to a given key
- Dictionaries are like lists except that they use keys instead of numbers to look up values

```
grades = dict() # initialize an empty dictionary
grades['Alice'] = 'A'
grades['Elsa'] = 'B'

grades = {"Alice": 'A', 'Elsa': 'B'}
```

- We can look up the value for a key using square brackets

```
print(grades['Alice'])
```

- We get a *KeyError* if we ask for a key that is not in the dictionary
- Dictionaries have a *get* method that returns a default value when we look up for a key that is not in the dictionary

```
print(grades.get('Mary', None))
```

Retrieving a list of keys and values of a dictionary

- We can get a list of keys of a dictionary through its method **keys**

```
grades = {"Alice": 'A', 'Elsa': 'B'}
names = grades.keys()
```

- We can also get a list of values through the method **values**

```
grades = {"Alice": 'A', 'Elsa': 'B'}
grades_values = grades.values()
```

- Dictionaries have a method called `items` that returns a list of tuples, where each tuple is a key-value pair

```
grades = {"Alice": 'A', 'Elsa': 'B'}
items = grades.items()
```

- Combining `items`, tuple assignment, and `for`, we can traverse the keys and values of a dictionary in a single loop

```
grades = {"Alice": 'A', 'Elsa': 'B'}
for name, grade in grades.items():
  print ("Student: {}, grade: {} ".format(name, grade))
```

Opening a file

- Before we can read the contents of the file, we must tell Python which file we are going to work with and what we will be doing with the file
- This is done with the **open()** function
- The **open(filename, mode)** function returns a *file handle*, which is a variable used to perform operations on the file
- handle = open(filename, mode)
- mode is optional and can be `'r'` if we are planning only to read the file or `'w'` if we are going to write to the file

```
handle = open('romeo-and-juliet.txt', 'r')
```

Counting the number of lines in a file

- Open a file in read-only mode
- Use a for loop to read each line
- Count the lines and print out the number of lines

```
handle = open('romeo-and-juliet.txt', 'r')
count = 0
for line in handle:
  count += 1

print('File has {} lines '.format(count))
```

Reading the whole file

- We can read the whole file into a single string with the method **read()**

```
handle = open('romeo.txt', 'r')
text = handle.read()
print(len(text))
```

Time for Action
Macro
Shift

Implementing a word count

- Write a Python code that give a text file, reads it and:
  1. Shows the number of unique occurrence of each word in the file
  2. Shows the top-10 words used in the text

Working with modules

- Certain features of Python are not loaded by default
- It includes both features included as part of the language as well as third-party features
- To use these features, we need to `import` the **modules** that contain them
- One approach comprise in import the module itself

```
import math
x = float(input('Enter a positive real number'))
math.sqrt(x)
```

- In this case, `math` is the module containing mathematical functions
- It is also possible to import the need functions explicitly to use the without qualification

```
from math import sqrt
x = float(input('Enter a positive real number'))
sqrt(x)
```

Generating pseudo-random numbers

- Python provides a modules called `random` to produce pseudo-random numbers
- The numbers are generated based on an internal state
- We can control the state through a `seed` to get reproducible results

```
import random
random.seed(123)
uniform_randoms = [random.random() for _ in range(4)]
print (uniform_randoms)
```

- We can use the method `choice` to randomly pick one element of a list

```
friends = ['Alice', 'Sophie', 'Elsa', 'Alain']
best_friend = random.choice(friends)
```

- We can use `random.sample` to choose a sample of elements without replacement

```
numbers = range(60)
winning_numbers = random.sample(numbers, 6)
```

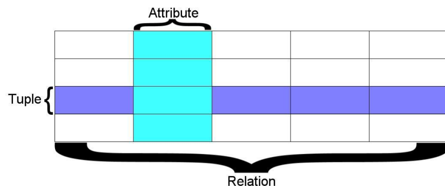
Relational databases

- Store data on rows and columns in tables
- They power rely in its ability to efficiently retrieve data from those tables and in particular where there are multiple tables and the relationships between those tables involved in the query

Terminology

- **Database** – contains many tables
- **Table or relation** – contains tuples and attributes
- **Tuple or row** – comprises a set of fields (i.e., columns) that generally represents an “object” like a person or a music track
- **Attribute also known as column or field** – comprise one of the possibly many elements of data corresponding to the object represented by the row

Relational model



- A relation is defined as a set of tuples that have the same attributes
- A tuple usually represents an object and information about that object
- Objects are typically physical objects or concepts.
- A relation is usually described as a table, which is organized into rows and columns
- All the data referenced by an attribute belong to the same domain and conform to the same constraints

Structured Query Language

- Structured Query Language (SQL) is the language we use to manipulate a database
- Through SQL we can:
  1. Create a table
  2. Retrieve the data
  3. Insert and update data
  4. Delete data

Database model

- A database model or database schema is the structure or format of a database described in a formal language supported by the **database management system.**
- In other words, a **database model** is the application of a data model when used in conjunction with a database management system
- Examples of database systems comprise:
  - Oracle – large, commercial, enterprise-scale, very very tweakable
  - MySQL – simpler but very fast and scalable - commercial open source
  - SQLite, Postgres, HSQL comprise other open sources database systems

SQLite

- SQLite is a popular database
- It is free and fast and small
- It is embedded in Python and a number of other languages
- It can be manipulate through the SQLite Browser (*sqlitebrowser.org*)

Creating a database table

- The code to create a database file and a table named `tracks` with two columns in the database is

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS tracks ')
cur.execute('CREATE TABLE tracks (title TEXT, plays INTEGER)')

conn.close()
```

- The `connect` connects to the database and stores in the file `music.sqlite3` in the current directory
- A **cursor** is like a file handle that we can use to perform operations on the data stored in the database
- Calling `cursor()` is conceptually similar to calling `open()` when dealing with text files in Python

Storing data onto a table

- We can add new data into a table using the SQL `INSERT` operation

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('INSERT INTO tracks (title, plays) VALUES ( ?,
    ? )', ( 'Thunderstruck', 20 ) )
cur.execute('INSERT INTO tracks (title, plays) VALUES ( ?,
    ? )', ( 'My Way', 15 ) )
conn.commit()
cur.close()
```

Retrieving the data of a table

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

rows = cur.execute('SELECT title, plays FROM tracks')

for row in rows:
   print (row)

conn.close()
```

Working with JSON

- JavaScript Object Notation (JSON) is a standard file format that uses human-readable text to transmit data objects consisting of attribute-value pairs and array data types
- It is used for storing and exchanging data independent of programming language and platform
- Python has a built-in package called `json`, which can be used to create and read data in JSON format

```
import json
data = '{ "name":"Elsa", "age":15, "city":"Australia"}'
people = json.loads(data)

print(people['name'])
```

Time for Action
Macro
Shift

Dealing with database and JSON data

- Every hour, the Paris City Hall publishes new data about the bicycles available for usage (i.e., Velib) at *bit.ly/2EkdKjZ*
- Write a Python function that reads the data and then insert them into an SQLite database