Introduction to Python

Oscar Dalmau

Centro de Investigación en Matemáticas

August 21, 2020

Outline

1. Overview
2. Basic Syntax
3. User Input/Output formatting
4. Data Types and Operations
   - Variables
   - Python's Core Data Types
   - Basic Operators
   - Data Types
     - Numbers
     - Strings
     - Lists
     - Tuples
     - Dictionaries
     - Sets
     - Files
5. List comprehensions

What is Python?

Python is a high-level, interpreted, interactive and object-oriented scripting language.

- Python is Interpreted: You do not need to compile your program before executing it.
- Python is Interactive: One can interact with the interpreter or the prompt directly to write your program
- Python is Object-Oriented: i.e. Python supports Object-Oriented style

Some Python Features

- Portable: Python can run on a wide variety of hardware platforms and has the same interface on all platform
- Extendable: You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- Databases: Python provides interfaces to all major commercial database
- GUI Programming

Some Python Features

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large application
- Supports automatic garbage collection.
- It can be integrated with C, C++ and Java.

Python Environment

Python is available on a wide variety of platforms: Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.), Win 9x/NT/2000, Macintosh (Intel, PPC, 68K) etc

- Install Python?
- Running Python.
  - Interactive Interpreter

```
 $python   # Unix/Linux
or
C:>python # Windows/DOS
```

Python Environment

Python is available on a wide variety of platforms including Linux and Mac OS X: Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.), Win 9x/NT/2000, Macintosh (Intel, PPC, 68K) etc

- Install Python?
- Running Python.
  - Script from the Command-line:

```
$python script.py    # Unix/Linux
C:>python script.py # Windows/DOS
```

Python Environment

Python is available on a wide variety of platforms including Linux and Mac OS X: Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.), Win 9x/NT/2000, Macintosh (Intel, PPC, 68K) etc

- Install Python?
- Running Python.
  - IDE: IDLE, Spyder, etc (Unix/Macintosh), PythonWin, Spyder, etc (Windows)
  - Jupyter: is an acronym meaning Julia, Python and R (now the notebook technology supports many programming languages).

Jupyter

- IPython Notebooks ('Jupyter') is web-based technology
- Jupyter notebooks are a series of 'cells' containing executable code, or markdown.
- Jupyter has LaTeX support for mathematical equations with MathJax (a web browser enhancement for display of mathematics).
- The notebooks can be saved and shared in .ipynb JSON format.

Jupyter

- Notebooks can be committed to version control repositories such as git and the code sharing site github.
- Jupyter notebooks can be viewed with nbviewer technology that github supports, see https://www.youtube.com/watch?v=eYVCH61fKyY.
- Jupyter offers code completion and easy access to help.
- See shortcuts_notebook_jupyter.ipynb

Python Environment

- Run interactive example: print 'Hello world!'
  - Create the helloworld.py file

```
#!/Users/osdalmau/anaconda/bin/python
 print "Hello, World!"
```

  - Create an executable file and run it

```
$ chmod +x helloworld.py
$./helloworld.py
```

  - Script from the Command-line:

```
$python helloworld.py     # Unix/Linux
```

Basic Syntax

- **Python Identifiers**: A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- **Reserved Words**: and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

Basic Syntax

- **Lines and Indentation**: There are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation. The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.
- **Multi-Line Statements**: Python allows the use of the line continuation character (\), but statements contained within the [], {} or brackets do not need to use the line continuation character
- **See** example1basicsyntax.py

Basic Syntax

- **Quotation in Python**: Python accepts single ('), double (") and triple (''' or """) quotes to denote string literal The triple quotes can be used to span the string across multiple lines.
- **Comments in Python**: A hash sign (#) that is not inside a string literal begins a comment.
- **Blank Lines**: A line containing only whitespace, possibly with a comment, is known as a blank line and Python ignores it.
- **Multiple Statements on a Single Line**: The semicolon (;) allows multiple statements on the single line
- **See** example2basicsyntax.py

Basic Syntax

- **Multiple Statement Groups as Suites**: A single code block is called a suite. Statements, such as if, while, def, and class require a header line and a suite. Header lines begin the with the keyword and terminate with a colon (:) and are followed by a code block (the suite)

```
if expression :
 suite
elif expression :
 suite
else :
 suite
```

Basic Syntax

- **Accessing Command-Line Arguments**: The Python sys module provides access to any command-line arguments (using sys.argv):
  - sys.argv: List of command-line arguments, sys.argv[0] is the program or the script name.
  - len(sys.argv): Number of command-line argument
- **See** example3basicsyntax.py

Print

```
>>> a = int(input('numero: '))
>>> print('numero: ', a)
>>> print('numero: {0:4d} {1:4d} {2:4d} {0:4.2f}'.
      format(a,a**2, a**3))
>>> print('numero:  {0:4d}  {1:10.2f} '.format(3,4)
>>> print("numero %.2f, %2d" % (a,a))
>>> print('numero binario:  {0:b}
     numero octal: {0:o} '.format(10))
>>> print('numero binario:  {0:b}
     numero octal: {0:o} '.format(10))
```

See Print.ipynb

Outline

1. Overview
2. Basic Syntax
3. User Input/Output formatting
4. Data Types and Operations
   - Variables
   - Python's Core Data Types
   - Basic Operators
   - Data Types
     - Numbers
     - Strings
     - Lists
     - Tuples
     - Dictionaries
     - Sets
     - Files
5. List comprehensions

Variables

Variables are memory locations to store value. When we create a variable one reserves some space in memory

- Assigning Values to Variables:

```
counter = 100     # An integer assignment
miles   = 1000.0  # A floating point
name    = "John"  # A string
```

- Multiple Assignment:

```
>>> x = y = z = 10
>>> a, b, txt = 1,4,'hello'
```

Outline

1. Overview
2. Basic Syntax
3. User Input/Output formatting
4. Data Types and Operations
   - Variables
   - Python's Core Data Types
   - Basic Operators
   - Data Types
     - Numbers
     - Strings
     - Lists
     - Tuples
     - Dictionaries
     - Sets
     - Files
5. List comprehensions

The Python Conceptual Hierarchy

Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. Expressions create and process objects.

Python objects

1. Objects are also known as **data structures**.
2. Python provides powerful object types as an intrinsic part of the language.
3. Usually, we don't need to code object implementations before you start solving problems.
4. It is almost always better to use a built-in object rather than implementing your own.

Standard types

In programming, data type is an important concept.
Variables can store data of different types, and different types can do different things.
Python has the following data types built-in by default, in these categories:

Table: Built-in Data Types

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview

Standard types

- Numbers: Python supports four numerical types i.e., int, float and complex

Standard types

- String: Strings are identified as a contiguous set of characters in between quotation marks (single or double).
- Subsets of strings can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.

Standard types

Python has five standard data types: Numbers, String, List, Tuple and Dictionary.

- List: Lists are the most versatile of Python's compound data type. A list contains items separated by commas and enclosed within square brackets ([]).
- A list can be of different data type.
- The values stored in a list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.

Standard types

Python has five standard data types: Numbers, String, List, Tuple and Dictionary.

- Tuple: Tuples contain items enclosed within parentheses (()) and separated by comma
- Tuples are read-only Lists

Standard types

Python has five standard data types: Numbers, String, List, Tuple and Dictionary.

- Dictionary: Dictionaries are kind of hash table type and consist of key-value pairs.
- A dictionary key can be almost any Python type, but are usually numbers or string Values can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ({}) and values can be assigned and accessed using square braces ([]).
- **See** Strings_Lists_Tuples_Dictbasic.ipynb

Data Type Conversion

Function	Description
int(x [,base])	Converts x to an integer. base specifies the base if x is a string, e.g., int('100',2)==6
chr(x)	Converts an integer to a Unicode character, e.g., print chr(345) return ř
ord(x)	Converts an integer to a Unicode character, e.g., print ord(u 'ř') return 345
complex(real [,imag])	Creates a complex number.

Data Type Conversion

Function	Description
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuple
eval(str)	Evaluates a string and returns an object.
str(x)	Converts object x to a string representation.
chr(x)	Converts an integer to a character.

Other functions: float(x), tuple(s), list(s) ,set(s), hex(x), oct(x)

See exampleOfconversion.ipynb

Outline

1. Overview
2. Basic Syntax
3. User Input/Output formatting
4. Data Types and Operations
   - Variables
   - Python's Core Data Types
   - Basic Operators
   - Data Types
     - Numbers
     - Strings
     - Lists
     - Tuples
     - Dictionaries
     - Sets
     - Files
5. List comprehensions

Basic Operators

Operator Type	Operators
Arithmetic	$+, -, *, /, **, \%, //$
Comparison	$==, !=, >, >=, <, <=$
Assignment	$=, +=, -=, *=, /=, **=, \%=, //=$
Bitwise	&, |, ^(xor), ~, <<, >>
Logical	and, or, not
Membership	in, not in
Identity	is, is not (compare the memory locations of two objects, i.e., id(x)==id(y))

See Operators.ipynb

Operators Precedence

Operator	Description
$**$	Exponentiation (raise to the power)
$\sim$	Bitwise one complement
$*,\ /,\ \%,\ //$	Multiply, divide, modulo and floor division
$+,\ -$	Addition and subtraction
$>>,\ <<$	Right and left bitwise shift
$\&$	Bitwise 'AND'
$\hat{},\ \mid$	Bitwise 'XOR' and regular 'OR'
$<=,\ <,\ >,\ >=$	Comparison operators

Operators Precedence

Operator	Description
$<>$, $==$ $!=$	Equality operators
in, not in	Memmership operators
not, or, and	Logical operators

Outline

1. Overview
2. Basic Syntax
3. User Input/Output formatting
4. Data Types and Operations
   - Variables
   - Python's Core Data Types
   - Basic Operators
   - Data Types
     - Numbers
     - Strings
     - Lists
     - Tuples
     - Dictionaries
     - Sets
     - Files
5. List comprehensions

Number

- Number data types store numeric value They are immutable data types which mean that changing the value of a number data type results in a newly allocated object. Number objects are created when you assign a value to the variable

```
>>> a = 3; print '(', a, ',',  id(a), ')'
>>> a = 4;  print '(', a, ',',  id(a), ')'
```

- You can also delete the reference to a number object by using the del statement

```
del var1[,var2[,var3[....,varN]]]
```

Number Type Conversion

- Type int(x) to convert x to a plain integer.
- Type float(x) to convert x to a floating-point number.
- Type complex(x) to convert x to a complex number with real part x and imaginary part zero.
- Type complex(x, y) to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Built-in Numeric Tools

Python provides a set of tools for processing number objects:

- Expression operators: $+, -, *, /, >>, **$, ..etc.
- Built-in mathematical functions: pow, abs, round, int, hex, bin, etc.
- Utility modules: random, math, etc.

See: Python Standard Library
$https://docs.python.org/3/library/$

Mathematical Functions

acos	asinh	atanh	cos	e	exp
frexp	hypot	ldexp	log10	pi	sin
acosh	atan	ceil	cosh	erf	expm1
fsum	isinf	lgamma	log1p	pow	sinh
asin	atan2	copysign	degrees	erfc	fabs
gamma	isnan	log	modf	radians	sqrt
factorial	tan	floor	tanh	fmod	trunc

```
>>> import math
>>> math.cos(math.pi)
>>> dir(math) # getting help
>>> help(math) # getting help
```

See Numbers.ipynb

Strings

- Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quote Python treats single quotes the same as double quote

```
>>> a = 'Hello'
>>> a = "Hello World!"
```

- They are our first example of a sequence, ie, a positionally ordered collection of other object Strictly speaking, strings are sequences of one-character strings; other, more general sequence types include lists and tuples.

Strings: sequence operations

- We can access to elements of the string (charactes and/or substring) by using [] bracket, and the indexing and slicing operation.
- Python offsets start at 0 and end at one less than the length of the string.
- Python also lets you fetch items from sequences such as strings using negative offset. You can also think of negative offsets as counting backward from the end.

Strings: sequence operations

- Indexing (S[i]) fetches components at offset The first item is at offset 0, S[0]. Negative indexes mean to count backward from the end or right.
- Slicing (S[i:j]) extracts contiguous sections of sequence The upper bound is noninclusive.
- Extended slicing (S[i:j:k]) accepts a step (or stride) k, which defaults to +1. Allows for skipping items and reversing order, S[::-1]

Strings

- Every string operation is defined to produce a new string as its result, because strings are immutable in Python, i.e., they cannot be changed in place after they are created. In other words, *you can never overwrite the values of immutable objects*
- You can change text-based data in place if you either expand it into a list of individual characters and join it back together

```
>>> S = 'shrubbery'
>>> L = list(S) # convert in a list
>>> L[1] = 'c'  # change a character
>>> ''.join(L)  # join
```

String Methods

Note: The list is not exhaustive

capitalize	ljust	splitlines
upper	rjust	rsplit
lower	split	rstrip
count	isdigit	isspace
isalnum	isupper	find
isalpha	islower	
isdecimal	join	

String Methods

```
>>> S = '1xxx,CIMAT,xxxx,CIMAT,xxxx'
>>> S.startswith('1xxx')
>>> where = S.find('CIMAT') # Search for position
>>> S.replace('CIMAT', 'CimatGto')
```

String Methods

```
>>> S.capitalize()
>>> S.lower()
>>> S[0].isdigit()
>>> S.split(',')
```

String Methods

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
>>> S.rjust(len(S) + 10)
```

Lists

- Lists are Python's most flexible ordered collection object type.
- Unlike strings, lists can contain any sort of object: numbers, strings, and even other lists.
- Lists may be changed in place by assignment to offsets and slices, list method calls, deletion statements, and more, i.e., they are mutable objects.

Common list operations

```
>>> L = [] # An empty list
>>> L = [123,'abc',1.23,{}] #Four items:indexes 0..
>>> L = ['Bob', 40.0, ['dev', 'mgr']] #Nested subli
>>> L = list('spam') # convert into a list
>>> L = list(range(-4, 4))
>>> L[i] # index
>>> L[i][j] # indexing nested lists
>>> L[i:j]  # Slicing
>>> len(L) # length
>>> L1+L2 #Concatenate, repeat, i.e., L=2*L1+3*L2
>>> 3 in L
```

Common list operations

```
L = [1,2,3]
L.append(4)  # Methods: growing, L = [1,2,3,4]
L.extend([5,6]) # L=[1,2,3,4,5,6]
L.append([5,6]) # It is different,L=[1,2,3,4,[5,6]]
L.insert(i, X) # insert object X before index i
L.index(X) # return first index of value
L.count(X) # return number of occurrences of value
L.sort() # sorting, reversing, *IN PLACE*
L.reverse() # reverse *IN PLACE*
L.pop(i) # remove and return item at index
L.remove(X) # remove first occurrence of value
```

Common list operations

```
del L[i]
del L[i:j]
L[i:j] = []
L[i] = 3  # Index assignment
L[i:j] = [4,5,6] #slice assignment
```

Tuples

s

Dictionaries

- Dictionaries are one of the most flexible built-in data types in Python.
- If lists are ordered collections of objects, you can think of dictionaries as unordered collections where items are stored and fetched by key, instead of by positional offset.
- Dictionaries also sometimes do the work of records or structs
- You can change dictionaries in place by assigning to indexes (they are mutable), but they don't support the sequence operations that work on strings and lists. Because dictionaries are unordered collections

Common dictionary operations

```
D={} # Empty dictionary
D={'name': 'Bob', 'age': 40} # Two-item dictionary
E={'cto':{'name': 'Bob','age':40}}# Nesting
D=dict(name='Bob',age=40) #Alternative construction
D=dict([('name', 'Bob'), ('age', 40)])#keywords, ke
D=dict(zip(keyslist, valueslist)) #zipped key/value
D=dict.fromkeys(['name', 'age'])#key lists
D['name']          # Indexing by key
E['cto']['age'] #
'age' in D # Membership: key present test
```

Common dictionary operations

```
D.keys() #          Methods: all keys,
D.values() # all values,
D.items() # all key+value tuples,
D.copy()#          copy (top-level),
D.clear()#          clear (remove all items),
D.update(D2)#          merge by keys,
D.get(key)#fetch by key, if absent default (or None
D.pop(key)#remove by key, if absent default (or err
D.setdefault(key, default?)#fetch by key, if absent
D.popitem()#remove/return any (key, value) pair; et
len(D)#Length: number of stored entries
```

Common dictionary operations

```
D[key]=42#Adding/changing keys
del D[key]#Deleting entries by key
list(D.keys())#Dictionary views (Python 3.X)
D1.keys() & D2.keys()#
```

Set data structure

- **Set** is a useful data structure. Sets behave mostly like lists with the distinction that they can not contain duplicate values. It is really useful in a lot of cases.
- **Some functions**: union, intersection, difference, clear, pop, remove, isdisjoint, issubset, issuperset

Set data structure

```
>>> s = set([]) % s = set()
>>> len(s)      % 0

>>> s1 = set([2,3,5,6])  or s1 = { 2,3,5,6}
>>> s2 = set([3,4]).       or s2 = {3,4}
>>> s3 = s1.union(s2)  s3 = {2,3,4,5,6}
>>> s4 = s1.intersection(s2)  s4 = {3}
>>> s5 = s1.difference(s2)  s5 = {2, 5, 6}
>>> s1.pop()  return 2 and remove it from s1,
                          s1 = {3,4,5}
>>> s1.clear()  s1= set()
```

See: DataTypesAndMethods.ipynb

Files-write

```
>>> f = open('prueba.txt', 'w')
>>> f.write('linea 1 \ n')
>>> f.write('linea 2 \ n')
>>> f.write('linea 3 \ n')
>>> f.close()
```

Files-read

```
f = open('prueba.txt', 'r')
data = f.read()
f.close()
print(data)
```

Files-read

```
>>> f = open('prueba.txt', 'r')
>>> print (f.readline())
>>> print (f.readline())
>>> f.close()
```

Files-read

```
>>> f = open('prueba.txt', 'r')
>>> lines = f.readlines()
>>> f.close()
>>> print(lines)
```

see: Files.ipynb

List comprehensions

1. In addition to sequence operations and list methods, Python includes a more advanced operation known as a **list comprehension expression**
2. **List comprehension** allows us to build a new list by running an expression on each item in a sequence, one at a time, from left to right.
3. For example:

```
>>> L  = [1,2,3]
>>> L1 = [i**2 for i in L]
```

List comprehensions

```
[f(x) for x in iterable]
[f(x) for x in iterable if condition]
[f(x) if condition else g(x) for x in iterable]
[f(x) if cond1 else g(x) for x in iterable if cond2
```

List comprehensions: Examples

```
#  Repeat characters in a string
>>> doubles = [c * 2 for c in 'spam']
>>> even = [x  for x in range(10) if x%2==0]
      # [0, 2, 4, 6, 8] even numbers between 0 and
>>>  [x  if  x%2==0 else x**2 for x in range(10) ]
      # [0, 1, 2, 9, 4, 25, 6, 49, 8, 81]
>>> [x.strip() for x in '  x , y , z  '.split(',')]
      #['x','y','z']
```

List comprehensions: Examples

```
List of character ordinals
>>> [ord(x) for x in 'spaam']
     #  [115, 112, 97, 97, 109],
Sets remove duplicates
>>>{ord(x) for x in 'spaam'},
     #  {112, 97, 115, 109}
>>> {x: ord(x) for x in 'spaam'}
     # {'p': 112, 'a': 97, 's': 115, 'm': 109}
>>> (ord(x) for x in 'spaam')
     # generator object <genexpr> at 0x000000000254
```

see: ListComprehension.ipynb