A Short and Incomplete Introduction to Python

Part 1: Basic Python syntax

Sigve Haug <sigve.haug@math.unibe.ch>,
Alexander Kashev <alexander.kashev@math.unibe.ch>
Science IT Support (ScITS), University of Bern

Based on a course by Riccardo Murri / Sergio Maffioletti
S3IT: Services and Support for Science IT, UZH

August 20–21, 2018

Python basics

Lines of Python code

Python code is interpreted line by line.

A line is terminated simply by pressing "Enter".
Python does not use semicolons or other separators.

A long line can be split in two by ending it with the character '\'; for example:

```
In [1]: "hello" + \
   ...: " world!"
Out[1]: 'hello world!'
```

Reference:

http://docs.python.org/reference/lexical_analysis.html#line-structure

Comments

Anything following the pound sign '#' is a comment and is not executed.

It's used to leave notes to yourself or others in code, or temporarily disable parts of code.

```
# This is a comment
print("Hello, world!") # Basic example

# Code below won't be executed
# print("Goodbye, world!")
```

Writing good comments is important for understanding code later!

Data and basic types

Python can operate on different **types** of data, e.g. numbers, strings, booleans (true/false), lists of other data, etc.

To add those to your code directly (as constants), you need **literals**.

The next slides show examples for some basic types.

Number literals

Python operates with **integers** and **floating point** numbers (floats).

Integers in Python 3 can be arbitrarily large, while floating point numbers are subject to precision limits.

A basic integer literal is just writted down as a number:
`-1`, or `1234`, or `10000000000000000000`

A number is understood as a floating point literal if it has a decimal dot in it:
`3.1415`, or `10.0`, or even `1.` and `.005`

Reference: Built-in Types: Numeric Types

String literals, I

Strings are the data type for text: sequences of individual characters. There are several ways to express string literals in Python.

Single and double quotes can be used interchangeably to delimit strings:

```
"a string"
'a string'
```

String literals, II

You can use the single quotes inside double-quoted strings, and vice versa:

```
"Isn't it ok?"
'"Yes", he said.'
```

Or, you can use **escape sequences** to prevent quotes from ending the string:

```
In [2]: print("\"It's unnecessarily complex,\" he thought.")
Out[2]: "It's unnecessarily complex," he thought.
```

Other useful escape sequences:
`\\` for \, `\n` for a line break, `\t` for a tab stop

String literals, III

Multi-line strings are delimited by three quote characters.

```
"""This is a string,
that extends over more
than one line.
"""
```

You do not need not use the backslashes "\" at the end of the lines to join them into one Python line.

Operators, arithmetic

All the arithmetic operators are defined in Python:
`+`, `-`, `*`, `/`, `**` (exponentiation x^y), etc.

Usually, the result of the operation is a float if at least one argument is.

Python's division always produces a float.
If you need integer division, you can use `//` for the result and `%` for the remainder:

```
In [3]: 11 / 3
Out[3]: 3.6666666666666665
In [4]: 11 // 3
Out[4]: 3
In [5]: 11 % 3
Out[5]: 2
```

Booleans and logic

Boolean (truth) values in Python are expressed using constants `True` and `False`.

Logical operators are expressed using plain English words: `and`, `or`, `not`.

Boolean values often come from numerical and string comparison: `<`, `>`, `<=`, `==`, `!=`, ...

Reference:

- http://docs.python.org/library/stdtypes.html#boolean-operations-and-or-not
- http://docs.python.org/library/stdtypes.html#comparisons

Your first exercise

How much is 2^{144} ?

(You have 1 minute time.)

Operators, II

Some operators are defined for non-numeric types:

```
>>> "Py" + 'thon'
'Python'
```

Some support operands of mixed type:

```
>>> "a" * 2
'aa'
>>> 2 * "a"
'aa'
```

Some do not:

```
>>> "aaa" / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Variables and assignment

You can't do much programming with just constants. To manipulate data, you need to store it somewhere.

Variables are names for pieces of data in memory. When used, they are substituted with their value.

Names must start with a letter, and can contain letters, numbers and underscores '_'.

You can assign **values** to them with the assignment operator =, and use them in other expressions:

```
In [6]: a = 11 # Assigning 11 to a
In [7]: b = 3
In [8]: c = a - b # Using saved values a and b
In [9]: c
Out[9]: 8
```

Assignment, II

Trying to use a variable that was never assigned produces an error.

You can reassign values to already declared variables. Python is said to have *loose typing:* a variable can point to any data, regardless of its type.

```
In [10]: a = 11
In [11]: a = "orange"
In [12]: a = True
```

For binary operators, you can use assignment shortcuts, for example:

`a = a + b` is the same as `a += b`.

Best practice: variable naming

While it's fast to write shorter variable names, it's important to make your code easy to understand.

Making longer but more meaningful names is recommended, especially if the variable is used in distant parts of the code. Compare:

```
In [13]: a = w * h
In [14]: area = width * height
```

Which is easier to read?

String interpolation

The `.format()` method can be used to substitute values into placeholder strings.

Placeholders can indicate substitutions by number (starting at 0):

```
>>> "This is slide {0} of {1}".format(20, 1001)
'This is slide 20 of 1001.'
```

You can use names instead of numbers (then the order parameter occur in `format()` does not matter):

```
>>> "Today is {month} {day}".format(day=2, month='March')
'Today is March 2'
```

Reference: https://pyformat.info/

String interpolation

The `.format()` method can be used to substitute values into placeholder strings.

Placeholders can indicate substitutions by number (starting at 0):

```
>>> "This is slide {0} of {1}".format(20, 1001)
'This is slide 20 of 1001.'
```

You can use names instead of numbers (then the order parameter occur in `format()` does not matter):

```
>>> "Today is {month} {day}".format(day=2, month='March')
'Today is March 2'
```

Reference: https://pyformat.info/

String interpolation

The `.format()` method can be used to substitute values into placeholder strings.

Placeholders can indicate substitutions by number (starting at 0):

```
>>> "This is slide {0} of {1}".format(20, 1001)
'This is slide 20 of 1001.'
```

You can use names instead of numbers (then the order parameter occur in `format()` does not matter):

```
>>> "Today is {month} {day}".format(day=2, month='March')
'Today is March 2'
```

Reference: https://pyformat.info/

Exercise

Exercise 1.B:
Save your weight (in kg) and height (in m) into variables.
Calculate your BMI and save it into a variable.

Hint: $\text{bmi} = \frac{\text{weight}}{\text{height}^2}$

Use `.format()` to display "My BMI is <number>"

Basic types

Basic object types in Python 3:

None	Special type with one value `None`.
bool	Truth values: `True`, `False`.
int	Integer numbers: `1`, `-2`, ...
float	Double precision floating-point numbers, e.g.: `3.1415`, `-1e-3`.
bytes	String of byte-size characters.
str	Text (string of UNICODE characters).
list	Mutable list of Python objects
dict	Key/value mapping

The type of a Python object can be gotten via the `type()` function:

```
In [3]: type('hello')
Out[3]: str
```