CM1101 Computational Thinking

Introduction to Python

Dr Kirill Sidorov
SidorovK@cardiff.ac.uk
www.facebook.com/kirill.sidorov

School of Computer Science and Informatics
Cardiff University, UK

Python

- Powerful, high-level language with excellent means of abstraction.
- Easy to get started!
- Named after Monty Python and the flying circus.
- Main teaching language in CM1101 and CM1103.

Python

Where is Python used?

`en.wikipedia.org/wiki/List_of_Python_software`

- Desktop applications and development tools: BitTorrent, Blender 3D, Dropbox, Mercurial, Bazaar…
- Games: Civilization IV, Eve Online, World of Tanks, Battlefield 2…
- Web: Django, Google App Engine, web2py…
- Science: NumPy, SciPy, Sage, Matplotlib…

Python is a very versatile language!

Recommended resources

- “Think Python 2e” — excellent free book:
  `http://greenteapress.com/wp/think-python-2e/`
- Excellent free online book of exercises:
  `http://learnpythonthehardway.org/book/`
- The “official” tutorial:
  `docs.python.org/3/tutorial/index.html`
- Language and standard library reference:
  `docs.python.org/3/reference/index.html`
  `docs.python.org/3/library/index.html`

High-level languages

- We write programs (in programming languages) “for” computers.
- But the real purpose of the programming language is to be readable and easily understandable by humans.
- High-level languages — provide better means of abstraction and composition $\rightarrow$ easier for humans to use (read, understand, write, design, compose).
- Low-level languages — provide poor levels of abstraction and composition $\rightarrow$ harder for humans to use.
  - But easier to translate to machine instructions.
  - *E.g.* the Assembly Language is basically mnemonic names for machine instructions, and not much more.

Compilers and interpreters

- Python is a high-level programming language.
- A program (in a high-level language) must be first translated into machine instructions before being executed.
- Compiler — a program that translates an entire high-level language program into machine instructions.
- Interpreter — does the same, but a little bit at a time.
    - Observe that philosophically they do the same thing; the difference is only practical.
- Python is an interpreted language. Python programs are executed by an interpreter.

REPL

Two ways to use an interpreter:

- Read-evaluate-print loop (REPL) = interactive mode
  - The interpreter executes Python commands as you enter them and immediately outputs the results (if any).
- File mode = non-interactive mode = batch mode
  - The interpreter executes an entire program stored in a file (or files).
  - By convention, we add the extension `.py` to the names of files that store Python programs.
  - A program can be stored in a file(s) and executed whenever you want to.
- REPL is useful for playing with the interpreter, but for long programs we use the non-interactive mode (file mode).

REPL

Using Python as an interactive calculator.

```
>>> 2+3
5
>>> 4*7
28
>>> 2 ** 5
32
>>> 4+3*2
10
>>> quit()
```

Values and data types

- All values belong to a certain data type:
  - `5` → integer (`int`)
  - `"Hello, World!"` → string (`str`)
  - `4.33` → floating point number (`float`)
- Data type determines:
  - How the values of this type are stored in memory.
  - What are the possible values.
  - What operations can preformed on values of this type.

Values and data types

In Python, you can find the type of any value using the command `type()`.

For example, these are some of the types that Python knows:

```
type(5) → <type 'int'>

type("Hello, World!") → <type 'str'>

type(4.33) → <type 'float'>

type(True) → <type 'bool'>
```

Variables

- A variable is a human-friendly name that refers to a value.
- An assignment statement creates new variables and assigns the values (initialises them), for example:

```
name = "Kirill Sidorov"
year = 2014
temperature = 36.6
```

Naming variables

- Try to choose meaningful names.
  - Variables exist for the benefit of the human!
- Ideally, describing what the variable is used for.
- In Python, variable names may contain letters, digits, and underscore.
- But, they MUST begin with a letter.
- Python is case-sensitive!
  So, `pitch` is not the same as `Pitch`

Naming variables

- Try to choose meaningful names.
  - Variables exist for the benefit of the human!
- Variable names can be arbitrarily long.
- They can contain multiple words.
- To use multiple words, separate each word with the underscore character: _

Example:

```
post_code = "CF24 3AA"
total_mark = 75
speed_of_light = 299792458
```

Reserved keywords

Python has some reserved keywords which have a special meaning, and cannot be used as variable names. Here they are:

```
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
```

Valid variable names

To summarise, the permitted variable names may contain letters, digits, underscore. Must begin with a letter. Cannot be the same as one of the reserved keywords.

Example: is there anything wrong with the following variables names?

```
lambda = 500

person@reception = "Matthew Strangis"

10Forward = 42
```

Operators

- These are symbols which denote operations to be undertaken on values or variables.

- A few examples:

+ - / * % < > <= >= **

- The operator `%`, for instance, is used to find the remainder after division:

```
17 % 5 = 2
```

- The operator `**` is used to compute powers (exponentiate):

```
3 ** 5 = 243
```

Order of operations

- When more than one operator is used in an expression, the order in which the prescribed operations will be carried out is important!

  Example:

  3 * 2 + 7 % 3 = ?

- Every language, including Python, has a set of rules that describe in what order are the operators evaluated. This is called operator precedence or order of operations.

Order of operations

- Below (some of) the operators are shown in descending order of precedence:

```
()
**
* / %
+ -
```

- Operators higher up in this list will be executed first.
- Operators on the same level in this list will be executed from left to right.

3 * 2 + 7 % 3 = 7

Order of operations

```
()
**
* / %
+ -
```

The rules can be remembered with PEMDAS:

- Parantheses have the highest precedence. What is inside the brackets is evaluated first: `4 * (4 - 2)` = 8.
- Exponents (powers), then
- Multiply, Divide, (and Remainder), then
- Add or Subtract.
- Otherwise just from left to right.

Booleans

- Logical (Boolean) values are used to represent the logical concepts of a statement being “true” of “false”.
- In Python, Boolean values are denoted as: `True` and `False`. (Note the capital first letter.)
- Python supports the following logical operations: `and`, `or`, `not`.

Logical operations

A	not A
False	True
True	False

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

There is no `xor` operation in Python. How would you go about implementing this operation in terms the ones you already know?

A bit of logic

Example

- *R* means “It is raining”
- *U* means “I have an umbrella”
- *W* means “I can go for a walk”

```
W = (not R) or U
```

A bit of logic

Example

- *A* means “Age is less than 18”
- *Y* means “Looks young”
- *I* means “Has a valid 18+ ID”
- *C* means “Can enter club”

```
C = (not A) and (I or (not Y))
```

but also:

```
(not I) or (not A) == True
```

(`I` implies `not A`)

Example

```
>>> not True
False
>>> False and True
False
>>> (not False) and (False or True)
True
>>> (False or True) and (False or (True and True))
True
>>> ((not False) and (not True)) or
    ((not True) and (not False))
False
```

Boolean values

- Certain operations in Python evaluate to Boolean values.
- A good example is comparison operations: <, >, == (equals), != (not equals), <=, >=.
- Precedence (higher to lower): arithmetic, comparisons, `not`, `and`, `or`.

Example

```
>>> 4 > 5
False
>>> (12 % 5) < 5
True
>>> 3 + 4 == 4 + 3
True
>>> ((1 > 2) or (3 < 4)) and (5 <= 5)
True
>>> (2 < 5) == (3 < 4)
True
```

Converting between data types

Often, you need to convert from one data type to another. Here are some examples:

```
>>> str(12)
'12'
>>> str(3.14)
'3.14'
>>> int('42')
42
>>> float('2.71')
2.71
```

What about converting to and from booleans?

Strings

- Strings are just sequences of characters.
- There are lots of useful commands in Python to return information about and to manipulate strings.

Example:

`len("Kirill")` $\rightarrow$ 6.

`"Kirill".count("l")` $\rightarrow$ 2.

Strings

- We can use double quotes to denote a string:
  `"Hello"`
- Or single quotes:
  `'Hello'`
- Or triple quotes (“docstrings”):

```
"""Here
is some
text
"""
```

- What is wrong here? How can we fix this?
  `"She said "Hi" and smiled"`

String concatenation

- Concatenation — joining two strings together, end to end.
- Use the + operator to concatenate strings.

Example:

```
"Kirill" + "Sidorov"     →     "KirillSidorov"
```

Accessing individual characters

- Remember, strings are just sequences of characters.
- The subscription operator `[]` is used to retrieve individual characters.

Example:

Consider a variable called `my_string` which contains the string `"Kirill"`. To find the third character of this string we would use:

`my_string[2]` $\rightarrow$ "r".

- We count from `0`!

Splitting a sentence

- If you have a sentence that you want to split into individual words, you would use the `split()` command.

Example:

Consider a variable called `my_sentence` which contains the string `"Kirill Sidorov 2017"`. To split the sentence into words we would use:

```
words = my_sentence.split()
```

- `split()` returns a list of words:

In this case: `['Kirill', 'Sidorov', '2017']`

Strings

- `s.lower()`, `s.upper()` returns the lowercase or uppercase version of the string.
- `s.strip()` returns a string with whitespace removed from the start and end.
- `s.isalpha()`, `s.isdigit()`, `s.isspace()` tests if all the string chars are in the various character classes.
- `s.startswith('other')`, `s.endswith('other')` tests if the string starts or ends with the given other string.
- `s.find('other')` searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found.
- `s.replace('old', 'new')` returns a string where all occurrences of `'old'` have been replaced by `'new'`.

Etc. See reference for more: `https://docs.python.org/3/library/stdtypes.html#string-methods`