BCH2203 Python for Biochemistry: 2. Lists and input/output

Ramses van Zon

Winter 2024

The `if` statement

The following code gives a "Good morning!" greeting if `hour` is less than 12.

```
>>> hour=11
>>> if hour < 12: print("Good morning!")
Good morning
>>> hour=16
>>> if hour < 12: print("Good morning!")
>>>
```

It does so for two case: `hour=11` and `hour=16`.

- `if` followed by a condition and a colon will execute what is **after the colon** if the condition is met.

Multi-line input

If we have **more than one statement** to do when the condition is met:

```
>>> hour=7
>>> if hour < 12:
...     print("Good morning!")
...     print("Would you like some coffee?")
...
Good morning
Would you like some coffee?
>>>
```

- *How did we enter a multiline piece of Python code at the >>> prompt?*
- Well, python noticed that the statement

  `if hour < 12:` requires more code, so it enters *multi-line mode*.
- Multi-line mode is indicated by `...`, and you can now continue typing code.
- To indicate that your multi-line input has been completed and should

  be interpreted and executed, you press Enter on an empty line.

Code blocks

If we have **more than one statement** to do when the condition is met:

```
>>> hour=7
>>> if hour < 12:
...     print("Good morning!")
...     print("Would you like some coffee?")
...
>>>
Good morning!
Would you like some coffee?
>>>
```

How does Python know which lines are part of the **statements that are executed** upon the condition being true?

- The `if` statement takes the next

  code block.
- The start of a code block is indicated by an **indented line**.
- That code block includes all lines that are indented in the same way.

User input with the `input()` function

- With the `input` function, we can ask the user to type in a value and store it in a variable.

```
>>> s=input()
...
>>> print(s)
...
```

- You can pass a string to it, which becomes the prompt from that input:

```
>>> s=input("Give a number: ")
Give a number: ...
>>> print(s)
...
```

- Regardless of the inputted value, the type of value that `input()` returns is always a string.

  You'd have to convert it yourself to a number of that's what you'd expect, e.g.

```
>>> s_as_int = int(s)
>>> s_as_float = float(s)
```

Scripting

- Okay, so we typed in the `input()` command, then we typed in a value (say '5'), and then that value was stored in `s` as a string.
- Why did we not just store the value in `s` (`s='5'`), then?
- The idea is that the input could be given by some other user, but since they'd have to be sitting right next to us as we are coding, they could type the `s='5'` statement.

```
>>> s=input("Give a number: ")
Give a number: ...
>>> print(s)
...
```

- We have arrived at a point where the interactive session has lost its utility.
- We want to create something that will execute the python commands elsewhere without typing them in interactively.
- An app, if you wish.

Script=program=application=app

- As far as python is concerned, they are all the same.
- It's something you can run and which performs a function.
- With running, we mean here typing "`python` *SCRIPTNAME*" on the command line, with *SCRIPTNAME* replaced by the name of your script.

Creating python scripts

- Creating a python script is as simple as storing the commands into a text file.
- Choose your editor, ensure it can save as 'plain' ASCII text. No .doc or .rtf, please.

  E.g., nano, emacs, vi, vim, sublime text, gedit, notepad, vscode, ...
- Make sure you understand where your editor saves your files.
  - Either save your file in the directory where your terminal is.
  - Or change directory in the terminal to the directory where you editor saves your files.
- Creating, editing, saving a text file differs per system.

The `else` statement

We saw the `if` statement last lecture, which executes a code block based on a condition.

What if the condition is not true, and we need a ***different set of statements*** to be executed?

Use "`else:`".

```
>>> hour=7
>>> if hour < 12:
...     print("Good morning!")
...     print("Would you like some coffee?")
... else:
...     print("Good afternoon!")
...     print("No more coffee for you!")
...
Good morning!
Would you like some coffee?
>>>
```

More conditions with "`elif`"

```
>>> hour=19
>>> if hour < 12:
...     print("Good morning!")
...     print("Would you like some coffee?")
... elif hour < 17:
...     print("Good afternoon!")
...     print("No more coffee for you!")
... else:
...     print("Good night!")
...
Good night!
>>>
```

Python's error messages

Because we cannot foresee every possible error, let's look at a typical uncaught Python error.

```
>>> print 17
  File "<stdin>", line 1
    print 17
          ^
SyntaxError: Missing parentheses in call to 'print'
```

Read the lines in the error messages carefully:

1. Something's up in line 1 in a file "<stdin>", i.e., the prompt.
2. The statement with the issue is printed, here, it is `print 17`.
3. The ^ more precisely pinpoints where there's an issue
4. The last line is most informative: there should have been parentheses in the call to 'print'.
   The type of this error is a 'SyntaxError'.

Python's error messages

Let's look at another error message:

```
>>> print(seventeen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'seventeen' is not defined
```

Read the lines in the error messages carefully:

1. *What's a traceback?*

   When the error occurs in the execution step, several function may be called before the error, and the traceback would show these.

2. Here, the error occurs in line 1 in a file "`<stdin>`", i.e., the prompt, but before a function has been called, i.e., on the "`<module>`" level.

3. Again, the last line is most informative: the variable seventeen has not been defined (yet?).
   The type of this error is a 'NameError'.

Python's error messages

Here's another one:

```
>>> print(11+'17')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

What was going on here?

A `TypeError`:

a and b are of different types and cannot be added by the operator +.

Repetitions/Loops

- Rather than failing if the user didn't enter an integer, it would be nice to try again.
- Loops are repetitions of a code block for different, given cases, or until a condition is fulfilled.
- So we could 'loop' until the entered string is an integer.

While loop

This is one way:

```
haveint=False
while not haveint:
  s=input("Give me an integer: ")
  try:
   i=int(s)
   haveint=True
  except:
   print("That is not an integer, try again!")
print(i)
```

- At the start of the while loop, `haveint` is checked, and python enters the the code block that belongs to `while` (the "body of the loop")
- If `i=int(s)` succeeds, `haveint` is set to True.
- `haveint` is checked at the next iteration.
- `print(i)` is outside the loop body.
- Note: it is possible to exist a loop somewhere in the middle of the loop body with the `break` keyword.

Lists

- A list is a collection of objects (e.g. integers, strings, floats..)
- We create a list by putting objects between square parentheses `[]`, separated by commas, e.g.,

```
>>> lst = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 'blast off!']
```

- List elements do not all have to be the same type.
- List elements can be lists themselves.

What can we do with these lists?

- We can access elements using the notation `LISTNAME[INDEX]`.

  E.g.:

```
>>> lst = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 'blast off!']
>>> print(lst[0])
10
>>> print(lst[10])
blast off!
```

  The first element has index 0.

- We can reassign elements of the list, too:

```
>>> lst[10] = 'abort'
>>> print(lst)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 'abort']
```

What can we do with these lists?

- Add an element to the end with append method:

```
>>> lst.append('not ready')
>>> print(lst)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 'abort', 'not ready']
```

- Remove an element by index with the pop method:

```
>>> lst.pop(2)
8
>>> print(lst)
[10, 9, 7, 6, 5, 4, 3, 2, 1, 'abort', 'not ready']
```

Note that the removed element is returned by pop.

- Obtain the length of the list:

```
>>> print(len(lst))
11
```

- From strings, you can lists with the split function:

```
>>> s = "These are words in one string"
>>> s.split(" ")
["These", "are", "words", "in", "one", "string"]
```

Loops with lists

- It is rather common to have to go over a list and do something with every element.
- This is a repetition, i.e., a loop.
- We could write

```
i = 0
while i<len(lst):
   x = lst[i]
   do_something_with(x)
   i = i + 1
```

- But python can do that with a `for` loop:

```
for x in lst:
   do_something_with(x)
```

Modules

Using Modules

- In addition to built in functions and command, you can also get additional functionality in Python using modules.
- Before you can start using that functionality, you have to `import` the corresponding modules.

  E.g.

```
import sys
import biopython
import matplotlib.pyplot
```

You can change the namespace that the modules functions end up in

- Changing the name of the module: `import numpy as np`
- Importing specific functions: `from numpy import ones`
- Importing everything: `from numpy import *`

Using Modules, continued

- There are many, many, many standard modules
- We will only get to look at a few very basic ones:
  - `sys`: system specific parameters and functions (command line arguments, exit, path, ...)
  - `os`: operating system stuff (chdir, stat, getenv, listdir, walk, ...)
  - `shutils`: High level file operations (copyfile, copytree, rmtree, move, ...)
- For more standard modules, see https://docs.python.org/3/library/index.html
- In addition, there are non-standard modules in the pypi repository, that you can install with the `pip` command on the terminal command line (not within python).

Python File I/O

Basic File Input and Output in Python

- Files contain your data
- Files are organized in directories or folders
- A directory is a file too
- Path: sequence of directories to get to a file

Python modules/packages for files

- built-in python file objects
- `os`, `os.path`
- `shutil`
- `pickle`, `shelve`, `json`
- `zipfile`, `tarfile`, ...
- `csv`, `numpy`, `scipy.io.netcdf`, `pytables`, ...

Basic (Text) File Input and Output in Python

- Get current directory:

  `os.getcwd()`

- Create directory:

  `os.mkdir('FOLDER1')`

- Change current directory:

  `os.chdir('FOLDER1')`

- Get file list:

  `os.listdir()`

- Get file list by wildcard pattern:

  `glob.glob('pattern')`

- Path manipulations: `os.path` module.

- Open file for read,write,read/write,append:

  `f = open('FOLDER1/WORLD.TXT','r') f = open('FOLDER1/WORLD.TXT','w') f = open('FOLDER1/WORLD.TXT','r+') f = open('FOLDER1/WORLD.TXT','a')`

- Write to file:

  `print(v,file=f)`, `f.write(s)`

- Read whole filer: `f.read()`, `f.readlines()`

- Read line by line: `f.readline()`

- Close file: `f.close()`

Summary

- This was a start of our overview of the Python language.
- Learning any (programming) language requires practice.
- This is where the weekly assignments come in.

Assignment 1: Survey Data Analysis

You are given the results of a survey which contained 10 yes-or-no questions.

The results are stored as lines in a text file called `2024-01-17-survey-results.txt`.
Each line contains one survey submission.
Each survey submission contain 10 characters, which can be either 'Y' or 'N', separated by commas.

Your task is to write a python script called `survey-analysis.py` to analyze the data in the following way:

- For each of the ten questions, compute and print the percentage of 'Y' answers.
- Also compute and print the number of survey submissions with zero 'Y' answers, then the number of surveys with one 'Y' answer, with two 'Y' answers, ... , and with ten 'Y' answers.

Your script should be submitted to the course site by January 26, 2024, at midnight.

Note: On the teach cluster, you can get the data file with the command

```
$ cp /scinet/course/bch2203/2024-01-17-survey-results.txt .
```