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

Ramses van Zon

Winter 2022

Scripts and user input

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, ...
- 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?
>>>
```

It's not the afternoon after hour=17!

Right, so let's fix that 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!
>>>
```

Errors

- Taking input from a user, and then validating it, is a rather big topic on its own.
- Your code should to some extent be prepared for thing to go wrong.
  ("defensive programming")
- E.g., what if `s=input()` is suppose to give a integer, the code does `int(s)`, but the input isn't integer? We get some funny error like:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '45.1'
```

- We will talk about Python error messages later, but users of your script do not wish to decipher that.

Error handling

- You could check if the string is in fact a number, as there's a function for that.
- It would look like this:

```
s=input("Give me an integer: ")
if s.isnumeric():
   i=int(s)
   print(i)
else:
   print("That is not an integer!")
```

- Good, but there may be other things that go wrong in the input that we did not catch.

- An alternative is the 'try first, deal with failure later' model: exceptions.
- This take the following form

```
s=input("Give me an integer: ")
try:
   i=int(s)
   print(i)
except:
   print("That is not an integer!")
```

- You can be more specific in the except on what kind of error you're catching, but let's not worry about that now.

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:

```
>>> a = 11
>>> b = '17'
>>> c = a + b
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

Try again

- In the read-an-int example, it would be nice to start over if the user didn't enter an integer.
- A 'go to beginning' statement does not exist in Python (no 'go-to's in fact), but loops are.
- Loops are repetitions of a code block for different, given cases, or until a condition is fulfilled.
- So we could 'loop' (do the same thing over and over again) until the entered string is an integer.
- This would be a `while` loop.

  (The other kind of loop is a `for`, which we will see later)

While loop

- In the read-an-int example, it would be nice to start over if the user didn't enter an integer.
- We could 'loop' until the entered string is an integer.

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.
- Note that `print(i)` is outside the loop body.

Escaping the loop

- If the expression after `while` is not true after the loop body is executed, the loop stops.
- The loop can also be stopped at any time in the loop body with the **`break`** keyword.

In both cases, execution of the script continues with the next non-indented line of code.

So instead of:

```
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)
```

We could also have used:

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

Note: break stops the loop, but not the script.
The `exit()` function can stop a script.

List

Lists

- A list is a collection of objects.
- We have not talked about objects before, but any data of all the types we have introduced count as objects: 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.
- Lists are objects, so 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!
```

  Note that the first element has index 0.

  (You can think of the index as an offset from the beginning of the list.)

- 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?

- You can 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']
```

- You can 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.
(`del lst[2]` would also have worked, but would not have returned removed element).

- The current length of the list is obtained from the `len` function.

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

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)
```

Other handy `list` manipulations

- `lst.index(value)`: Find the index of `value` in `lst`.
- `lst.count(value)`: Count number of occurrences of `value` in `lst`.
- `lst.extend(otherlist)`: Append all elements of otherlst.
- `lst.insert(index,object)`: Insert object in list at position `index`.
- `lst.remove(value)`: Remove the first element that is equal to value in `lst`.
- `lst.copy()`: Create a copy of ‘`lst`.
- `lst.clear()`: Remove all elements from `lst`.
- `lst.reverse()`: Reverse the `lst`.
- `element in list`: Check if `element` is in `lst`.
- `lst.sort()`: Put `lst` in sorted order.
- `sorted(lst)`: Create a sorted version of `lst`.

List comprehension

- List comprehensions are a short hand way of creating lists.
- They combine a for loop, appends, and if statements.
- Example: list of all squares that are divisible by 4 and are less than 100.

without list comprehensions:

```
squarelist=[]
for i in range(100):
    i2 = i**2
    if i2%4 == 0:
       squarelist.append(i2)
```

with list comprehensions:

```
squarelist=[i**2 for i in range(100) if i**2 % 4 == 0]
```

General syntax:

```
[ <EXPRESSION> for <VARIABLE> in <LIST-LIKE> if <CONDITION>]
```

The `if` is optional, e.g. `[i**2 for i in range(4)]` gives `[0,1,4,9]`.

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).
- These modules are put in a special location that python knows about, so they do not have to reside in the directory of your script.

  (note: If you ever end up with modules outside of this special location, you can append them to the `PYTHON_PATH` variable.)

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 file: `f.read()`, `f.readlines()`

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

- Get/set file pointer:

  `f.tell()`, `f.seek(position)`

- Close file: `f.close()`

Metadata

File metadata describes the file and its properties:

- File name
- File size
- Location on disk
- File type (though often through magic identifiers)
- Dates/times
- Read/write permissions
- ...

- Size:

```
>>> os.path.getsize('FOLDER1/WORLD.TXT')
```

- Permissions (linux)

```
>>> st=os.stat('FOLDER1/WORLD.TXT')
>>> st.st_mode & stat.S_IXUSR
>>> st.st_mode & stat.S_IWUSR
```

- Change, modification, access time

```
>>> st.st_ctime, st.st_mtime, st.st_atime
>>> datetime.fromtimestamp(round(st.st_mtime))
```

Content metadata

File content metadata described properties of the data stored in the file:

- What is the data?
- Where did it come from?
- Who made it/owns it/....?
- Format of the data
- Units
- ...

This type of metadata is not kept by the file system, but is very important for the long run. Adding content metadata requires some effort. You could use a separate file, but we will see that formats like NetCDF allow you to add metadata in the date file itself.

Minimizing I/O Operations

- Disk I/O is usually the slowest part of a pipe line.
- If manipulating data from files is most of what you do, try and minimize iops.

Bad

```
>>> s='Hi world\n'
>>> for c in s:
...     f=open('hiworld.txt','a')
...     f.write(c)
...     f.close()
```

Good

```
>>> s='Hi world\n'
>>>
>>> f=open('hiworld.txt','w')
>>> f.write(s)
>>> f.close()
```

- **Work in memory and reuse data if you can.**

Closing a file automatically

- Closing a file when you done flushes any buffers and ensures that what is written actually gets to disk.
- But it's easy to forget.
- The `with` statement can automatically close the file for you:

```
>>> with open('hiworld.txt','w') as f:
...   f.write('Hi world!\n')
...
```

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: Survey Data Analysis

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

The results are stored in a text file called `2022-01-19_survey-results.txt`, of which each line contains one survey entry.

Each survey entry is a string of 10 characters, which can be either 'Y' or 'N'
(the first character is the answer to the first question, the second character the answer to the second, etc.)

Your task is write a python script called `survey_analysis.py` to analyze the data.

In particular:

- Your script should read the file `2022-01-19_survey-results.txt` into a list.
- For each of the ten questions, it should print the percentage of 'N' and 'Y' answers.
- It should also print the number of surveys 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, 2022, at midnight.

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

```
$ cp /home/l/lcl_uotphy1610/lcl_uotphy1610s1959/2022-01-19_survey-results.txt .
```