Introduction to Python

Quantitative Applications for Data Analysis

Alexey Fedoseev

March 4, 2025

The PATH

When you type a command in your terminal, computer checks particular directories on your computer for the file with the name matching your command. If it finds it simply runs it, if not - you see

```
error: command not found.
```

For example, the command `ls` that we use all the time is actually a file on your computer. You can find where it is located using the command `which`.

```
$ which ls
/bin/ls
```

The PATH

How does the computer know where to look for the file? It checks the value of the specific variable `$PATH` set by the system. To check directories that are in the `$PATH` on your computer, use the command `echo`.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin
```

The directories are separated with the colon "`:`". The `$PATH` is searched from the beginning to the end, with the first matching executable being run. So directories at the beginning of `$PATH` take precedence over those that come later.

The software installer usually will prepend (i.e. add to the beginning) to the `$PATH` the new directory where the new software has been installed. However, you can also do it yourself.

```
$ export PATH="/Users/alexey/miniforge3/bin":$PATH
$ echo $PATH
/Users/alexey/miniforge3/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:
/Library/TeX/texbin
```

Installing Miniforge

Miniforge is a distribution of Python and R designed for scientific computing. It includes thousands of packages managed by the `conda` package management system.

You can download Miniforge here:

https://conda-forge.org/download/

Miniforge allows us to create a named, isolated, working copy of Python that maintains its own files, directories, and paths so that you can work with specific versions of libraries or Python itself without affecting other Python projects.

After the installation is finished, check whether `conda` was properly installed by checking its version:

```
$ conda -V
conda 24.11.3
```

Permanently adding to PATH

If the previous command gave you an error like this

```
-bash: conda: command not found
```

you need to manually prepend Miniforge to the `PATH`. Open the file `~/.bashrc` (`~/.zshrc` on newer Macs) or `~/.bash_profile` (depending which one you already have) using `nano` and add the following line at the end of the file (replace it with the directory where Miniforge is located on your computer)

```
export PATH="/Users/alexey/miniforge3/bin:$PATH"
```

After saving the change restart all terminal windows.

If you do not have `~/.bashrc` or `~/.bash_profile` files on your computer, simply create `~/.bash_profile` file with the aforementioned line.

```
$ cat ~/.bash_profile
export PATH="/Users/alexey/miniforge3/bin:$PATH"
```

Notes for Windows users (for Anaconda users)

Windows users do not have `bin` directory. Instead add the following lines into your `~/.bash_profile` when using Git Bash (replace it with the actual directory on your computer)

```
export PATH="/c/Users/scinet/Anaconda3":$PATH
export PATH="/c/Users/scinet/Anaconda3/Scripts":$PATH
```

After saving the file, restart the git bash windows. Test the installation in a new window.

```
$ source activate base
$ which python
/c/Users/scinet/Anaconda3/python
$ which conda
/c/Users/scinet/Anaconda3/Scripts/conda
$ python -i
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] ::
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
```

`python` versus `ipython`

There are two main ways to run Python interactively: regular Python and iPython ("Interactive Python"). They both have advantages and disadvantages:

- Regular Python is what you get when you type `python` at the command prompt.
- There aren't as many special features built into regular Python.
- But regular Python is what you get when you run Python scripts, so you're sure to get consistent behavior between your scripts and the Python command line.
- iPython has tab-line completion built-in, interactive plotting and other features.
- But iPython is not what you have when you run scripts.

Virtual environments

Python applications will often use packages and modules that don't come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library's interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution for this problem is to create a virtual environment, a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A's environment.

Starting Python

To start Python open a terminal and type `python` (or `ipython`). Use the command `exit()` to quit.

```
$ python
Python 3.12.8 | packaged by conda-forge | (main, Dec  5 2024, 14:19:53) [Clan
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python has a slightly different prompt than R, however, you can type the commands, computer will execute it and display the result right away.

Importing modules

Most of the default functions in Python are contained in packages. These are similar to “libraries” in R. There are several ways to import packages:

```
>>> import platform
>>> platform.system()
'Darwin'
>>> import time as t
>>> t.time()
1550679097.931794
>>> from calendar import isleap
>>> isleap(2020)
True
```

Variables in Python

Variables are reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory.

Variable names must be made up of only letters, numbers, and underscore (_). They cannot begin with a number and cannot contain dots or spaces!

```
>>> myvar = 10
>>> print(myvar)
10
```

Python data types

Python has several standard data types:

- Numbers
- Strings
- Booleans
- Container types
    - Lists
    - Sets
    - Tuples
    - Dictionaries

We aren't going to cover all of the data and container types, since we're focusing on Scientific data analysis.

Integers in Python 3

In Python 3, there is effectively no limit to how long an integer value can be. It is only constrained by the amount of memory your system has.

```
>>> import math
>>> math.factorial(500)
30605751221644063603537046129726862938858880417357699941677674125947653317
67168674655152914224775733499391478887017263688642639077590031542268429279
06974559841225476930271954604008012215776252176854255965356903506788725264
32189626429936520457644883038890975394348962543605322598077652127082243763
94491201286786753683057122936819436499564604981664502277165001851765464693
40112226034729724066333258583506870150169794168850353752137554910289126407
15715483028228493795263658014523523315693648223343679925459409527682060806
22328123873838817049600000000000000000000000000000000000000000000000000000
00000000000000000000000
```

Floating-Point Numbers

Floats have a decimal point and integers do not have a decimal point. So even though 4 and 4.0 are the same number, 4 is an integer while 4.0 is a float.

Before you start calculating with floats you should understand that the precision of floats has limits, due to Python and the architecture of a computer. Some examples of errors due to finite precision are displayed below.

```
>>> 1.13 - 1.1
0.029999999999999805
>>> 1 + .0000000000000001
1.0
```

Strings

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

```
>>> word = "Hello World"
>>> print(word)
Hello World
>>> print(word + " again!")
Hello World again!
```

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

```
>>> print(word[0], word[6])
H W
>>> print(word[6:11])
World
```

Booleans

Python supports standard boolean variables and operations.

Booleans behave as you would intuitively expect.

- "and" and "&" are the same.
- "or" and "|" are the same.
- "not" is written out (don't use the "!" symbol).

```
>>> truth = True
>>> truth and False
False
>>> not truth
False
>>> truth or False
True
>>> truth | False
True
```

Some notes

Some notes about Python features, in contrast to R.

- The assignment operator is the equal sign, as it should be.
- Variables cannot have periods in their names. Periods are part of the object-oriented syntax in Python.
- As with R, comments start with a `#` sign.
- For boolean operators:
  - Python uses `True`, not `TRUE`. Also, you must use the full word, not just `T`.
  - The `not` operator is written out (don't use "`!`").
  - The only time "`!`" is used is for the "not equal to operator" ("`!=`", the opposite of "`==`").
- Python is case sensitive (`A` is different from `a`).

Dynamic typing

Like R, Python uses dynamic typing which means you can re-use a variable over and over again.

```
>>> truth = 2
>>> print(truth)
2
>>> type(truth)
<class 'int'>
>>>
>>> truth = True
>>> print(truth)
True
>>> type(truth)
<class 'bool'>
```

Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

```
>>> mix_list = ["apples", 2, True, [4,10], False]
>>> print(mix_list)
['apples', 2, True, [4, 10], False]
```

List indices start at 0, and lists can be sliced. Slicing is done with `[start:finish]`, but does not include the `"finish"` element.

```
>>> print(mix_list[0])
apples
>>> print(mix_list[0:1])
['apples']
>>> print(mix_list[0:2])
['apples', 2]
```

Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the `append()` method.

```
>>> print(mix_list)
['apples', 2, True, [4, 10], False]
>>>
>>> mix_list[0] = "oranges"
>>> print(mix_list)
['oranges', 2, True, [4, 10], False]
>>>
>>> mix_list.append("books")
>>> print(mix_list)
['oranges', 2, True, [4, 10], False, 'books']
```

Dictionaries

A list is an ordered sequence of objects, whereas dictionaries are unordered sets. The main difference between lists and dictionaries is that items in dictionaries are accessed via keys and not via their position. Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type.

To access dictionary elements, you can use the square brackets along with the key to obtain its value.

```
>>> campus = {"name": "UTSC", "location": "Scarborough", "enrollment": 12980}
>>> print(campus)
{'name': 'UTSC', 'location': 'Scarborough', 'enrollment': 12980}
>>> print(campus["name"])
UTSC
```

Updating the dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown in a simple example given below.

```
>>> campus['name'] = 'University of Toronto Scarborough'
>>> campus['established'] = 1964
>>> del campus['location']
>>> del campus['enrollment']
>>> print(campus)
{'name': 'University of Toronto Scarborough', 'established': 1964}
>>>
>>> campus.keys()
dict_keys(['name', 'established'])
>>> campus.values()
dict_values(['University of Toronto Scarborough', 1964])
```

Writing scripts

Create the file todo.py with the following contents.

```python
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("day",
                    choices=["today", "tomorrow"],
                    help="Select the day to see tasks")
args = parser.parse_args()

toDo = {"today": "sleep"}
toDo["tomorrow"] = "do nothing"

print(args.day.capitalize(), "I am going to", toDo[args.day])
```

```
$ python todo.py today
Today I am going to sleep
```

Running the script

```
$ python todo.py tomorrow
Tomorrow I am going to do nothing

$ python todo.py
usage: todo.py [-h] {today,tomorrow}
todo.py: error: the following arguments are required: day

$ python todo.py -h
usage: todo.py [-h] {today,tomorrow}

positional arguments:
  {today,tomorrow}  Select the day to see tasks

optional arguments:
  -h, --help        show this help message and exit
```