Lab 04: Python on the command-line

CMPT 145

Laboratory 04 Overview

Section 1: Pre-Lab Reading Slide 3

- Python programs as tools not applications.
- The command-line environment for running tools.
- Turning your Python scripts into modules.

Section 2: Laboratory Activities Slide 38

Section 3: What to hand in Slide 48

Section 1

Pre-Lab Reading

PyCharm is not Python

PyCharm is not Python

- PyCharm is an integrated development environment (IDE) which coordinates and manages tasks like editing and running Python scripts all in one application.
- PyCharm does not run your scripts directly. It uses the Python application.
- The Python application has no editor, no graphical user interface.
- The Python application is like a function.
  - Its input is the name of a Python script.
  - Its output is either 0 (success) or 1 (run-time error)

How PyCharm uses Python

- PyCharm does not run your scripts directly. It uses the Python application.
- The Python application is like a function.
- When you hit the Run button in PyCharm, it's something like a function call.
    1. PyCharm starts the Python application, and gives it the name of your script.
    2. Python executes your script.
    3. When you script is finished, the Python interpreter halts.

Python is an interpreted language

- Python is an interpreted language
  - Reads a Python script one line at a time.
  - Performs the actions given on each line, then goes to the next.
  - Includes all the software needed for lists, dictionaries, etc.

Compiled vs Interpreted Languages

- Python is an interpreted language
- C/C++ is a compiled language: an application called a compiler creates a file containing only machine language code, which can be executed directly by the computer.
- Java is a hybrid of compiled/interpreted.
- Compiled languages can result in faster applications.
- Interpreted languages are more portable to other systems (e.g., Windows, Linux, Mac)
- Both are valuable in different ways.

PyCharm's Python Console

- You can run the Python interpreter without any script.
- This causes the Python interpreter to be interactive.
- You can type a line of code, or an expression, and Python will execute it immediately.
- It's a fancy kind of Python-enhanced calculator.
- Useful for experimentation!

Python scripts as tools

- In previous courses, Python programs had to interact with a user to be considered useful.
  - Asking politely for input, repeating on invalid data.
  - Conversational, chatty, output.
- Interactive programs are useful if you need guidance on how to use it.
- Alternatively, Python scripts can also be tools:
  - Get inputs without any politely worded prompt.
  - Produce results without any extra chattiness.
- Tools are useful if you know how to use them, and don't want the extra chattiness.

Freeing the tool from the IDE

- Python scripts are not tied to PyCharm.
- To run a Python script, we can start the Python interpreter ourselves.
- The simplest, and most flexible way is to work on something called a command-line, also called, a Terminal or Command Prompt.
- Fortunately, PyCharm gives us one of those, too!
- In this lab session, we'll use PyCharm's Terminal tab.
- In the future, we'll introduce the UNIX command-line as a tool external to PyCharm.

Command-Line: Background

- Before GUIs, only programmers used computers!
- At that time, programmers did everything using an application called a command-line.
- The command-line is a simple app that repeats the following steps:
  (a) The computer shows that it is ready for a command.
  (b) User types a command then types the RETURN/ENTER key.
  (c) Computer runs or "executes" the command.

Command-Line: Utility

- Not every program must be an interactive application.
- Sometimes a tool should be direct, not chatty.
- The command-line was a perfect environment for this kind of software.
- It's less user-friendly, to be sure, but also very useful.
- The UNIX set of tools have literally hundreds of individual programs written by programmers for programmers.
- We'll learn a little now, and a little more in later labs.
- Their utility is apparent after you know them!

Using PyCharm's Terminal

- At the bottom of the PyCharm window is a button labelled Terminal.
- Clicking on this button starts a command-line from within PyCharm.
- If PyCharm is running on Linux or Mac, the Terminal window is UNIX.
- If PyCharm is running on Windows, the Terminal is (default) Microsoft's Command Prompt.
  - A poor imitation of the UNIX version!

Using UNIX within PyCharm

- For Mac and Linux, PyCharm Terminals are UNIX by default. You don't need any further set-up.
- For Windows, the Command Prompt is a poor imitation of UNIX.
  - To start learning UNIX command-line tools, we will change PyCharm settings.
  - This will work on departmental Windows computers.

UNIX command-line for PyCharm

Windows ONLY

- If you are working on departmental Windows computer:
  1. Using the File menu, open Settings.
  2. Find the Tools heading, and and Terminal sub-heading.
  3. In the Terminal panel (right), find Application Settings
  4. Shell path: `C:\Program Files\git\bin\bash.exe`
  5. Okay!
- See next slide for instructions to check that it worked.

UNIX command-line for PyCharm I

Windows ONLY

- If you are working on departmental Windows computer:
- If you have changed the Settings from the previous slide
- To check that it worked, open the Terminal tab near the bottom of your PyCharm window.
  - You should see some coloured text, and a dollar sign $ prompt.
  - Type python and ENTER/RETURN.
  - You should see something about Python 3.6.5. This is your Python interpreter!
  - Type some Python expressions, for fun!
  - Type `exit()` to leave Python.

Getting a UNIX command-line in PyCharm

Windows ONLY

- If you are on your personal Windows computer system:
  1. Go to the lab to do this.
  2. When you're done, consider installing 'Git for Windows', then trying the above.
  3. There are other ways to add UNIX command-line tools to Windows. Google after your work is done.
- UNIX command-line is really that important.

A simple Python program

```
1  # fact.py
2
3  example = 10
4
5  def factorial(x):
6      """
7      Calculate the product of numbers 1 to x.
8      """
9      total = 1
10     for i in range(1,x+1):
11         total *= i
12     return total
13
14 print(factorial(example))
```

You can find this program in the Laboratory folder.

Running `fact.py` in the Terminal

```
$ python3 fact.py
3628800
```

- The behaviour of `fact.py` is static, because to change its behaviour, we have to use the editor.
- We could use console input to improve it.

Command-line arguments

- The command-line can run Python programs!
- Python's console input and output is directed to the command-line.
- We'll see how to send information to a Python program from the command-line.
- We call this kind of information "command-line arguments"; it's similar to the way we send arguments to a function in Python.

The value of sending information to a program

Consider if we could tell `count.py` to use a different value for the variable `example`. The program would be much more useful.

```
1 $ python3 fact2.py 5
2 120
3 $ python3 fact2.py 10
4 3628800
5 $ python3 fact2.py 15
6 1307674368000
```

Being able to send a program information through the command-line is what we mean by "command-line arguments".

Getting information from the command-line

```
1  # fact.py
2  # version 2
3
4  import sys as sys
5
6  example = int(sys.argv[1])
7
8  def factorial(x):
9      """
10     Calculate the product of numbers 1 to x.
11     """
12     total = 1
13     for i in range(1,x+1):
14         total *= i
15     return total
16
17 print(factorial(example))
```

We use the module `sys`, and a list in that module called `argv`. Nothing else changed.

The list `sys.argv`

- When the command-line runs your Python program, it sends most of the command to the Python interpreter.
- Python initializes the `sys.argv` list and then runs your program.
- Your scripts can look at the `sys.argv` list, or ignore it.
- The first item in the `sys.argv` list (at index 0) is the name of your script. This is a UNIX tradition.
- The data in the `sys.argv` list are strings. You may need to convert the data, as in our example.
- Note: A script that uses command-line arguments should be run from the command-line, not PyCharm.

Command Line Arguments via Terminal

On the command line, arguments are passed to a Python script by listing them after the script filename:

- Arguments are separated by spaces on the command-line.
- To indicate a string argument that contains spaces (like a sentence), use quotation marks (e.g. `'Good job!'` or `"Hello, world"`).

For example:

```
$ python3 scriptname.py arg1 arg2 arg3 ...
```

Summary

- The Python interpreter is independent of any IDE.
- The UNIX command line allows us to emphasize scripts as tools.
- The Python interpreter can be used as a tool on the command line.
- Python scripts can be used as tools on the command line.
- We can send information to a Python script through command line arguments.
- We learned about the command line using PyCharm, but like Python, the command line is independent of any IDE.

Review: Acquiring Arguments within Python

Extract command line arguments using the `sys` module:

- Arguments are stored in `sys.argv` as a list of strings.
- `sys.argv[0]` contains the name of the script.
- Any command line arguments are in the list starting at index 1.
- If no arguments were given, `sys.argv` has length exactly 1.

```
1 import sys
2
3 prog_name = sys.argv[0]    # program name
4 args_list = sys.argv[1:]   # list of arguments
```

Scripts vs. Modules

Scripts (recap)

Definition

A script is just a file containing some Python code.

- It can use functions defined in its own file
- It can import Python modules.
- Running a script (in PyCharm or on the command-line) accomplishes some work we want done.

Global Scope

Definition

The Python global scope is any code in a script outside any function.

- A script must have some code in the global scope.
- If it doesn't, the script does not do anything!

Script example

The following script has a function (lines 3-7), and then some code (lines 9-10) in the global scope.

```
1  # count.py
2
3  def sum_to(x):
4      total = 0
5      for i in range(x+1):
6          total += i
7      return total
8
9  example = 100
10 print("Global code in count.py", sum_to(example))
```

Without lines 9-10, the script only defines a function and would do nothing else.

Example: Importing a script with global code

The following script imports the script `count.py`.

```
1 import count as count
2
3 example = 50
4 print("Global code in count3.py", count.sum_to(example))
```

When this script runs, the global code in `count.py` runs first!

```
1 Global code in count.py 5050
2 Global code in count3.py 1275
```

Modules (recap)

- A module is also a script.
- It defines functions and other Python things.
- It may import other Python modules.
- We import a module to have access to its definitions.

We probably don't want the module to run global code.

Module example

The following module has a function (lines 3-7), but no code that runs in the global scope.

```
1 # count1.py
2
3 def sum_to(x):
4     total = 0
5     for i in range(x+1):
6         total += i
7     return total
8
9 #end of file
```

Preventing global code from executing

The following script has a function (lines 3-7), and then some code (lines 9-11) in an if statement.

```
1  # count2.py
2
3  def sum_to(x):
4      total = 0
5      for i in range(x+1):
6          total += i
7      return total
8
9  if __name__ == '__main__':
10     example = 100
11     print("Global code in count2.py", sum_to(example))
```

Notes on the example

- The variable `__name__`:
  - Created by Python when a script is run.
  - A global variable!
  - Otherwise, it's just a normal Python variable.
- We can check its value, but we better not change it!
- It's value depends on how the script is used:
  - If the file is being run as a script, `__name__` has the value `'__main__'`
  - If the file is being imported as a module, `__name__` refers to the module's name as a string.

Example: Global code is not executed

The following script imports the script `count2.py`.

```
1 import count2 as count
2
3 example = 50
4 print("Global code in count3.py", count.sum_to(example))
```

When this script runs, the global code in `count2.py` does not get executed.

```
1 Global code in count3.py 1275
```

Section 2

Laboratory Activities

Activity 1

- Download the `fact.py` program (Slide 19), and change it so that it behaves as in our example (Slide 23).
- Run the new version of the `fact.py` program in your PyCharm Terminal. At least 3 times with 3 different integers!
- Copy/paste the output of your 3 different examples from the PyCharm Terminal to a file called `lab04-transcript.txt`.

Activity 2

- Run the new version of the `fact.py` program, but without any command-line arguments.
- Observe the error that is reported!
- Add an if-statement to `fact.py` so that it only prints the result if exactly 1 command-line argument is given.
  Hint: Check the length of `sys.argv`!
- If your script detects a missing command-line argument, have it display a helpful message reminding the user to give an integer argument.
- Copy/paste the output of improved version from the PyCharm Terminal to a file called `lab04-transcript.txt`.

Activity 3

- Download the script `self-avoiding-random-walk.py` from the Laboratory.
- Add this script to your Lab04 project.
- Run `self-avoiding-random-walk.py` a few times in the PyCharm Terminal. Note that the output varies a little.
- Modify the script so that it uses command-line arguments to initialize the variables:
  - `n`: grid width and height
  - `trials`: number of times to repeat for an average

Activity 3 continued

- Run the revised version of `self-avoiding-random-walk.py` with different values for `n` and `trials`.
- Use the command-line to explore different values for `n` and `trials`. Find input values that consistently lead to an output of around 40-60 percent dead ends.
- See next slide for hints!
- Copy/paste the output of your exploration of `n` and `trials` from the PyCharm Terminal to your `lab04-transcript.txt` file.

Activity 3 continued

Hint: Keep running the script using different values for `n` first, leaving `trials` small. When you see values close to 50%, increase trials to get a more stable result.
Hint: Precision is not important. Notice how easily you can change the value of a command line argument. Your application is now a tool! Now move on!

Scripts vs. Modules

Modules vs. Scripts

ACTIVITY 4:

1. Download the files: `runcount.py` and `count.py` from Lab04 on Moodle.
2. Make sure `runcount.py` runs!
3. Notice that `count.py` has no code that executes at the global level.

Running scripts

Activity 5:

1. Add one print statement

```
1 print('Global code in count')
```

   to `count.py` after all the operations.
2. Run `count.py` as a script. You should see the print statement's output.
3. Run `runcount.py` as a script. You should see `count.py`'s output.
4. Hand in the console output showing the console output described above.

Modules vs. Scripts

Activity 6:

1. Add the conditional to `count.py` after all the definitions:

```
1 if __name__ == '__main__':
2     print('Global code in count')
```

2. Run `count.py` as a script. You should still see the print statement's output.
3. Run `runcount.py` as a script. You should no longer see `count.py`'s output.
4. Hand in the console output showing the console output described above.

Section 3

Hand In

What To Hand In

Hand in your `lab04-responses.txt` file showing:

- Three different examples from of running your script from Activity 1.
- An example of the output from Activity 2, especially if no command-line arguments are used.
- A copy/paste from the Terminal showing that you used the command-line in your exploration of `n` and `trials` for Activity 3.
- Console output from Activity 5, and Activity 6.
- Don't hand in any code this week!