Lab 03: Python on the command-line

CMPT 145

Laboratory 03 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 42

Section 3: What to hand in ▸ Slide 63

Section 1

Pre-Lab Reading

UNIX Command-Line Introduction

Command-Line: Background

- Modern computer systems use graphical user interfaces (GUIs) with drop-down menus and mouse pointers, or touch-screens.
- Prior to the use of GUIs, users did everything using an application called a command-line.
- The UNIX Command-Line is an interactive application based on console input/output, repeating the following steps:
    1. The computer shows that it is ready for a command.
    2. User types a command, then types the RETURN key.
    3. Computer runs or "executes" the command.

UNIX Command-Line: What Can It Do?

- Everything you are used to doing with a GUI system can be done with a command-line:
- For example, the command-line can be used to:
    - create files and folders
    - copy or move files
    - show file and document contents
    - upload/download files from servers
    - search for information in folders, files, and documents
    - send documents to the printer
    - permanently delete files and documents
- But the UNIX command-line is far more than this!

UNIX Command-Line: What Else Can It Do?

- You can use it to run other console applications.
  - UNIX programmers have created hundreds of such applications for programmers.
  - These are designed to be used together in pipelines.
  - Each command does something simple; each is like a function.
- You can write shell-scripts that can help you do repetitive tasks. E.g.,
  - List all the documents in any subfolder that mention the word "ADT"
  - Global search replace across multiple documents in multiple files
  - Create PDF Lecture slide documents for all chapters

UNIX Command-Line vs UNIX Command

- UNIX Command-line is an interactive console application.
- UNIX commands are typically non-interactive applications.
- There are no menus to list the commands, so the user needs a reference manual to look-up all the commands.
- Common ones are memorized through repeated use.
- You'll learn important ones in CMPT 214.
- In CMPT 145, we need just to realize that Python can be run on the command-line.

UNIX Command-Line: The Context of a Command

- This concept is very important.
- A UNIX command is almost always a single word, or an acronym, related to the purpose of the command.
- The context for a command is the environment in which the command is executed.
- An important aspect of context for a command is the folder (or directory) in which you are working.
    - This is known as the current working directory.
- Note: The words *folder* and *directory* refer to the same thing; *folder* is more modern, but the command-line often uses the older term *directory*.

Basic UNIX command-line commands

- `pwd` (“print working directory.”) displays the folder you are currently working in.
- `ls` lists the contents of a folder with the command.
- `mkdir` creates a new folder in the current working directory.
- `cd` (“change directory”) allows you to change (or “move to”) your working directory.
- `more` will display the contents of a named document to the command-line window.

A few other UNIX command-line commands

- `clear` clears the command-line window.
- `date` displays the date in the command-line window.
- `whoami` displays the user's login name.
- `!!` repeats the previous command, exactly.
- `python3` starts up a Python interactive session in the command window.

PyCharm is not Python

Motivation

- We teach Python in CMPT 140, 141, 145 because it is a useful and friendly language.
- We use PyCharm because it helps students:
  - Edit, run, debug Python scripts in one application.
  - The editor highlights syntax errors.
  - Runtime errors link back to the script.
- Students may believe that:
  - PyCharm is Python
  - Python scripts can only be executed in PyCharm
- If true, Python would be friendly, but not too useful.

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. There is a separate Python application called by PyCharm.
- 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.

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 using them.
- Alternatively, Python scripts can 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 the command-line, Terminal or Command Prompt.
- Fortunately, PyCharm gives us one of those, too!
- In this lab session, we'll use PyCharm's Terminal tab.

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

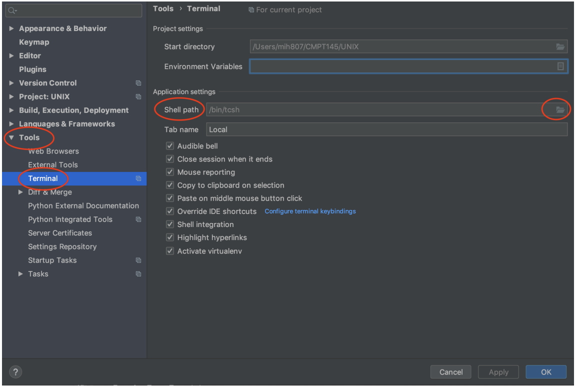
- For Mac and Linux, PyCharm Terminals are UNIX by default. You don't need any further set-up.
- For Windows, the Command Prompt is a poor imitation of UNIX.
  - To start learning UNIX command-line tools, we will change PyCharm settings.
  - This will work on departmental Windows computers.

Getting a UNIX command-line in PyCharm

Windows ONLY

- If you are on your personal Windows computer system:
  1. Install 'Git for Windows'
     - You may have installed it earlier in the semester!
     - Open Preferences : Tools : Terminal
     - Look for Shell Path; use the folder icon to search for `git-bash.exe`.
     - See next slide for visual!
  2. 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.

Setting up UNIX shell for PyCharm on Windows



Tools › Terminal For current project

Project settings

Start directory /Users/mih807/CMPT145/UNIX

Environment Variables

Application settings

Shell path /bin/tcsh

Tab name Local

☑ Audible bell

☑ Close session when it ends

☑ Mouse reporting

☑ Copy to clipboard on selection

☑ Paste on middle mouse button click

☑ Override IDE shortcuts Configure terminal keybindings

☑ Shell integration

☑ Highlight hyperlinks

☑ Activate virtualenv

Click the folder icon (far right) to search for `git-bash.exe`.

A simple Python program

```python
# fact.py

example = 10

def factorial(x):
    """
    Calculate the product of numbers 1 to x.
    """
    total = 1
    for i in range(1,x+1):
        total *= i
    return total

print(factorial(example))
```

You can find this program in the Python Code Examples 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"
    - 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 `fact.py` to use a different value for the variable `example`. The program would be much more useful.

```
$ python3 fact2.py 5
120
$ python3 fact2.py 10
3628800
$ python3 fact2.py 15
1307674368000
```

Being able to send data to program using the command-line is what we mean by “command-line arguments”.

Getting information from the command-line

```python
# fact.py
# version 2

import sys as sys

example = int(sys.argv[1])

def factorial(x):
    """
    Calculate the product of numbers 1 to x.
    """
    total = 1
    for i in range(1,x+1):
        total *= i
    return total

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.

```python
import sys

prog_name = sys.argv[0]    # program name
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`.

```python
import count as count

example = 50
print("Global code in count3.py", count.sum_to(example))
```

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

```
Global code in count.py 5050
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`.

```python
import count2 as count

example = 50
print("Global code in count3.py", count.sum_to(example))
```

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

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

Section 2

Laboratory Activities

UNIX Command-Line Activities

UNIX Command-Line: Getting Started

macOS and Linux

ACTIVITY: Open your command-line!

- For Linux and macOS, find the Terminal tab in PyCharm.
  - This is the same as Linux Konsole, or macOS Terminal.

UNIX Command-Line: Getting Started:

Windows

ACTIVITY: Open your command-line!

- For Windows only:
  - You installed Git for Windows? You configured PyCharm to use git-bash?
  - PyCharm : Settings : Tools : Terminal : Shell path : (browse for git-bash.exe)
  - You only have to do this once.

UNIX Command-Line: Getting Started

REPL.IT

ACTIVITY: Open your command-line!

- For REPL.IT:
    - Open an existing project or crate a new one.
    - Use `F1` or `Control Right Mouse` to open an action menu.
    - Search for “Open Shell”.

Determining the Working Directory with `pwd`

ACTIVITY: To find out the directory (or folder) in which you are currently working, type `pwd` in the command prompt, followed by the RETURN key.

```
% pwd
/student/abc123
%
```

- The command `pwd` abbreviates “print working directory.”
- The command displays the path from the root to your working directory.
- This is the context for the commands you type.

Working Directories

macos, Linux

- On Computer Science Linux computers, you'll see something like:

```
% pwd
/student/abc123
%
```

The abc123 is your NSID.

- On your home macos and Linux computers, you'll see your user account name; whatever you used when you created your account. Something like:

```
% pwd
/Users/username
%
```

Working Directories

Windows

- In the Computer Science Windows computers, you'll see something like:

```
% pwd
\\cshome\\abc123
%
```

The `abc123` is your NSID.

- On Windows at home, the path might look different. Windows uses backslash (\) not slash (/).

```
% pwd
\\Home\\username
%
```

Honestly, Windows is different for the sake of being different. But now they are locked in to it.

Working Directories

REPL.IT

- REPL.IT creates a virtual Linux computer for each project you create. You'll see something like:

```
% pwd
/runner/project
%
```

Your project is the only thing this virtual computer is used for. When you close your browser window, REPL.IT stores the state of your project in the cloud. When you open your project up again, a new virtual computer is started up for you again.

Listing the Contents of a Directory with `ls`

- On the command-line, you can list the contents of a folder with the `ls` command.
- ACTIVITY: Use the `ls` command by typing it into the command-line.
- By default, `ls` lists the contents of the current working directory. Depending on your context, you will see different contents.

Creating Folders with `mkdir`

- The command `mkdir` creates a new folder in the current working directory.
- ACTIVITY: Type `mkdir cmpt145` to create a new folder named “`cmpt145`”.
- ACTIVITY: Use `ls` to check if the folder was created!
- Note: Spaces are meaningful to the command-line. If you type the command `mkdir cmpt 145`, you’ll get two new folders (`"cmpt"` and `"145"`), not one with a space!
- `mkdir` is an example of a command that requires an argument.

Changing Working Directory

- An important aspect of a command's context is the folder in which the command is issued.
- It is possible to "move" to a different folder, and the new folder will be the context of commands that follow.
- On the command-line, this can be done with the command `cd`.
  - The command is an acronym for *change directory*.
- ACTIVITY: In the Terminal, change your working directory to the folder you created earlier, by typing `cd cmpt145`.
  - This is another example of a command that can take an argument.
- Type `pwd` to verify that it changed successfully.

Creating New Files

ACTIVITY:

(a) Open a text editor. Some options:
  - macOS: TextEdit
  - Linux: kate
  - Windows: Notepad
  - REPL.IT: Use the new file icon.

(b) Type some text into the editor window. It doesn't matter what you type here!

(c) Save the text as a file named "`lab03file.txt`" in your "`cmpt145`" folder.

(d) On the command-line, use the command `ls` to verify that it is there.

...and `more`!

We can scroll through a text file with the `more` command.

ACTIVITY:

(a) Typing `more lab03file.txt` into the command-line will display the file you created earlier in the command window.

(b) You should see all the text you typed.

(c) If you typed more than can be seen in a single window, `more` will limit the display to what fits in the window. To see moreof the file, press the SPACE BAR.

About Spaces in Document Names, and Folder Names

- Names of files and folders are allowed to contain spaces.
- On the command-line, spaces are used to separate different parts (“arguments”) of commands.

Note:

Generally, until you are reasonably familiar with the command-line interface, it’s wise to avoid files and folders with spaces in names of folders or documents, including PyCharm projects! Use the underbar character ‘_’ instead!

Summary

We will introduce other commands as we need them.
In this part, we managed files using the command-line:

(a) In the default directory (home directory), we created a folder named "`cmpt145`"

(b) Within folder "`cmpt145`", we created a file (using a text editor) named "`lab1file.txt`"

(c) We created a folder named "`lab1`" under "`cmpt145`"

Commands used:

- `pwd`
- `mkdir`
- `cd`
- `ls`
- `more`

Activity 1

- Download the `fact.py` program (Slide 23), and change it so that it behaves as in our example (Slide 27).
- 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 `a9q2-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 `a9q2-transcript.txt`.

Activity 3

- Download the script `self-avoiding-random-walk.py` from the Laboratory.
- Add this script to your Lab03 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 a file called `a9q3-transcript.txt`.

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!

Section 3

Hand In

What To Hand In

You'll be handing in some parts of your work for Assignment 9! Read that document for "What to hand in."