Code re-use

Eric Franzosa (franzosa@hsph.harvard.edu)

http://franzosa.net/bst273

Overview

- Announcements
- Original plan: parallel computing and workflows in `doit`
- New plan: “code re-use”
  - Making your scripts executable
  - Making your modules findable
  - Installing new packages from the web
- *A bunch of things I wish I’d learned much earlier*
- I’ll introduce `doit` at the end, and we may come back to it on Thursday

Types of code re-use

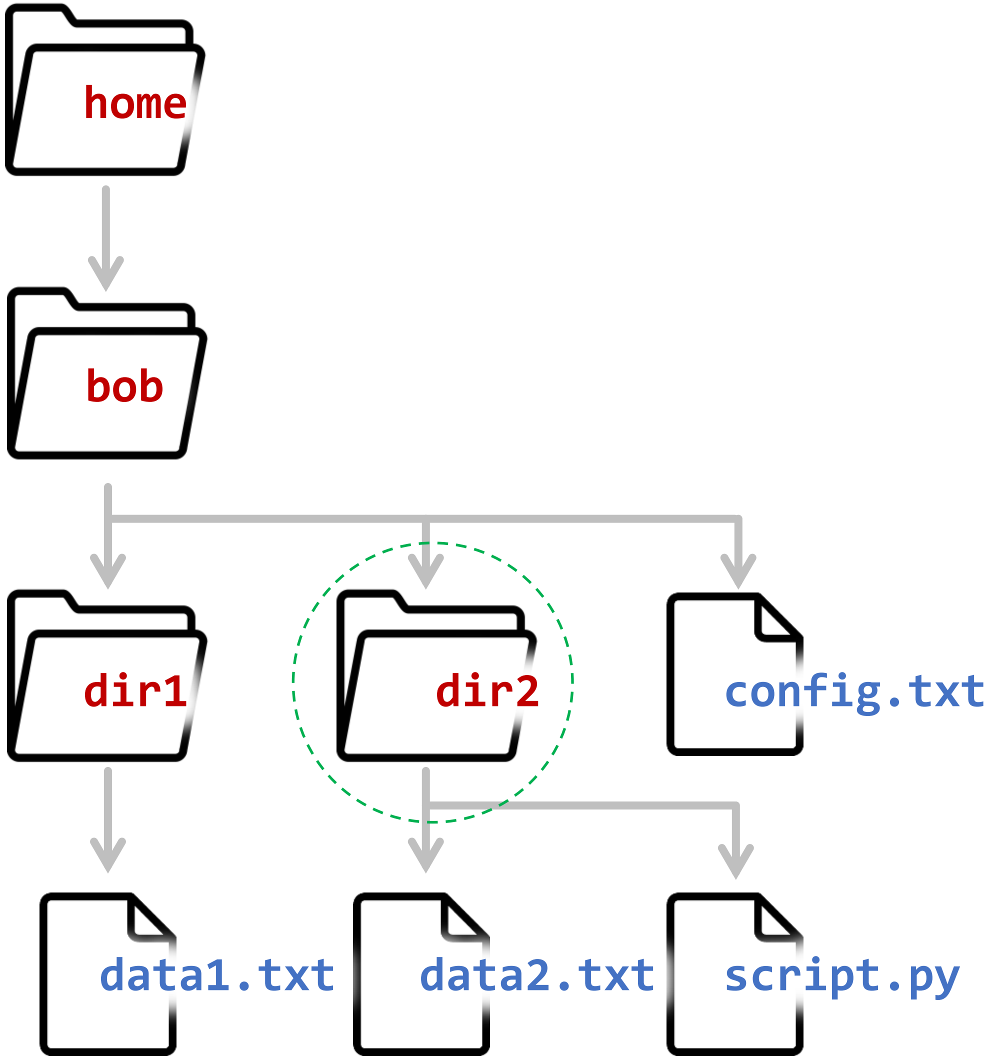
- Turning a one-time script into a reusable program
  - The same way we use **`grep`**
- Importing an existing element of a script into another script
  - The Python module approach

Script re-use

Elements of script re-use

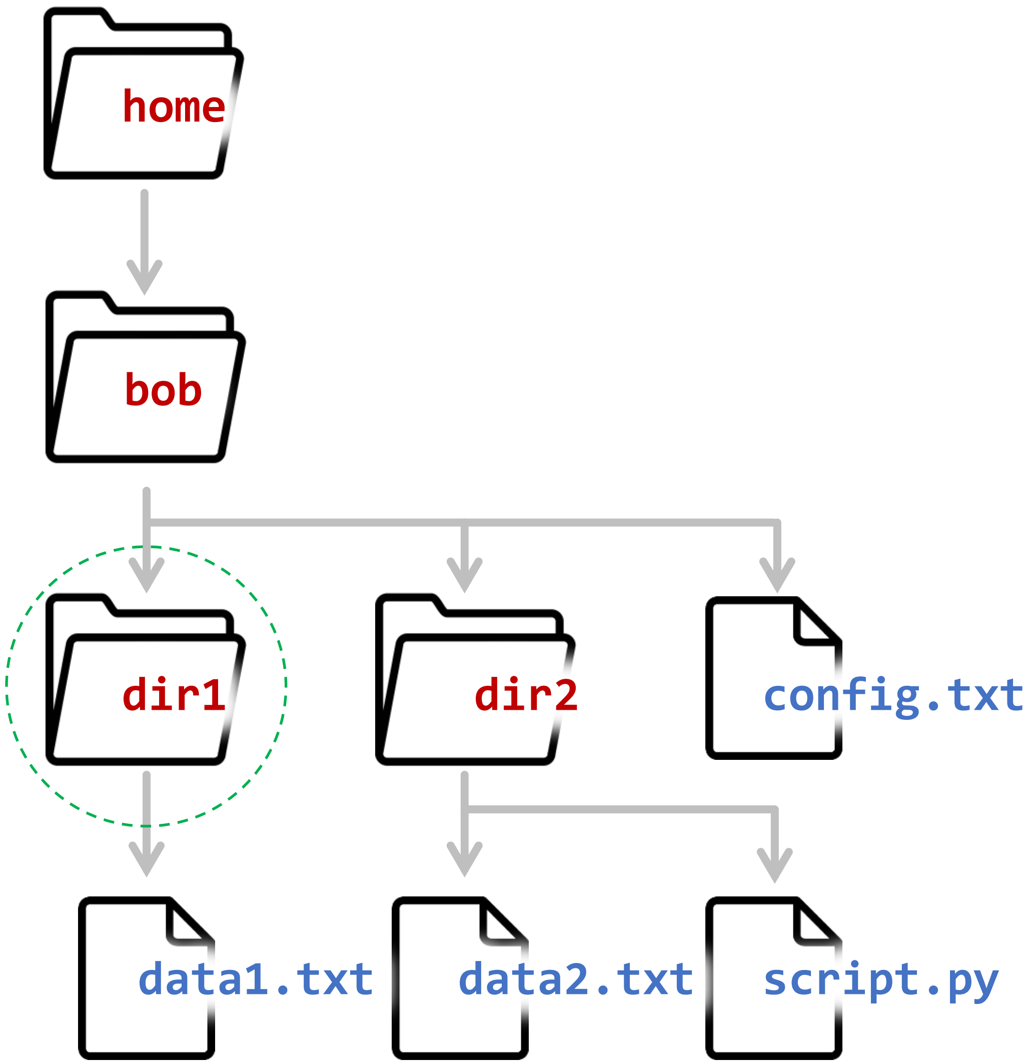
- By future you
    - Making the script as generic as possible
    - Implementing a helpful command-line interface
    - ***Being able to run the script from anywhere***
- For others
    - Publishing the script online (as a public Github repository or Python package)
    - Producing a “manual” to document the script (e.g. a README.txt file)
    - Advertising your script

Finding a script



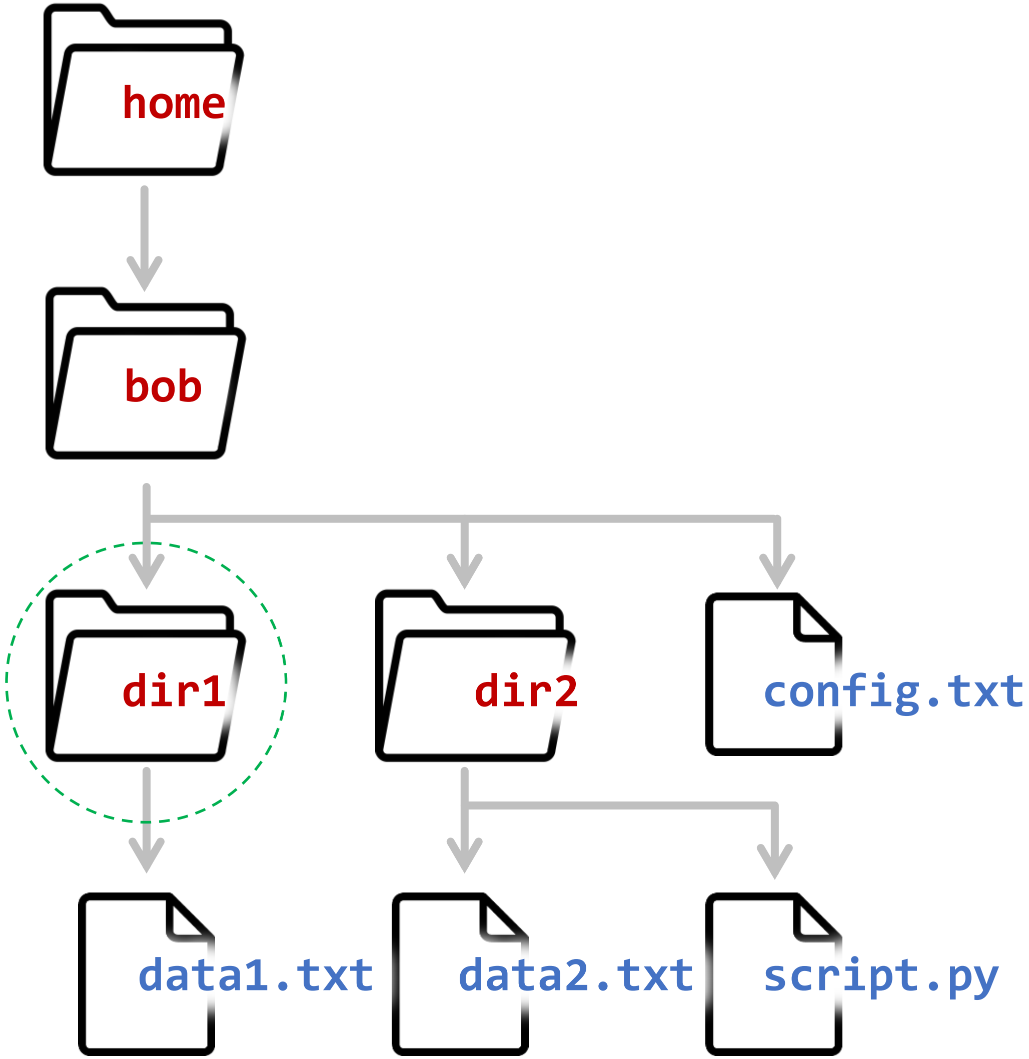
- So far, we've focused on running scripts that are present in our current working directory (`dir2` currently)
- We can directly run our script on `data2.txt` from the working directory
  - `python script.py data2.txt`
- We can run the script on data1.txt using parent directory syntax:
  - `python script.py ../dir1/data1.txt` (*Mac/Linux style*)
  - `python script.py ..\dir1\data1.txt` (*Windows style*)

Finding a script



- We could also move to dir1 and call our script from there with the same style of parent directory syntax:
  - `python ../dir2/script.py data1.txt` (*Mac/Linux style*)
  - `python ..\dir2\script.py data1.txt` (*Windows style*)

Finding a script



- If we're working in dir1 we could also call the script by its *absolute path*
  - *This doesn't require knowing where the script is relative to us*
- On Mac:
  - `python /home/bob/dir2/script.py data1.txt`
- On Windows:
  - `python C:\home\bob\dir2\script.py data1.txt`

Finding a script

- This gets really old, really fast
- You will be tempted to simply copy scripts to the current working directory
  - e.g. When starting a new project
- There are a number of problems with this approach
  - You wind up with many script copies floating around
  - New features aren’t back-propagated to old versions
  - Still not helpful for executing the script in child folders of the project

A better way: The system PATH

- **PATH** is an environment variable
  - Just like a Python variable, it’s a programming structure for storing data
  - Environment variables “belong” to your Operating System, not any single script
  - We saw an example way back in Lecture 2 with **$HOME**
- **PATH** is a list of locations that your operating system searches through to find a program requested from the command line
  - ***$ program***
  - Search through **PATH** and execute the FIRST matching option you find
  - Not required to run ***program*** from the current directory, even if present
    - Use ***$ ./program*** for that

A better way: The system PATH

- Keep your scripts in one (or a few) centralized locations
- Add those locations to the **PATH**
- Works for repositories as well
- Mechanics are slightly different on Mac (and Linux) vs. Windows

PATH on Mac/Linux

PATH on Mac/Linux

- Execute: `echo $PATH` to see your current settings

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/home/efranzosa/hg/hclust2/:/home/efranzosa/hg/zopy/scripts/:/home/efranzosa/hg/breadcrumbs/breadcrumbs/scripts:/home/efranzosa/.local/bin:/home/efranzosa/.linuxbrew/bin:/home/efranzosa/hg/metaphlan2
```

- What you end up with is a *colon*-delimited list of *absolute* file paths
- We can clean it up with a command-line chain…
- Execute: `echo $PATH | sed "s/:/\n/g"`

```
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
…
```

PATH on Mac/Linux

- We can modify the path with the `export` command

  -
    ```
    $ export PATH="$PATH:/some/other/location"
    ```
- In Mac/Linux (bash) command-line syntax, this says, “set **`PATH`** equal to everything in **`PATH`** plus **`/some/other/location`**”
- Note, the above syntax means that your new location will be searched *last*
  - An existing program in PATH with the same name will be used first
  - Hence, this syntax is used more often (despite looks less intuitive):

    -
      ```
      $ export PATH="/some/other/location:$PATH"
      ```

PATH on Mac/Linux

- These changes will be lost when you start a new Terminal
- To make the changes permanent, put them in your `~/.bashrc` file
  - `.bashrc` stands for "bash read config"
  - Lives in your home folder (`~`); stores settings for command-line work
  - Because this file begins with `.`, it is hidden by default
- Add the "`export PATH`" command from the previous slide to the end of your `.bashrc` file to make this addition permanent
  - Then restart the Terminal or execute: `$ source ~/.bashrc` to update your settings
  - You can now execute scripts in `/some/other/location` from anywhere
  - e.g. `$ script.py`

PATH on Mac/Linux

- Note, when we run Python scripts from the Terminal like this
  - `$ python script.py`
- We are actually calling the `python` program, which is located in the PATH, with the name of the script as an argument
- To directly execute a script:
  - `$ script.py`
- It must begin with a special line of text called a "shebang":
  - `#!/usr/bin/python` (OR) `#!/usr/bin/env python`
  - *You may have noticed this in the homework scripts*

File permissions on Mac/Linux

- If you get “Permission denied,” tell the system it’s OK to execute this file:
  - `$ chmod u+x /some/other/location/script.py`
- Files on Mac and Linux computers have a special set of permissions
  - (**r**)eadable – can look at the file or folder
  - (**w**)ritable – can modify/delete the file or folder
  - e(**x**)ecutable – can execute the file as a program
- These permissions are stratified over three types of people
  - (**u**)ser – you
  - (**g**)roup – people in your working group (other than you)
  - (**o**)thers – everyone else in the universe
- Execute: `$ man chmod` to learn more about these options

File permissions on Mac/Linux

- When you execute `ls -l`, files are listed along with their permissions

```
total 381K
drwxrwxr-x 4 efranzosa huttenhower_lab   57 May 31  2016 build
drwxrwxr-x 2 efranzosa huttenhower_lab  123 Aug  9  2016 dist
drwxrwxr-x 2 efranzosa huttenhower_lab  107 Jun 21  2016 examples
drwxrwxr-x 7 efranzosa huttenhower_lab  343 Sep 14  2017 humann2
-rw-rw-r-- 1 efranzosa huttenhower_lab 1.2K May 28  2016 LICENSE
drwxrwxr-x 2 efranzosa huttenhower_lab  188 May 31  2016 humann2.egg-info
-rw-rw-r-- 1 efranzosa huttenhower_lab 1.2K Aug 26  2016 MANIFEST.in
-rw-rw-r-- 1 efranzosa huttenhower_lab  16K May  3  2017 history.md
-rwxrwxr-x 1 efranzosa huttenhower_lab 1017 Jul  6  2016 readme.md
-rw-rw-r-- 1 efranzosa huttenhower_lab  27K Oct 26  2017 setup.py
-rw-rw-r-- 1 efranzosa huttenhower_lab  201 Aug  9  2016 counter.txt
-rw-rw-r-- 1 efranzosa huttenhower_lab 2.2K Sep  7  2017 bitbucket-pipelines.yml
```

- The initial string of chars indicates if the file is a directory (`d`) or not (`-`) followed by the `rwx` permissions for you, group, and others

PATH on Mac/Linux

- If you ever have any doubt about which script you’re executing, or where it lives, you can run:

  -
    ```
    $ which script.py
    ```
- This will return the first match to script.py in your PATH (i.e. the one that would be executed if you just ran `$ script.py`)

  -
    ```
    /some/other/location/script.py
    ```

PATH on Windows

PATH on Windows

- Execute: `echo %PATH%` to see your current settings

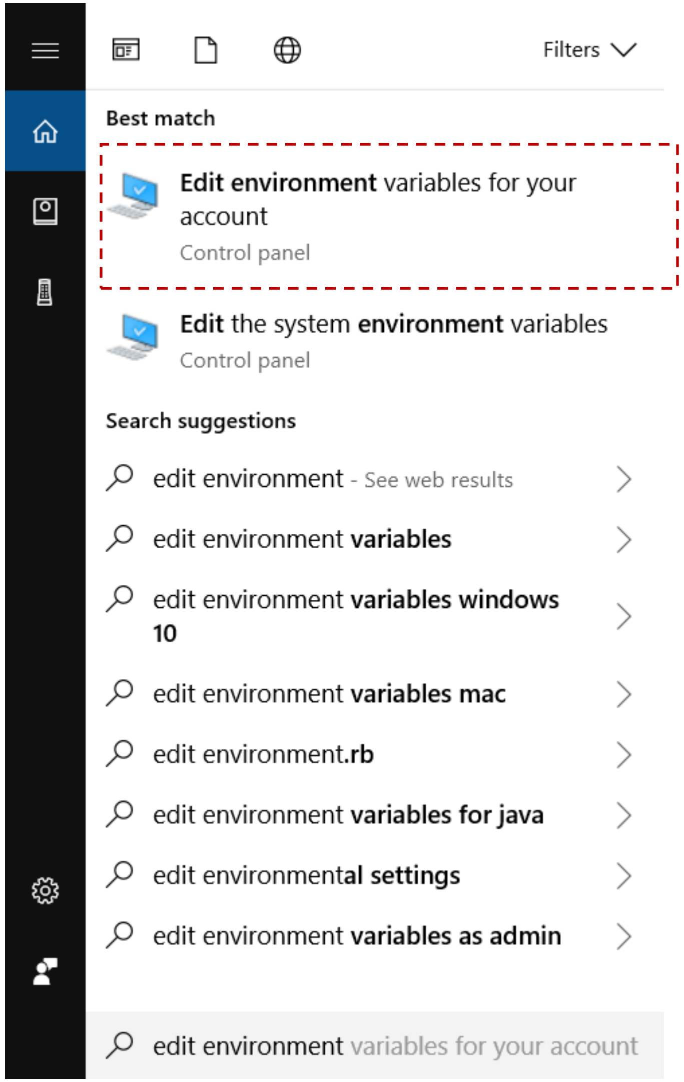
  ```
  C:\Program Files\PuTTY\;C:\Program Files (x86)\Gow\bin;C:\Program
  Files\Git\cmd;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\Syste
  m32\WindowsPowerShell\v1.0\;C:\WINDOWS\System32\OpenSSH\;C:\Test;C:\Users\Eric
  Franzosa\AppData\Local\atom\bin;C:\ProgramData\Anaconda2;
  ```

- What you end up with is a *semicolon*-delimited list of *absolute* file paths
- We can clean it up with a command-line chain… (if you have **gow** installed)
- Execute: `echo $PATH | sed "s/;/\n/g"`

  ```
  C:\Program Files\PuTTY\
  C:\Program Files (x86)\Gow\bin <- note the presence of Gow here!
  C:\Program Files\Git\cmd
  C:\WINDOWS\system32
  C:\WINDOWS
  C:\WINDOWS\System32\Wbem
  …
  ```

PATH on Windows

- Editing the PATH is actually somewhat easier on Windows vs. Mac/Linux
- On Windows 10, search for “edit environment variables” and click the first hit
- If you need to find this location manually (or on other versions of Windows) it’s usually under…
    - Control Panel > System > Advanced System Settings > Edit Environment Variables (or something similar)



Best match

Edit environment variables for your account
Control panel

Edit the system environment variables
Control panel

Search suggestions

edit environment - See web results

edit environment variables

edit environment variables windows 10

edit environment variables mac

edit environment.rb

edit environment variables for java

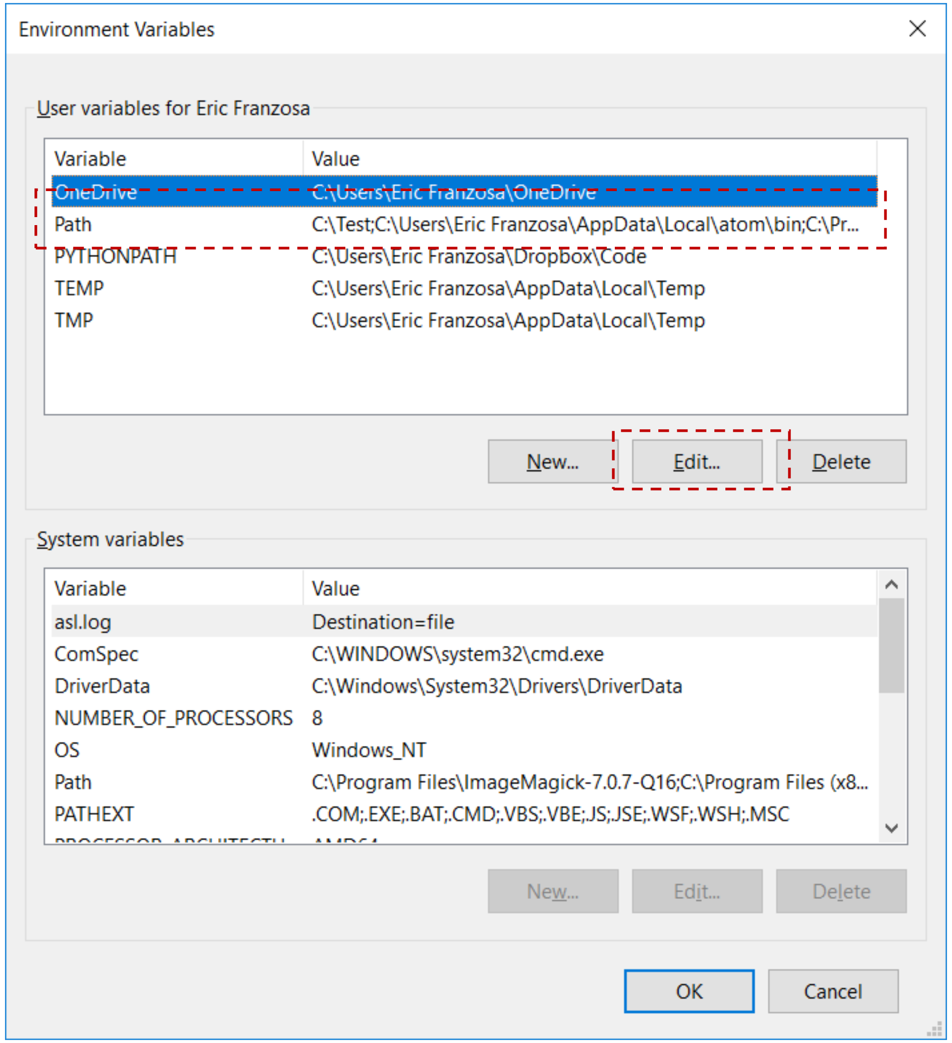
edit environmental settings

edit environment variables as admin

edit environment variables for your account

PATH on Windows

- You’ll see a Window like this listing all environment variables on your computer
- The ones in the top panel belong to you
- The ones in the bottom panel belong to the system (OR) all users
    - Relevant in “*install for all users*” dialogs
- Select your “Path” and click “Edit”



Environment Variables

User variables for Eric Franzosa

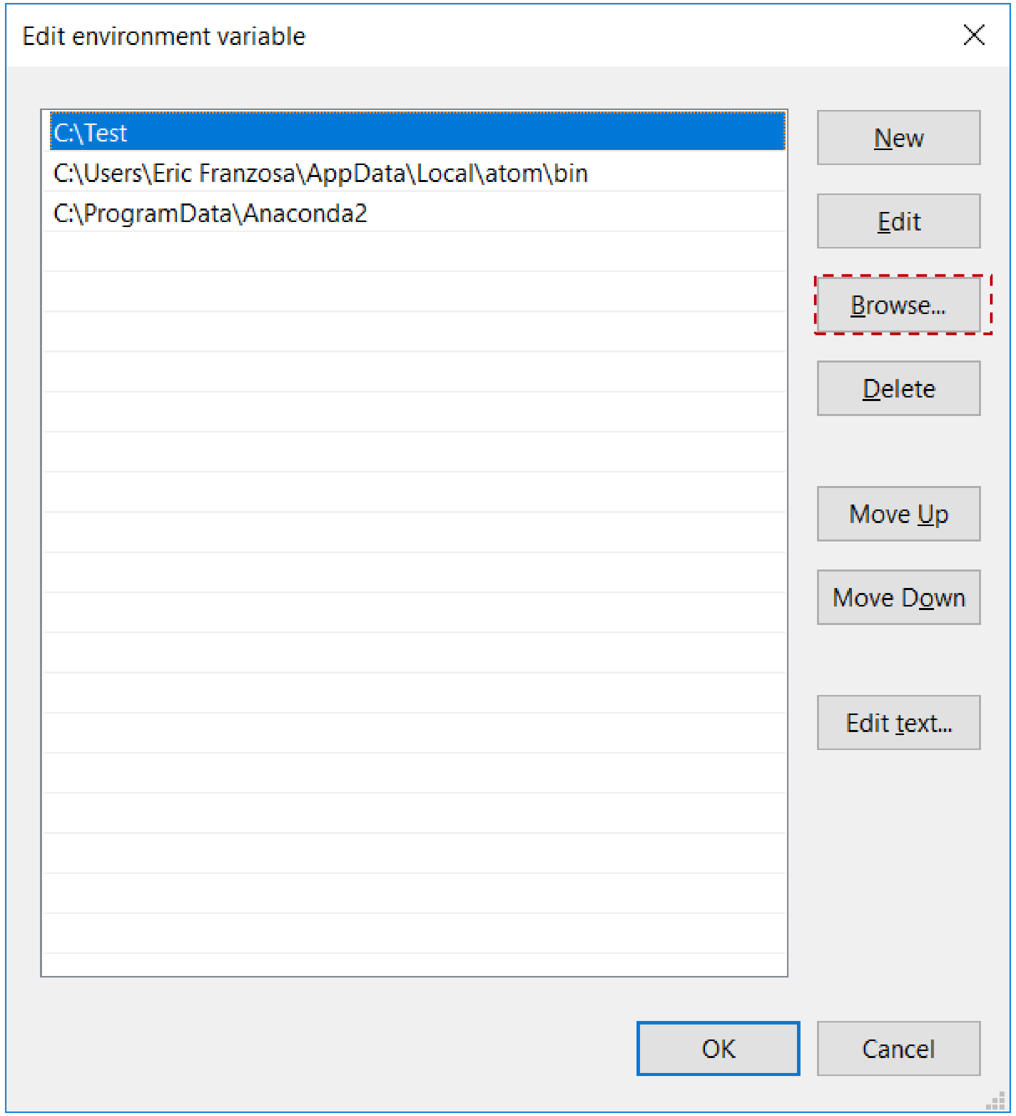
Variable	Value
OneDrive	C:\Users\Eric Franzosa\OneDrive
Path	C:\Test;C:\Users\Eric Franzosa\AppData\Local\atom\bin;C:\Pr...
PYTHONPATH	C:\Users\Eric Franzosa\Dropbox\Code
TEMP	C:\Users\Eric Franzosa\AppData\Local\Temp
TMP	C:\Users\Eric Franzosa\AppData\Local\Temp

System variables

Variable	Value
asl.log	Destination=file
ComSpec	C:\WINDOWS\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	8
OS	Windows_NT
Path	C:\Program Files\ImageMagick-7.0.7-Q16;C:\Program Files (x8...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTU	AMD64

PATH on Windows

- You can now simply browse for the location(s) that you want to add
- Then click OK to save and leave this window and OK again to leave the previous window



Edit environment variable

C:\Test
C:\Users\Eric Franzosa\AppData\Local\atom\bin
C:\ProgramData\Anaconda2

PATH on Windows

- While it’s still good practice to include shebangs (`#!`) in Python code you write on Windows, Windows doesn’t understand these by default
- Instead, if you execute a Python script on Windows, it will open the script in your editor of choice (e.g. Atom)
- To avoid this, you need to use the Windows “open with” menu and set `.py` files to always open with `python.exe`
    - Located in your Anaconda3 folder
- ***This is not a perfect solution; I need to investigate it further…***

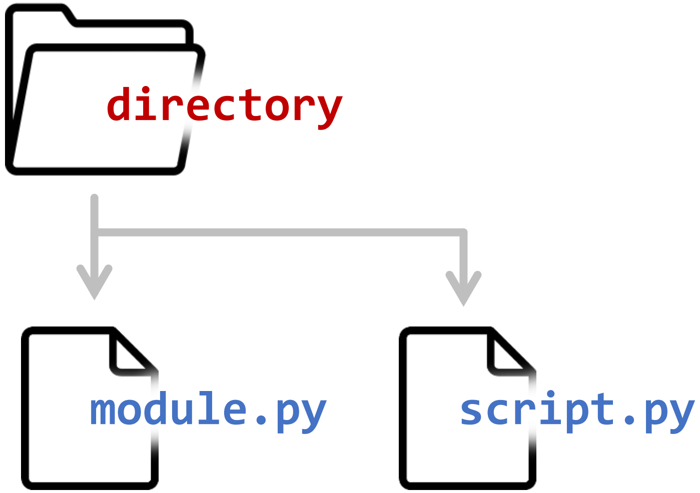
Module re-use

Module re-use

- What if I don’t want to re-run a whole script, but rather want to use some part of it (e.g. a function) in another script?
- This is where modules come in

Module concepts

- The following example assumes I have two Python files in the same folder
  - `script.py` is a new script I am working on
  - `module.py` is some existing code that I want to re-use



Module concepts

module.py (*open in Atom*)

```
# an approximation of pi
pi = 3.14

# area of a circle
def area( r ):
    return pi * r ** 2

# first few primes
primes = [2, 3, 5, 7, 11]
```

script.py (*open in Atom*)

```
import module

print( module.pi )
```

"module" is a **namespace:** a collection of previously defined objects (variables, functions, etc.)

We request individual objects using "." syntax

(*a terminal*)

```
$ python script.py

  3.14
```

Module concepts

module.py (*open in Atom*)

```
# an approximation of pi
pi = 3.14

# area of a circle
def area( r ):
    return pi * r ** 2

# first few primes
primes = [2, 3, 5, 7, 11]
```

script.py (*open in Atom*)

```
from module import area, pi

print( pi )
print( area( 2 ) )
```

We can also import *specific* variables/functions from a module into the main namespace as a comma-separated list.

(*a terminal*)

```
$ python script.py

  3.14
  12.56
```

Module concepts

`module.py` *(open in Atom)*

```
# an approximation of pi
pi = 3.14

# area of a circle
def area( r ):
    return pi * r ** 2

# first few primes
primes = [2, 3, 5, 7, 11]

# say hello
print( “Hello, World!” )
```

`script.py` *(open in Atom)*

```
import module

print( module.pi )
```

Module code is executed when it’s imported. This will cause “Hello, World!” to print before pi.

(a terminal)

```
$ python script.py

  “Hello, World!”
  3.14
```

Module concepts

module.py (*open in Atom*)

```
# an approximation of pi
pi = 3.14

# area of a circle
def area( r ):
    return pi * r ** 2

# first few primes
primes = [2, 3, 5, 7, 11]

# say hello in script mode
if __name__ == “__main__”:
    print( “Hello, World!” )
```

script.py (*open in Atom*)

```
import module

print( module.pi )
```

We can use a special conditional to indicate that some code should only be run when the module is run as a script!

(*We’ll come back to this next week*)

(*a terminal*)

```
$ python script.py

  3.14
```

Module concepts

module.py (*open in Atom*)

```
# an approximation of pi
pi = 3.14

# area of a circle
def area( r ):
    return pi * r ** 2

# first few primes
primes = [2, 3, 5, 7, 11]

# say hello in script mode
if __name__ == “__main__”:
    print( “Hello, World!” )
```

This module is just a Python script and can also be executed.

(*a terminal*)

```
$ python module.py

  “Hello, World!”
```

Finding modules

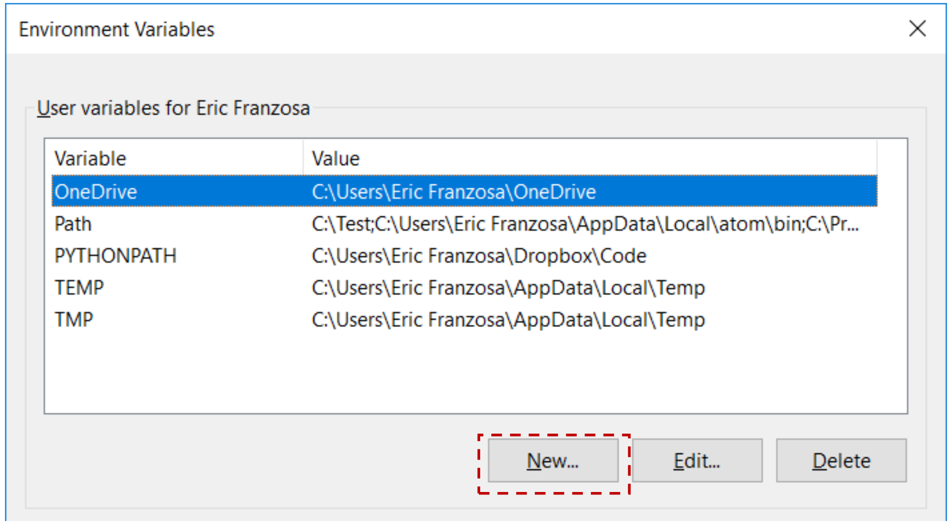
- When you include a line like “`import module`” in a Python script, Python first looks for a file called `module.py` in the current working directory
  - Note that this is different from how the system searches for programs
- Failing that, it then looks to a system variable called the `PYTHONPATH`
  - Very similar to the system PATH: a list of locations to search to find Python modules
- Finally, it searches through a number of other locations specified by your particular Python installation
  - You can see the full list with:
    - `import sys`
    - `print( sys.path )`

PYTHONPATH on Mac/Linux

- You can manipulate **PYTHONPATH** exactly as we manipulated **PATH**

PYTHONPATH on Windows

- You can manipulate **PYTHONPATH** exactly as we manipulated **PATH**
- However, **PYTHONPATH** may not be an existing environment variable on your system (Anaconda does not define one by default)
- You can use the "New…" option to create **PYTHONPATH**, then populate it using the methods we used for **PATH**



Environment Variables

User variables for Eric Franzosa

Variable	Value
OneDrive	C:\Users\Eric Franzosa\OneDrive
Path	C:\Test;C:\Users\Eric Franzosa\AppData\Local\atom\bin;C:\Pr...
PYTHONPATH	C:\Users\Eric Franzosa\Dropbox\Code
TEMP	C:\Users\Eric Franzosa\AppData\Local\Temp
TMP	C:\Users\Eric Franzosa\AppData\Local\Temp

Importing with `.` syntax

- Let’s say you’ve created a bunch of useful functions in a bunch of useful scripts that you want to organize (and maybe share) as one module
  - Saved in a folder called `python_stuff`
- Saving `python_stuff` as a repository is a good start
- Add an empty file to `python_stuff` called `__init__.py`
- This will allow you to do things like
  - `import python_stuff.stats_stuff`
  - `from python_stuff.stats_stuff import my_t_test`
- Helps to avoid collisions with existing Python packages

Getting new scripts and modules

Method 1: *manually*

- Clone a Python repository from Github
- Add the newly created folder to your **PATH** and **PYTHONPATH**
- Many repositories will contain subfolders for scripts and module code
  - The script folder goes in **PATH**
  - The module folder, which may be called `src/` or have the same name as the repository itself, goes in **PYTHONPATH**

Method 2: `setup.py`

- Clone a Python repository from Github
- Execute the included **`setup.py`** file
  - `python setup.py install`
  - `python setup.py install --user` (*if you don't have admin rights*)
- A special Python "installer" that will, among other things, add scripts to the **`PATH`** and make module code **`import`**-able
- May also compile non-Python code components

Method 3: `pip`

- Download and install with one command
  - `pip install` *`package`*
- Makes an effort to satisfy Python dependencies
  - For example, if *package* itself imports *package2*
- Packages come from `pypi.org`, the Python Package Index
- 100Ks of packages available

Method 4: conda

- Download and install with one command
    - `conda install` *`package`*
- Makes an effort to satisfy Python and **non-Python** dependencies
    - For example, if *package* itself imports *package2* and calls other programs
- Rapidly becoming the preferred way to install Python software
- Graphical interface to conda is bundled with Anaconda
    - Anaconda Navigator

doit

The `doit` workflow manager

- Lots of great documentation online: **`http://pydoit.org/`**
- Allows you to define workflows in Python
    - Workflow = sequence of tasks where output of one task becomes input to the next
        - Map 10 samples’ worth of RNA-seq reads to a reference genome
        - Quantify transcript abundance
        - Run differential expression (DE) statistics
        - Make a plot of DE genes
    - A larger-scale version of a command-line chain
- Will only (re)run a task if 1) it’s never been run before or 2) one of the “dependencies” (a program or an input file) has changed
- Download and install with one command: `$ conda install doit`