basic python Documentation

Release 1.0.1

Matt Hanson

Aug 21, 2018

Must Have Python Basics

1 What is Python 3
- 1.1 A brief history of python . . . 5
- 1.2 Python versions . . . 5
- 1.3 Interactive python, running scripts and IDEs . . . 5
- 1.4 Philosophy of coding, The Zen of Python, and pep8 . . . 6

2 Building Python on Your computer 9
- 2.1 Installing python and packages . . . 10
- 2.2 Using python environments . . . 10
- 2.3 Our recommended python installation for this course . . . 11
- 2.4 Our recommended python installation for further work . . . 11
- 2.5 Build your own python environment file for environmental scientists . . . 11
- 2.6 Setting up and managing virtual environments with conda . . . 13

3 Basic Python Objects, Variables, and Operators 17
- 3.1 Variables . . . 17
- 3.2 Numbers: Integers and Floats . . . 18
- 3.3 Boolean . . . 20
- 3.4 Strings . . . 21
- 3.5 The print function . . . 22
- 3.6 The Python None . . . 22

4 The List 23
- 4.1 What’s a list . . . 23
- 4.2 length, indexing, and slicing . . . 23
- 4.3 list specific operations and functions . . . 25
- 4.4 The tuple . . . 26

5 Dictionaries 27
- 5.1 What’s in a dictionary . . . 27
- 5.2 Important dictionary functions . . . 28

6 Strings, the good, the bad, and the ugly 31
- 6.1 String indexing . . . 31
- 6.2 String operators & functions . . . 31
- 6.3 Formatted output . . . 33

7 Conditional Statements **35**

8 Loops in Python **37**
8.1 For loops in python 37
8.2 Useful builtins for for loops 38
8.3 While loops in Python 39

9 Functions in Python **41**
9.1 What is a function? 41
9.2 Understanding function documentation 42
9.3 Creating your own function in python 43
9.4 using *args and **kwargs 44

10 Using Packages in python **45**
10.1 External packages 45
10.2 Importing from your own python scripts 46
10.3 Adding a folder to the PYTHONPATH 47

11 Practice Exercises **49**

12 Handy Builtin modules **51**

This course is is an open source training course to give environmental scientists the foundation they need to begin learning and using the scientific tools that have been developed for python. The course focuses on simplifying python language into two buckets: **the must haves** and **the nice to haves**. The course is far from comprehensive, but we believe that it's better for people to get cracking, and fill in the gaps as they develop their skills.

> **Expected course duration: 1-2 days**

CHAPTER 1

What is Python

The official description of python is:

> Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

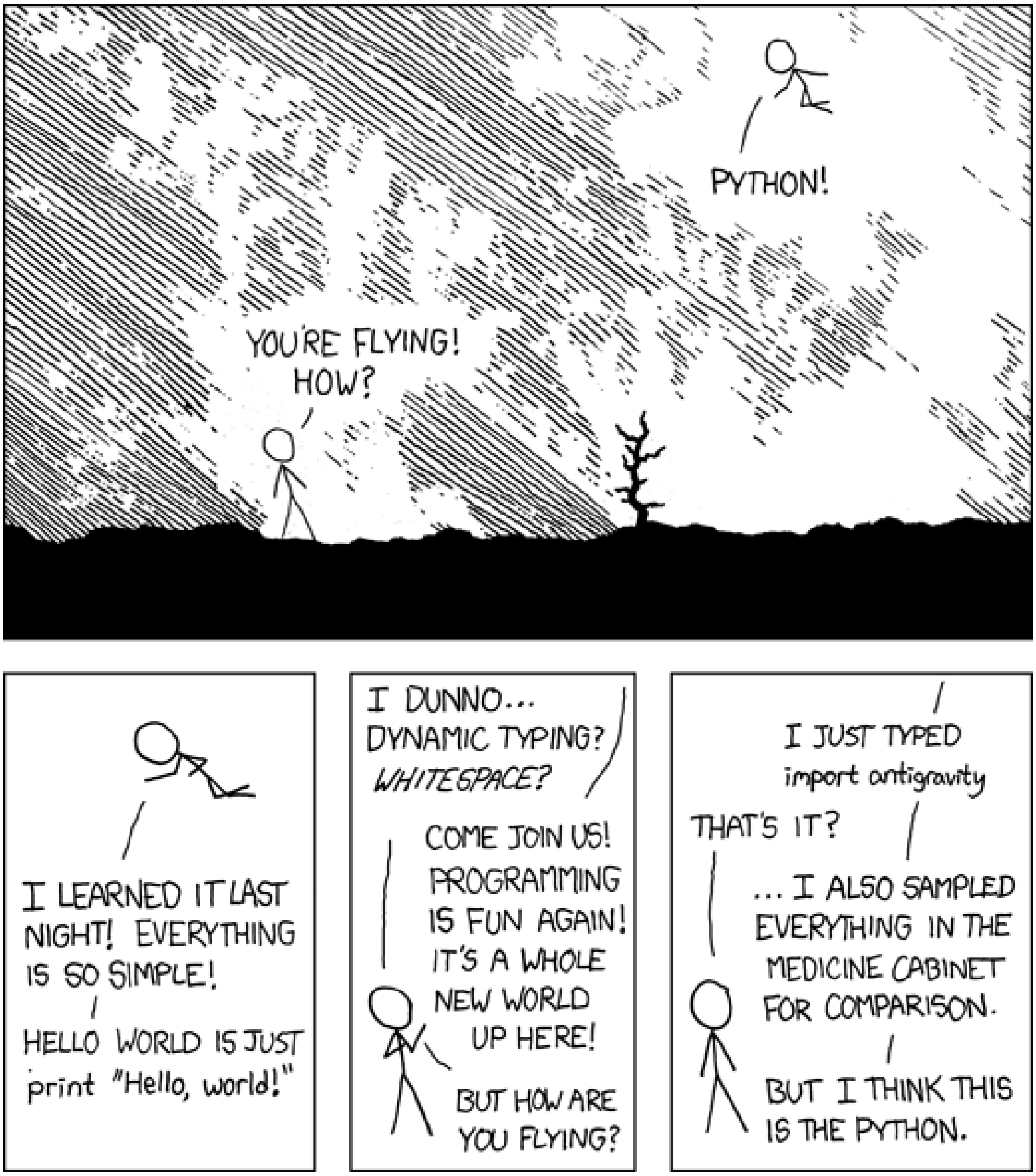
All that to say that Python is an open-source programming language that is easy to read, easy to write and fast to develop.

Perhaps python best described in XKCD by Randell Munroe:

p.s. in a console type *import antigravity* and see what happens.

Using python for environmental science has several advantages over traditional means:

1. It's free and doesn't need a licence. The skills you learn in python can be used everywhere.
2. Repeatability. Anyone on earth can reproduce your analysis, which is the fundamental goal of science.
3. Speed and efficiency. Do the work once and reuse the code.
4. Automation. Take the soul sucking tasks and give them to a non-sentient being (and no, we're not talking about a grad)
5. Big data. Only with a programming language can you manage the big data that's rapidly becoming the norm.
6. Interactive systems. Python allows interactive plotting, reporting, and systems that simply are not possible in traditional systems.
7. Community. There is an enormous online community of scientific python users as demonstrated by Programming: Pick up Python


YOU'RE FLYING! HOW?
PYTHON!
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

1.1 A brief history of python

Guido Van Rossum published the first version of Python code (version 0.9.0) at alt.sources in February 1991. The creation of python was heavily influenced by the development of the programming language ABC. Since then it as seen explosive growth to become one of the most common programming languages in the world.

What about the name python? Guido van Rossum, the creator of Python wrote "Over six years ago, in December 1989, I was looking for a 'hobby' programming project that would keep me occupied during the week around Christmas. My office … would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus)."

1.2 Python versions

There are currently two main versions of python - Python 2 and Python 3. Python 3 is the branch that is being actively developed, while Python 2 only has legacy fixes. Python 2 only exists because Python 3 was not backwards compatible with Python 2. All new programmers should go straight to Python 3 unless there's a really good reason (e.g. legacy packages in 2.7). The current stable version is Python 3.6, which is the version we will use in this course.

1.3 Interactive python, running scripts and IDEs

1.3.1 Interactive python

Python has the option to run in an interactive mode. This way every command that you enter is executed and you immediately see the output. This is a great way to play around with commands in python and build some simple knowledge, but it is quite limiting when it comes to doing complicated analysis.

1.3.2 Running scripts

You can also package up your python commands into a script (basically a text file ending with .py). To run the script on windows you can execute it with the command prompt: path_to_pythonpython.exe path_to_script.py. This will then run through all of the commands in the python script. The advantages to running a script rather than doing work through interactive python is that you keep all of your work. Your project effectivly documents itself. This is really the best practice for good repeatable science.

1.3.3 IDEs

People often find that writing python scripts in notepad or similar can be a bit tedious. Instead environmental scientists often turn to integrated development editors or IDEs. An IDE normally consists of a source code editor, build automation tools, and a debugger. Most modern IDEs also have intelligent code completion, which makes scripting much quicker.

IDEs are essentially the programmers equivilant of a Harry Potter wand. Google IDEs and there are certainly lots of strong opinions. Regardless of which IDE you end up with (and you will certainly end up with one of them), there are some important considerations. IDE's make it easy to pass code from a script editor into a console, which is really useful for debugging and testing things as you develop them. Despite this you must never:

1. create a finished script that runs different things based on the lines that you comment out.

2. create a finished script that relies on some other command being run in an interactive python console.
3. create a finished script that relies on commands being run in some order other than the order that they are written in.
4. create a finished script that relies on some lines of code not being run.

The reason for these four rules is code repeatability. One of the biggest advantages to science in any programming language is that given the inputs anyone (including yourself) can replicate the work that you've completed. Break any of the aforementioned rules and it is usually quicker to redo the work from scratch than use the code you've produced.

As far as specific IDEs we would recommend that people start with **Spyder** because it is a relatively lightweight IDE that is open source and comes with the anaconda build that you have already installed. As time goes on, particularly once you start developing tools then keep your eye out for other IDEs just in case they suit you better. We won't give suggestions, it's better to simply ask around - someone will have an opinion.

1.4 Philosophy of coding, The Zen of Python, and pep8

1.4.1 Philosophy of coding

For new comers to python (or any programming language) some useful tips:

1. Make lots and lots of mistakes. Red on the screen isn't a failure, it's a natural part of the process.
2. You are probably not making more mistakes, you're just catching more of them.
3. Document everything. Comments (started with the #) are your friend.
4. Accept that things will take longer in the short term, but shorter in the long term.

1.4.2 The Zen of Python

Long time Pythoneer Tim Peters succinctly channels the Benevolent Dictator for Life's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one –obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.

- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

1.4.3 Pep8

There is a documented style guide for python; it's called PEP8. While following PEP8 isn't necessary, it does make for much more readable (and thus less error prone) code. Through this course we will try our best to follow pep8 and as you start to write scripts we'd encourage you to start taking it on board.

CHAPTER 2

Building Python on Your computer

- *Installing python and packages*
- *Using python environments*
- *Our recommended python installation for this course*
- *Our recommended python installation for further work*
- *Build your own python environment file for environmental scientists*
  - *Advanced visualisation packages*
  - *Data access packages*
  - *Geo-spatial analysis*
  - *Advanced statistical analysis*
  - *Other*
- *Setting up and managing virtual environments with conda*
  - *Creating an environment from an environment.yml file*
  - *Activating an environment*
  - *Deactivating an environment*
  - *Determining your current environment*
  - *Viewing a list of your environments*
  - *Viewing a list of the packages in an environment*
  - *Installing new packages in an environment*
  - *Using pip in an environment*
  - *Sharing an environment*

– *Exporting the environment file*
– *Creating an environment file manually*
– *Removing an environment*

2.1 Installing python and packages

Python itself is rather lightweight. It contains some basic objects, and a few basic builtin packages. Nevertheless it is quite a powerful language off the shelf. For this course you will not need anything beyond the basic builtin python.

There are a number of ways to install python, but one of the best for scientists is via Anaconda. Anaconda is a free and open source distribution of data science packages for Python/R and can manage your packages, dependencies, and environments. There are two ways to install python from Anaconda either 1) simply install Anaconda, which has most of the packages you may need and many that you will never use or 2) install miniconda (a very light weight version) and then add the packages you need when you need them.

Both the full Anaconda and the Miniconda installations come with the core Python installation and the conda package manager. In addition to this, the full Anaconda installation comes with 1,400+ additional packages.

For this course we recommend installing miniconda as that is all you need and we discuss this later. As you progress it is good experience to start creating environments for your projects. In the long run it will save you time as you will not need to fight through dependency issues (where one packaged depends on a specific version of another).

Packages are really the ace in the hole when it comes to python. There are tens of thousands of packages that have been developed to make programming easier. Complex statistical analysis?, check; interactive data visualisation?, check. Basically if you need to do something, there's probably a package for that. For this course you will not be using any packages that are not already built into python. That said as soon as you begin really doing work in python you'll use packages galore.

In order to use packages, you have to install them. At the end of the day, the definitive expert on how to install a given package is that's package website, but most packages that you will be interested in using can be easily installed with either pip or conda which is discussed below.

2.2 Using python environments

Can you imagine working on different projects, each needing a different set of packages, and perhaps some of them requiring conflicting versions. Sounds like a nightmare. Using conda, rather than setting up one bloated python build to try and do everything you are better off building a number of virtual python environments.

A virtual python environment is both a python build and a number of packages. This build has a dedicated name, can host specific versions of a package, and even specific versions of python (e.g. 2.7 or 3.6). You can import and export virtual python environments as lightweight text files, which makes collaborating even easier. In addition to collaboration there are a couple of advantages to working in this style:

1. Reproducibility and clarity. When you finish up your project you can include an environment file and everyone will know exactly what you used to make your code.
2. Dependencies. You don't need to worry about competing dependencies ever again, you know that your code will run.
3. Safeguard from depreciation. You won't need to worry about dusting off some code only to realise that it won't run because some key function was depreciated.

A very detailed description and tutorial about python environments can be found on FreeCodeCamp.

2.3 Our recommended python installation for this course

This installation includes basic python and the IDE spyder

1. **Install miniconda**
   (a) go to https://conda.io/miniconda.html and download the appropriate python 3.6 installer and accept all of the defaults
2. **Create a virtual environment for this course (bpes) for basic python for environmental scientists**
   (a) open anaconda prompt and enter:

```
conda create -n bpes python=3.6 spyder
```

2. When conda asks you to proceed, type `y`:

```
proceed ([y]/n)?
```

3. That's it python and spyder for this course should now be installed. To use python with spyder, in the start menu (under anaconda) you should see spyder (bpes). Open that up and get cracking!

2.4 Our recommended python installation for further work

1. **If you have not, install miniconda**
   (a) go to https://conda.io/miniconda.html and download the appropriate python 3.6 installer and accept all of the defaults
2. **Create a virtual environment for your project**
   (a) create a .yml files from the packages you need below.
   (b) open an anaconda prompt
   (c) Create the environment from the `environment.yml` file:

```
  conda env create -f [environment.yml]

The first line of the ``yml`` file sets the new environment's
name. The ``environment.yml`` can also be the explicit path to the .yml␣
↪file.
```

   (d) enter y and press enter when prompted with 'are you sure'
3. That's it python and spyder for your specific project should now be installed. To use python with spyder, in the start menu (under anaconda) you should see a version of spyder followed by your virtual environment's name. Open that and get cracking!
4. Each time you start a new project go back to 2 and create a new virtual environment.

2.5 Build your own python environment file for environmental scientists

As a base for any environment file we suggest the following build:

```
name: [insert_your_enviroment_name_here]
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.6
  - spyder
  - numpy
  - matplotlib
  - pandas
  - scipy
```

This build has the core of pythons scientific data processing (python + numpy, pandas, and scipy) as well as the core data visualisation tool (matplotlib), and somewhat optionally, the spyder IDE. We default to the conda-forge channel, as it is often the best anaconda channel to make all of the packages play nice together.

Depending on what you need in your project you can add on any number of packages. Below, we've put together some tables of packages that we've found to be high quality and easily usable. Rather than re-producing the installation instructions, which could then go out of date, we've simply included a link to the package documentation. You can of course :ref: *add packages <course-env>* after you've built the environment. Just be sure to export a new environment file to hold in your git repository.

2.5.1 Advanced visualisation packages

package	utility / comments
bokeh	Interactive data visualisation
seaborn	Statistical data visualisation
holoviews	Simplified data visualisation for quick plotting

2.5.2 Data access packages

2.5.3 Geo-spatial analysis

2.5.4 Advanced statistical analysis

package	utility / comments
scikit-learn	Machine learning in python
statsmodels	Generalised statistical models in python

2.5.5 Other

package	utility / comments
scikit-image	Scientific image processing in python
networkx	Complex network analysis in python
flopy	Python interface for Modflow Suite models
Pyemu	Linear base model independent uncertainty analysis (e.g. PEST)

2.6 Setting up and managing virtual environments with conda

The instructions below on how use a conda environments are a simplified version of the instructions given here. You can read through the instructions, but they here more as a guide if/when you need them. For instructions on how to create the recommended python environment for this course, please go back to *this section*.

Use the Terminal or an Anaconda Prompt for the following steps.

1. To create an environment:

   ```
   conda create --name myenv
   ```

   NOTE: Replace `myenv` with the environment name.

2. When conda asks you to proceed, type `y`:

   ```
   proceed ([y]/n)?
   ```

This creates the myenv environment in `/envs/`. This environment uses the same version of Python that you are currently using, because you did not specify a version.

To create an environment with a specific version of Python:

```
conda create -n myenv python=3.6
```

2.6.1 Creating an environment from an environment.yml file

Use the Terminal or an Anaconda Prompt for the following steps.

1. Create the environment from the `environment.yml` file:

   ```
   conda env create -f environment.yml
   ```

The first line of the `yml` file sets the new environment's name. The `environment.yml` can also be the explicit path to the .yml file.

For details see :ref:'Creating an environment file manually

<create-env-file-manually>'.

2.6.2 Activating an environment

To activate an environment:

- On Windows, in your Anaconda Prompt, run `activate myenv`
- On macOS and Linux, in your Terminal Window, run `source activate myenv`

Conda prepends the path name `myenv` onto your system command.

2.6.3 Deactivating an environment

To deactivate an environment:

- On Windows, in your Anaconda Prompt, run `deactivate`
- On macOS and Linux, in your Terminal Window, run `source deactivate`

Conda removes the path name `myenv` from your system command.

TIP: In Windows, it is good practice to deactivate one environment before activating another.

2.6.4 Determining your current environment

Use the Terminal or an Anaconda Prompt for the following steps.

By default, the active environment—the one you are currently using—is shown in parentheses () or brackets [] at the beginning of your command prompt:

```
(myenv) $
```

If you do not see this, run:

```
conda info --envs
```

In the environments list that displays, your current environment is highlighted with an asterisk (*).

By default, the command prompt is set to show the name of the active environment. To disable this option:

```
conda config --set changeps1 false
```

To re-enable this option:

```
conda config --set changeps1 true
```

2.6.5 Viewing a list of your environments

To see a list of all of your environments, in your Terminal window or an Anaconda Prompt, run:

```
conda info --envs
```

OR

```
conda env list
```

A list similar to the following is displayed:

```
conda environments:
myenv                  /home/username/miniconda/envs/myenv
snowflakes             /home/username/miniconda/envs/snowflakes
bunnies                /home/username/miniconda/envs/bunnies
```

2.6.6 Viewing a list of the packages in an environment

To see a list of all packages installed in a specific environment:

- If the environment is not activated, in your Terminal window or an Anaconda Prompt, run:

  ```
  conda list -n myenv
  ```

- If the environment is activated, in your Terminal window or an Anaconda Prompt, run:

```
conda list
```

To see if a specific package is installed in an environment, in your Terminal window or an Anaconda Prompt, run:

```
conda list -n myenv scipy
```

2.6.7 Installing new packages in an environment

#. To install a new package in the environment .. code-block:: bash

conda install -n myenv scipy # install the package

1. To install a specific version of a package:

```
conda install -n myenv scipy=0.15.0
```

TIP: It's best to Install all the programs that you want in this environment at the same time. Installing 1 program at a time can lead to dependency conflicts.

2.6.8 Using pip in an environment

To use pip in your environment, in your Terminal window or an Anaconda Prompt, run:

```
conda install -n myenv pip
source activate myenv
pip <pip_subcommand>
```

2.6.9 Sharing an environment

You may want to share your environment with someone else—for example, so they can re-create a test that you have done. To allow them to quickly reproduce your environment, with all of its packages and versions, give them a copy of your `environment.yml file`.

2.6.10 Exporting the environment file

NOTE: If you already have an `environment.yml` file in your current directory, it will be overwritten during this task.

1. Activate the environment to export:
   - On Windows, in your Anaconda Prompt, run `activate myenv`
   - On macOS and Linux, in your Terminal window, run `source activate myenv`

   NOTE: Replace `myenv` with the name of the environment.
2. Export your active environment to a new file:

   ```
   conda env export > environment.yml
   ```

   NOTE: This file handles both the environment's pip packages and conda packages and you can replace the `environment.yml` with a path of your choosing.
3. Email or copy the exported `environment.yml` file to the other person.

2.6.11 Creating an environment file manually

You can create an environment file manually to share with others.

EXAMPLE: A simple environment file:

```
name: stats
dependencies:
  - numpy
  - pandas
```

EXAMPLE: A more complex environment file:

```
name: stats2
channels:
  - javascript
dependencies:
  - python=3.6   # or 2.7
  - bokeh=0.9.2
  - numpy=1.9.*
  - nodejs=0.10.*
  - flask
  - pip:
    - Flask-Testing
```

You can exclude the default channels by adding `nodefaults` to the channels list.

```
channels:
  - javascript
  - nodefaults
```

2.6.12 Removing an environment

To remove an environment, in your Terminal window or an Anaconda Prompt, run:

```
conda remove --name myenv --all
```

(You may instead use `conda env remove --name myenv`.)

To verify that the environment was removed, in your Terminal window or an Anaconda Prompt, run:

```
conda info --envs
```

The environments list that displays should not show the removed environment.

CHAPTER 3

Basic Python Objects, Variables, and Operators

3.1 Variables

Everything in python can be considered to be either a variable, an object, or an operator. An object can be everything from a number, to a function, to something more complex like a class. For the moment let's not worry too much about objects, in fact most of this course is about how to create and use the plethora of objects that exist in the python language. Briefly an operator is something that does something to a variable or an object (e.g. =, +, -, *). We'll talk about operators in a moment. Instead for now let's focus on the variable in python.

A python variable is basically the name we give to an object in a script. Assignment of a variable is simple using the = operator:

```
In [1]: x = 42  # the variable in this case is x and we have assigned the value of 42␣
↪to the variable
```

A variable can be named anything except one of the built in keywords of python. To get a list of these keywords you can use the help function:

```
In [2]: help('keywords')

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass
```

While you can name a variable anything, some options are better than others. As previously mentioned style is important. The PEP 8 standards suggest that variables should be lowercase, with words separated by underscores as necessary to improve readability. While PEP 8 isn't necessary, it is a really good habit to get into. It is also very important not to give a variable the name of any of the builtin functions and variables, otherwise you cannot use the builtin function again in your script. That said don't panic about knowing every builtin variable, most integrated development editors will raise some sort of warning when you overwrite a builtin name. Also if you try to use a builtin function again it will simply raise an exception, for example:

```
# now we'll be naughty and overwrite the help function, really don't do this...
In [3]: help = 42

# if we try to use the help function it will raise an exception
In [4]: help('keywords')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-9d2f16847670> in <module>()
----> 1 help('keywords')

TypeError: 'int' object is not callable
```

if you make this mistake, fix it in your script and reload you interpreter.

Why did this happen? It has to do with how python assigns variables. When we assigned the value of 42 to *x* above the number 42 was created in the computer's memory and the variable *x* was pointed to that memory via a unique object ID. python has a built in function id(), which allows us to see the this ID. This is helpful as we can see how python handles memory. Take a look at the example below:

```
In [5]: x = 42

In [6]: id(x)
Out[6]: 94152819945280

In [7]: y = x

In [8]: id(y)
Out[8]: 94152819945280

In [9]: x = 15

In [10]: y
Out[10]: 42

In [11]: id(x)
\\\\\\\\\\\\Out[11]: 94152819944416

In [12]: id(y)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[12]: 94152819945280
```

Note that when we set *y* = *x* all it did was point *y* to the same bit of computer memory that *x* is pointing to. When we re-assigned *x* (to 15) it points at a different part of memory leaving *y* unchanged with a value of 42. So when we overwrote the help function above, all we did was point the variable *help* to a new object (42)

When an object is no longer in use (e.g. no variables are pointing to it) a part of python called the garbage collector will remove the object from memory so your computer doesn't have too much on it's mind.

3.2 Numbers: Integers and Floats

There are three main ways to portray numeric values in python - integers, floats, and complex.

An Integers is as you would expect, a number without a decimal point (e.g. 1, 2, 3, or -5000). Floats on the other hand are numbers with a decimal point. We won't really talk about complex numbers here, but it is useful to know that python can handle complex numbers.

```
In [13]: type(1)
Out[13]: int

In [14]: type(3.1415)
\\\\\\\\\\\\\Out[14]: float

In [15]: type(3.)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[15]: float
```

the type function in python tells you what type a given object or variable is.

There are a number of operations that you can do with numeric values:

```
In [16]: x = 2

In [17]: y = 3.5

In [18]: z = -5

In [19]: x + y  # the sum of x and y
Out[19]: 5.5

In [20]: x - y  # the difference of x and y
\\\\\\\\\\\\\Out[20]: -1.5

In [21]: x * z  # the product of x and z
\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[21]: -10

In [22]: z / x  # the quotient of z and x (2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[22]: -2.5

In [23]: z // x  # the floored quotient of z and x (3)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23]: -3

In [24]: z % x  # the remainder of z / x
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[24]: 1

In [25]: abs(z)  # the absolute value of z
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[25]:␣
↪5

In [26]: int(y)  # the integer of y rounded down
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[26]:␣
↪3

In [27]: float(x)  # x converted to a float
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Ou
↪2.0

In [28]: z ** x  # z to the power of x
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪25

In [29]: x = 42
```

(continues on next page)

(continued from previous page)

```
In [30]: x += 60 # add 60 to x and assign the new value back to x

In [31]: x
Out[31]: 102

In [32]: x = 10

In [33]: x *= 10 # multiply x by 10 and assign the new value back to x

In [34]: x
Out[34]: 100
```

some notes on these operations:

1. You can mix numeric types. Where possible python tries to maintain the numeric type throughout the operation, but it will change the type if needed (e.g. from int to float).
2. The behaviour of division in python 2.7 is different to python 3.6. This course assumes python 3.6 see more here.
3. Floored means always towards - infinity so -1.1 floored is -2 and 1.9 floored is 1.
4. Order of operation applies to mathematical formulas in python as normal so:

```
In [35]: 5 / (3 + 2)
Out[35]: 1.0

In [36]: 4 ** (1 / 2)
\\\\\\\\\\\\\Out[36]: 2.0
```

3.3 Boolean

A boolean value in python is either True or False (case sensitive). As with numeric data there a several basic operations that can be preformed on boolean data

```
In [37]: True or False
Out[37]: True

In [38]: True or True
\\\\\\\\\\\\\\Out[38]: True

In [39]: True and True
\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[39]: True

In [40]: True and False
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[40]: False

In [41]: not True
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[41]: False

In [42]: not False
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[42]: True

In [43]: all([True, True, False]) # this uses a list, which will be described in the␣
↪next section
```

(continues on next page)

(continued from previous page)

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[43]:␣
↪False

In [44]: any ([True, False, False]) # this uses a list, which will be covered in the␣
↪next section
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪True
```

order of operations also applies to boolean operations, so:

```
In [45]: True and (True or False)
Out[45]: True

In [46]: False or (True and False)
\\\\\\\\\\\\\\Out[46]: False
```

Boolean values can be converted to integers and floats where True = 1 and False = 0

```
In [47]: int(True)
Out[47]: 1

In [48]: int(False)
\\\\\\\\\\\Out[48]: 0
```

3.4 Strings

Strings are made up of different characters (e.g. a, b, c, %, &, ?, etc.). Every sentence ever written can be considered as a string. You can make strings in a number of ways by wrapping characters ' and " so for example:

```
In [49]: x = 'my string'

In [50]: y = "also my string"

In [51]: z = "my string can contain quotes of the other type 'like this one'"

In [52]: x
Out[52]: 'my string'

In [53]: y
\\\\\\\\\\\\\\\\\\\\\Out[53]: 'also my string'

In [54]: z
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[54]: "my string can contain quotes␣
↪of the other type 'like this one'"

In [55]: x = """
   ....: triple " or ' can define a string that splits
   ....: a number of lines
   ....: like this
   ....: """
   ....:

In [56]: x  # \n is the symbol for new line.   \' is the symbol for '
Out[56]: '\ntriple " or \' can define a string that splits\na number of lines\nlike␣
↪this\n'
```

(continues on next page)

(continued from previous page)

```
# numbers can be represented as strings
In [57]: x = '5'

In [58]: x
Out[58]: '5'

# and number stings can be converted to floats and ints
In [59]: int(x)
\\\\\\\\\\\\\Out[59]: 5

In [60]: float(x)
\\\\\\\\\\\\\\\\\\\\\\\\Out[60]: 5.0

# though python isn't smart enough to convert everything to a numeric value and will␣
↪raise an exception
In [61]: x = 'five'

In [62]: int(x)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-62-acaae37f5ab0> in <module>()
----> 1 int(x)

ValueError: invalid literal for int() with base 10: 'five'
```

There are many different operators and ways to manage strings, for more information please see *this chapter on strings*

3.5 The print function

Up to now in order to see the contents of a variable we have simply been calling the variable. This works fine in an interactive python environment, but when running a python script from start to finish you need the print function. The print function is easy to use and will simply print the variable, so for instance:

```
In [63]: x = 'some string'

In [64]: print(x)
some string

In [65]: print(1,1,2,2,3)
\\\\\\\\\\\\1 1 2 2 3
```

3.6 The Python None

In python there is a built in value for an absence of a value, defined as *None* (case sensitive). You likely will not encounter this value until you start working with functions, but it's important to know that it exists.

CHAPTER 4

The List

4.1 What's a list

One of the most fundamental objects in python is the list. A list simply holds multiple objects, these can be of any type. A list can be generated in two ways:

```
# 1) with the [ ]
In [1]: my_list = [1, 2, 'some value', True]

In [2]: my_list
Out[2]: [1, 2, 'some value', True]

# 2) by converting some other object (an iterable) to a list with the list function
In [3]: my_other_list = list((1, 2, 3, 4, 5))  # this uses a tuple, which will be␣
↪discussed later

In [4]: my_other_list
Out[4]: [1, 2, 3, 4, 5]

# a list can even hold other lists (nesting)
In [5]: my_nested_list = [[1, 2], [3, 4], [5, 6]]

In [6]: my_nested_list
Out[6]: [[1, 2], [3, 4], [5, 6]]
```

4.2 length, indexing, and slicing

You'll notice in the example above that we mentioned an iterable. While it's not strictly the definition you can think of an iterable as any sort of object that is a container for multiple objects. The number of objects in an iterable like a list is also known as the length of the list. Unsurprisingly there is a python function to get the length of an iterable like a list called *len()*

```
In [7]: my_list
Out[7]: [1, 2, 'some value', True]

In [8]: len(my_list) # using the lists defined above
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[8]: 4

In [9]: my_other_list
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9]: [1, 2, 3, 4, 5]

In [10]: len(my_other_list)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[10]: 5

# note for nested lists it will only give the length of the outermost list
In [11]: len(my_nested_list)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[11]:␣
↪3
```

Great, I've put my very important data into a list, but how on earth can I get it out? Through indexing and slicing. Indexing is getting out a single item in the list, while slicing is getting out a subset of the list. Python is a zero based indexing language. That's a big phrase to say that the first (or more accurately the zeroth) item is given a position index of 0, so for example *my_list*:

Item	1	2	'some value'	True
Position	0	1	2	3

```
In [12]: my_list
Out[12]: [1, 2, 'some value', True]

# to index things use brackets []
In [13]: my_list[0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[13]: 1

In [14]: my_list[3]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[14]: True

# trying to index something that does not exist e.g. my_list[4] will raise an␣
↪exception
# to access items from a sublist in a nested list you use chained brackets
In [15]: my_nested_list
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[15]: [[1, 2], [3, 4],
↪ [5, 6]]

In [16]: my_nested_list[0][1] # returns the second item of the first list in my_
↪nested_list
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[1
↪2

# you can also access things from the back end of the list:
In [17]: my_list[-1] # the last item
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪True

In [18]: my_list[-2] # the second to last item
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪'some value'
```

(continues on next page)

(continued from previous page)

```
# slicing uses brackets and a :
In [19]: my_list[0:2]  # everything from the zeroth item up to, but not including the␣
↪2nd item
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[1, 2]

In [20]: my_list[1:]  # everything from the first item up to and including the last␣
↪item
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[2, 'some value', True]

In [21]: my_list[:3]  # everything from the first item up to, but not including the␣
↪3rd item
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[1, 2, 'some value']
```

4.3 list specific operations and functions

Almost every python object has some functions built into them that will act on the instance of the object. These functions are accessed with a '.', e.g. my_var.some_function(). Lists are no different, here we'll showcase the operators and keywords associated with lists and two most important functions associated with lists, *.append()* and *.extend()*:

```
# adding two lists creates a new list with all of the elements joined together
In [22]: list1 = [1,2,3]

In [23]: list2 = [4,5,6]

In [24]: list1 + list2
Out[24]: [1, 2, 3, 4, 5, 6]

# multiplying a list and an integer creates a new list with the previous list␣
↪repeated n times
In [25]: list1 = ['spam']

In [26]: list1*3
Out[26]: ['spam', 'spam', 'spam']

# the in keyword asks the question is some element in a list and returns a boolean
In [27]: my_list = [1,2,3,[4,5]]

In [28]: 2 in my_list
Out[28]: True

In [29]: 4 in my_list  # note that only the top most layer of the list is searched
\\\\\\\\\\\\\\Out[29]: False

In [30]: [4,5] in my_list  # it will also search for more complex objects (e.g. other␣
↪lists)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[30]: True

# .append() adds something to a list
In [31]: my_list = []  # create an empty list

In [32]: my_list.append(1)
```

(continues on next page)

(continued from previous page)

```
In [33]: my_list
Out[33]: [1]

In [34]: my_list.append('my string')

In [35]: my_list
Out[35]: [1, 'my string']

In [36]: my_list.append([1,2])

In [37]: my_list
Out[37]: [1, 'my string', [1, 2]]
```

Note that when you append a list to a list it creates a nested list. If instead you want to append all of the values of an iterable (like a list) to another list then you need to use the *.extend()* function:

```
In [38]: my_list = []   # create an empty list

In [39]: my_list.extend([1,2,3,4])

In [40]: my_list
Out[40]: [1, 2, 3, 4]

# note if you try to pass a non-iterable to extend it will raise an exception
```

4.4 The tuple

There is another basic python object, the tuple. A tuple is denoted similarly to a list, but using () instead of [] tuples are generated and sliced exactly the same as lists, but they are immutable. This concept is beyond this lesson, but will be covered in *here*

CHAPTER 5

Dictionaries

5.1 What's in a dictionary

Another key basic object in python is the dictionary. Dictionaries are iterables, but unlike lists (and tuples if you got that far) values in a dictionary are indexed by a user specified key rather than a positional argument. the default structure of a dictionary is {key: value}. The value can be any object in python (numbers, strings, boolean, lists, and so on). Keys are a bit more proscriptive; the rules are a bit more complex, but for beginners it's normally enough to know that numbers and strings are able to be keys. For more details on exactly what python objects can be keys in a dictionary see: Why Lists Can't be Dictionary Keys. Note as of python version 3.6 dictionaries will remember the order that the data was input, previously they were un-ordered. Now for some examples:

```
In [1]: my_dict = {'key': 'value'}

In [2]: my_dict
Out[2]: {'key': 'value'}

# to access data from a dictionary you use brackets and the key
In [3]: my_dict['key']
\\\\\\\\\\\\\\\\\\\\\\\\\Out[3]: 'value'

# it is possible to assign new values to a dictionary
In [4]: my_dict['new_key'] = 'new_value'

In [5]: my_dict['new_key']
Out[5]: 'new_value'

# it is also possible to overwrite existing keys
In [6]: my_dict['key'] = 'peanut butter'

In [7]: my_dict['key']
Out[7]: 'peanut butter'

# you can also create dictionaries from nest lists (or tuples) of key, value pairs␣
↪and the dict() function:
```

(continues on next page)

(continued from previous page)

```
In [8]: temp = [
   ...:          ['peanut butter', 'jam'],
   ...:          ['eggs', 'bacon'],
   ...:          ['muslie', 'milk']
   ...:          ]
   ...:

In [9]: my_food = dict(temp)

In [10]: my_food
Out[10]: {'peanut butter': 'jam', 'eggs': 'bacon', 'muslie': 'milk'}

In [11]: my_food['eggs']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[11]: 'bacon'

# just like lists dictionaries have length
In [12]: len(my_food)   # returns the number of key, value pairs
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[12]:␣
↪3

# the keyword *in* also works, but only looks in the dictionaries keys
In [13]: 'eggs' in my_food
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out
↪True

In [14]: 'bacon' in my_food
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪False
```

5.2 Important dictionary functions

Just like lists, dictionaries have a number of useful built in functions here we'll showcase the four most important - *.update()*, *.keys()*, *.values()*, *.items()*

```
# .update adds the entries of a second dictionary to the first
In [15]: my_dict = {'Bert': 'Ernie'}

In [16]: another_dict = {'Sherlock Holmes': 'Watson', 'Batman': 'Robin'}

In [17]: my_dict.update(another_dict)

In [18]: my_dict
Out[18]: {'Bert': 'Ernie', 'Sherlock Holmes': 'Watson', 'Batman': 'Robin'}

# .keys(), .values(), .items() are all about accessing all of the keys, values,
# and (key, value) pairs in a dictionary, respectively.
# they are useful to see what's in your dictionary and for iteration which we'll talk␣
↪about later
In [19]: my_dict.keys()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[19]:␣
↪dict_keys(['Bert', 'Sherlock Holmes', 'Batman'])

In [20]: my_dict.values()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪dict_values(['Ernie', 'Watson', 'Robin'])
```

(continues on next page)

(continued from previous page)

```
In [21]: my_dict.items()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
 ↪dict_items([('Bert', 'Ernie'), ('Sherlock Holmes', 'Watson'), ('Batman', 'Robin')])

# note that the behaviour of these functions are slightly different in python 2.7
```

CHAPTER 6

Strings, the good, the bad, and the ugly

6.1 String indexing

The character components of a string can be accessed the same way that a list can be. Each character as a positional assignment, so for example:

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
h	e	l	l	o		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10

You can index and slice a string just like you would a list, so for example:

```
In [1]: my_string = 'hello world'

In [2]: my_string[0]
Out[2]: 'h'

In [3]: my_string[-1]
\\\\\\\\\\\\Out[3]: 'd'

In [4]: my_string[3:9]
\\\\\\\\\\\\\\\\\\\\\\\\Out[4]: 'lo wor'
```

6.2 String operators & functions

Strings have access to two operators + and *, and the behaviour is just like lists, so:

```
In [5]: my_string = 'spam'

In [6]: my_string + 'eggs'
```

(continues on next page)

(continued from previous page)

```
Out[6]: 'spameggs'

In [7]: my_string * 5
\\\\\\\\\\\\\\\\\\\Out[7]: 'spamspamspamspamspam'
```

There are a number of useful functions associated with string objects. Python strings are case sensitive so:

```
In [8]: 'spam' == 'SPAM'
Out[8]: False

In [9]: 'spam' != 'SPAM'
\\\\\\\\\\\\\\Out[9]: True
```

Luckily there are a number of functions to assist with various capitalisation choices that you may want:

```
In [10]: my_string = 'your mother was a haMster and your father smElt of elderberries'

In [11]: my_string.lower() # all characters to lower case
Out[11]: 'your mother was a hamster and your father smelt of elderberries'

In [12]: my_string.upper() # all characters to upper case
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[12]:
↪'YOUR MOTHER WAS A HAMSTER AND YOUR FATHER SMELT OF ELDERBERRIES'

In [13]: my_string.capitalize() # first character to upper
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪'Your mother was a hamster and your father smelt of elderberries'

In [14]: my_string.title() # the first character of each word capitalized
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪'Your Mother Was A Hamster And Your Father Smelt Of Elderberries'
```

There are also functions to help break up and put together strings.

```
In [15]: print(my_string)
your mother was a haMster and your father smElt of elderberries

# .split() splits the sting into a list of stings by removing the characters␣
↪specified (a space in this case)
In [16]: temp = my_string.split(' ')

In [17]: type(temp)
Out[17]: list

In [18]: print(temp)
\\\\\\\\\\\\\\['your', 'mother', 'was', 'a', 'haMster', 'and', 'your', 'father',
↪'smElt', 'of', 'elderberries']

# ''.join() takes as list of strings and joins them together with the characters in␣
↪proceeding string
In [19]: '-'.join(temp)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪'your-mother-was-a-haMster-and-your-father-smElt-of-elderberries'
```

Finally there are some useful functions to remove characters from strings:

```
# ''.strip() removes characters from the beginning and/or end of a sting, the default␣
↪is a space
In [20]: test = '*********lunch*lunch***********'

In [21]: test.strip('*')
Out[21]: 'lunch*lunch'

# ''.replace(old, new) finds all instances of the old string and replaces it with the␣
↪new string
In [22]: test.replace('lunch', 'dinner')
\\\\\\\\\\\\\\\\\\\\\\\Out[22]: '*********dinner*dinner***********'
```

6.3 Formatted output

In python there is an elegant way to create formatted string outputs using the *''.format()* method. The basic premise of the format method is that you pass the arguments in the function into specific places into the proceeding string. So for example:

```
# passing arguments('spam' and 'eggs') by position into the {}
In [23]: print("I don't like {} or {}".format('spam', 'eggs'))
I don't like spam or eggs

# passing arguments by keywords into the {}
In [24]: print("""{p} ran away. {adv} ran away away.
   ....: When danger reared it’s ugly head, he {adv} turned his tail and fled.
   ....: {p} turned about and gallantly he chickened out""".format(p='Brave Sir Robin
↪', adv='Bravely'))
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\Brave Sir Robin ran away. Bravely ran away away.
When danger reared it’s ugly head, he Bravely turned his tail and fled.
Brave Sir Robin turned about and gallantly he chickened out

# you can pass anything into the .format method that can be passed to the print␣
↪function
In [25]: my_list = [1,2,3]

In [26]: my_dict = {'Bert': 'Ernie'}

In [27]: my_number = 42

In [28]: my_new_string = 'like lists : {}, dictionaries: {}, and numbers: {}.'.
↪format(my_list, my_dict, my_number)

In [29]: print(my_new_string)
like lists : [1, 2, 3], dictionaries: {'Bert': 'Ernie'}, and numbers: 42.
```

Passing values back into a string is useful enough, but the .format() method of strings gives significantly more control over how the object is transformed into a sting. Below we’ll cover the most commonly used formatting options for environmental scientists, but For a deeper dive into the full capabilities of python formatted output, check out this lovely tutorial.

```
In [30]: '{:4d}'.format(42)   # at least 4 digits, padded with spaces
Out[30]: '  42'
```

(continues on next page)

(continued from previous page)

```
In [31]: '{:04d}'.format(42)   # at least 4 digits, padded with zeros
\\\\\\\\\\\\\\\\Out[31]: '0042'

In [32]: '{:4d}'.format(42666666) # note that this can go beyond 4 digits
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[32]: '42666666'

In [33]: '{:6.2f}'.format(3.14159265)   # at least 6 digits, padded with spaces, with␣
↪exactly 2 digits after the decimal point
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[33]: '  3.14'

In [34]: '{:06.2f}'.format(3.14159265)   # as above but padded with zeros
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[34]: '003.14
↪'
```

CHAPTER 7

Conditional Statements

Conditional statements in python allow code to do different things in different circumstances. These statements use the *if*, *elif*, and *else* keywords. The easiest way to show conditional statements in python is through a simple example:

```
# what will i do?
In [1]: raining = True

In [2]: if raining:  # start of the if statement
   ...:      print('clean the garage')  # what I will do if the condition is true
   ...: else: # start of the otherwise (else) clause
   ...:      print('weed the garden')  # what I will do if raining == False
   ...:
clean the garage
```

In human speak this says that if it is raining (which it is), I'll clean the garage otherwise (else), I'll weed the garden. Python identifies that something is in the if clause (or the else clause) by the indentation of the code. After the if statement the code is indented by exactly 4 spaces be for the actions (print('clean the garage')). While this may seem pedantic, at the end of the day it creates much clearer, human readable code that is less prone to typos and other errors.

Before we can get onto more complex conditional statements we need to talk about the conditional operators:

Operator	meaning	applies to
==	equals	most objects
!=	does not equal	most objects
<	less than	numbers
>	greater than	numbers
<=	less than or equal	numbers
>=	greater than or equal	numbers
all()	all elements True	iterable of boolean
any()	any element True	iterable of boolean

Armed with these comparisons and the *elif* keyword we can create much more complicated conditional statements. *elif* is an abbreviation for else if and allows the user to specify more conditions, For example:

```
In [3]: weather = 'sunny'

 # what will I do?
In [4]: if weather == 'raining':
   ...:      print('clean the garage')
   ...: elif weather == 'sunny':
   ...:      print('go to the beach')
   ...: elif weather == 'raining hellfire':
   ...:      pass  # the python keyword pass will move past the conditional statement␣
↪without doing anything
   ...: else:
   ...:      print('weed the garden')
   ...: print('ok then')
   ...:
go to the beach
ok then
```

CHAPTER 8

Loops in Python

Ever been in the position where you have to do the same thing again, and again, and again. It's painful and it feels like there must be a better way. There is, it's called a loop.

8.1 For loops in python

The main loop that environmental scientists will use is a for loop. The standard format for a for loop is:

```
for i in iterable:
    action
```

let's break that down. The iterable could be any python iterable or iterator (for now don't worry about the difference), for instance a list. Python will then create a variable (i) pointing to the first object in the list. Next python will then take some action that is in the indented (4 spaces again) line of code. After all the action code if completed python will then set i to the second item in the list and repeat. A simple working example is:

```
In [1]: menu = ['Egg and Spam', # the iterable
   ...:         'Egg, bacon and Spam',
   ...:         'Egg, bacon, sausage and Spam',
   ...:         'Spam, bacon, sausage and Spam',
   ...:         'Spam, egg, Spam, Spam, bacon and Spam',
   ...:         'Spam, Spam, Spam, egg and Spam',
   ...:         'Spam, Sausage, Spam, Spam, Spam, Bacon, Spam, Tomato and Spam',
   ...:         'Spam, Spam, Spam, Spam, Spam, Spam, baked beans, Spam, Spam, Spam␣
↪and Spam']
   ...:

In [2]: for dish in menu:  # dish is the new variable
   ...:     print(dish)  # print is the action
   ...:
Egg and Spam
Egg, bacon and Spam
Egg, bacon, sausage and Spam
```

(continues on next page)

(continued from previous page)

```
Spam, bacon, sausage and Spam
Spam, egg, Spam, Spam, bacon and Spam
Spam, Spam, Spam, egg and Spam
Spam, Sausage, Spam, Spam, Spam, Bacon, Spam, Tomato and Spam
Spam, Spam, Spam, Spam, Spam, Spam, baked beans, Spam, Spam, Spam and Spam

In [3]: print("I don't like spam!")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪don't like spam!
```

8.2 Useful builtins for for loops

Python has a number of builtin functions to assist iteration in for loops. First is the range function which can be used with 1-3 arguments to generate a sequence of integers. The sequence is held in a special object built for iteration, but we can easily transform it into a list to give an idea of what is in the range.

```
In [4]: list(range(10)) # the integers from zero up to, but not including 10
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [5]: list(range(5, 11)) # the integers from 5 up to, but not including 11
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[5]: [5, 6, 7, 8, 9, 10]

In [6]: list(range(5, 26, 5)) # every 5th integer from 5 up to, but not including 26
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[6]: [5, 10, 15,
↪ 20, 25]

# the range function in action
In [7]: for i in range(2,11,2):
   ...:     print( i * 10)
   ...:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\20
40
60
80
100
```

The second is the zip function, which combines the elements of n equal length iterables together so that they can be accessed at the same time.

```
In [8]: data1 = ['batman', 'bert', 'calvin']

In [9]: data2 = ['robin', 'ernie', 'hobbes']

In [10]: list(zip(data1,data2))
Out[10]: [('batman', 'robin'), ('bert', 'ernie'), ('calvin', 'hobbes')]

# using the zip function in a for loop is easy
In [11]: for lead, sidekick in zip(data1, data2):
   ....:     print('{} & {}'.format(lead, sidekick))
   ....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\batman & robin
bert & ernie
calvin & hobbes
```

In this example rather than the for loop creating 1 new variable it creates two (lead, sidekick) for the two components of the zip object. You can zip togeather as many iterables a you need. This examples also uses the .format function of strings which is explained *here*

The enumerate function is very similar to the zip function. It essentially numbers the elements of the iterable passed to it.

```
In [12]: for i, sidekick in enumerate(data2):
   ....:     print('{} of {}'.format(i+1, len(data2)))
   ....:     print(sidekick)
   ....:
1 of 3
robin
2 of 3
ernie
3 of 3
hobbes
```

Lastly in the *lesson on dictionaries* we mentioned the *.items()* function of the dictionary which makes it easy to iterate through a dictionaries keys and values.

```
In [13]: my_dict = {'peanut butter': 'jam', 'eggs': 'bacon', 'muslie': 'milk'}

In [14]: for key, value in my_dict.items():
   ....:     print('{} is the key to {}'.format(key, value))
   ....:
peanut butter is the key to jam
eggs is the key to bacon
muslie is the key to milk
```

There is another package which has even more tools for easy iteration with for loops called itertools. It is beyond the scope of this course, but there is more information here.

8.3 While loops in Python

There is another type of loop in python called a while loop. A while loop continues to iterate until a condition becomes false. For environmental scientists, this loop isn't used that regularly, but it is important to at least know of it's existence. More detailed information about the while loop can be found here.

```
In [15]: my_number = 0

In [16]: while my_number < 3:
   ....:     print(my_number)
   ....:     my_number +=1
   ....:
0
1
2
```

CHAPTER 9

Functions in Python

9.1 What is a function?

A function is a block of organised, contained, reusable code. You can imagine using the same block of code, copying and pasting is again and again in your script. What are the chances that in one of those copy and pastes you'll make a mistake - almost 100%! One of the basic principles in good programming is "do not to repeat yourself" both to avoid mistakes, but also to make code more human readable. Functions allow you to wrap up code into a package that you can use again and again or allow you to use other people's work to make the job quicker. During the course so far, we've already showcased a number of functions, like the print() function. We have also used methods which are functions that are tied to a specific object instance, like *[].append()*.

Before we get into creating your own functions, let's get a better handle on the structure of functions in python. The general framework of a function is:

```
def function_name(*args, **kwargs):
    outdata = some actions
    return outdata
```

In human speak. The def keyword lets python know you're creating a function named *function_name*, which takes as inputs some *args* and *kwargs*. An arg is a positional argument, who's value is defined by the position in the function call, so in this fictional function call:

```
new_val = function_name(1, 2, trend_type=3)
```

1 is the first arg, 2 is the second arg. 3 on the other hand is a kwarg or keyword argument; the value of this argument is defined by the keyword trend_type. Following the : in the function definition, there is some indented code that typically does some action and creates a new variable. The end of the function is typically marked by the *return* keyword. This tells python to create a variable (new_val) and point it to the object that was defined as outdata inside the function. We need to return the object of outdata because as a general rule variables created outside a function are not accessible inside a function unless they are passed in as arguments, and those created inside a function are only accessible outside the function if they are explicitly passed out of the function by the keyword *return*.

9.2 Understanding function documentation

There are heaps of already developed functions that probably do almost exactly what you need to do, but in order to use them you need to understand what an existing function does and it's quirks. The best source of knowledge for this is the function documentation. It can take a bit of time to get good at interpreting documentation, so let's make a start of it. To start we'll look at the the built in function to enumerate iterables *enumerate()*. Original documentation is here, but it is included here as well:

enumerate(iterable, start=0)

> Return an enumerate object. iterable must be a sequence, an iterator, or some other object which supports iteration. The __next__() method of the iterator returned by enumerate() returns a tuple containing a count (from **start** which defaults to 0) and the values obtained from iterating over **iterable**.

Let's look at what the first line of the documentation *enumerate(iterable, start=0)* is telling us. First enumerate can take up to two arguments (*iterable* and *start*), second we need to pass at least one argument (*iterable*) as it has no default value (it isn't equal to anything), and third that if we do not pass a value for *start* it will be set equal to 0.

The body of the text explains what the function does (creates an object, which for our purposes can be converted to a list of tuples that contain a number, starting from start) It also gives more details on the arguments (in this case that iterable, must be some object that supports iteration). So armed with this knowledge are you surprised by the result of:

```
In [1]: my_list = ['a','b','c']

# remember the list() here is just being used to get away from the unnecessary
# complexity of the enumerate object
In [2]: list(enumerate(my_list, start=10)) # FYI the space after the comma and no␣
↪spaces between the '=' is pep8 standard
Out[2]: [(10, 'a'), (11, 'b'), (12, 'c')]

# above we passed the arguments as show in the documentation
# we can also pass everything as kwargs in any order we choose
In [3]: list(enumerate(iterable=my_list, start=10))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[3]: [(10, 'a'), (11, 'b'), (12, 'c')]

In [4]: list(enumerate(start=10, iterable=my_list))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[4]:␣
↪[(10, 'a'), (11, 'b'), (12, 'c')]

# or everything as args, but it will break if we shift the order
In [5]: list(enumerate(my_list, 10))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[(10, 'a'), (11, 'b'), (12, 'c')]
```

Let's look at one more, the built in print function. The actual print function documentation can be found here, but it's also copied below:

print(*objects, sep=' ', end='n', file=sys.stdout, flush=False)

> Print objects to the text stream file, separated by sep and followed by end. sep, end, file and flush, if present, must be given as keyword arguments.
>
> All non-keyword arguments are converted to strings like str() does and written to the stream, separated by **sep** and followed by **end**. Both **sep** and **end** must be strings; they can also be None, which means to use the default values. If no objects are given, print() will just write end.
>
> The **file** argument must be an object with a write(string) method; if it is not present or None, sys.stdout will be used. Since printed arguments are converted to text strings, print() cannot be used with binary mode file objects. For these, use file.write(...) instead.

Whether output is buffered is usually determined by file, but if the **flush** keyword argument is true, the stream is forcibly flushed.

Changed in version 3.3: Added the flush keyword argument.

Let's look at the *objects in the first line of the documentation. This effectively means that all positional arguments passed to print will be treated the same way. The keyword arguments work just like they would in enumerate, but because there is *objects they obviously have to be passed as kwarg.

```
In [6]: print(1,2,3,4,5,6)
1 2 3 4 5 6

In [7]: print(1,2,3,4,5,6, sep='!', end='ok\n') # the \n is a new line so the output␣
↪prints normally
\\\\\\\\\\\\1!2!3!4!5!6ok
```

Object documentation can be found using the *help()* function (e.g. *help(enumerate)*), or by simply googling the object in question. We typically find googling the function to be more effective as it's often much easier to read. The final word of wisdom about function documentation is to treat it a bit like a puzzle. If you don't quite understand what a function does, spend some time playing with it in a console. See how it reacts to different inputs, try to break it, and if you still can't understand what's going on it's time to check google and stack overflow.

9.3 Creating your own function in python

So we've talked a lot about using functions that others have build, but what about creating your own. If you find yourself copying and pasting code a lot, it's probably time to make a function. My personal guideline is that if I've used the same bit of code, with or without minor tweaks, three times, then it's time to package it up as a function. Looking back at the basic structure of a function:

```
def function_name(*args, **kwargs):
    outdata = some actions
    return outdata
```

Defining your own is not that challenging. As an example let's define a function to convert temperature in fahrenheit to celsius or kelvin:

```
# defining the function
In [8]: def fahrenheit_to_ck(temp, out_c=True):
   ...:     """
   ...:     convert temperature in degrees fahrenheit to celsius or kelvin
   ...:     :param temp: the temperature in fahrenheit as float
   ...:     :param out_c: boolean, if True convert to celsius, if False kelvin
   ...:     :return: temperature in c or k, float
   ...:     """
   ...:     c = (temp - 32) * 5 / 9
   ...:     k = c + 273.15
   ...:     if out_c:
   ...:         return c
   ...:     else:
   ...:         return k
   ...:
```

```
# Now to use the function
In [9]: print(fahrenheit_to_ck(451))  # use the default and return celsius
232.77777777777777
```

(continues on next page)

(continued from previous page)

```
In [10]: print(fahrenheit_to_ck(451, False)) # return kelvin instead
\\\\\\\\\\\\\\\\\\\505.92777777777775
```

The *def fahrenheit_to_ck(temp, out_c=True):* tells python that a function called fahrenheit_to_ck is being created, that it takes two arguments (*temp*, *out_c*) and that *out_c* has a default value of *True*. The name of you function should ideally be short (this one is pushing it), but descriptive, made of lowercase letters separated with underscores where needed for clarity. The next few lines wrapped in triple quotation marks is the docstring. The docstring is in built documentation for the function. It's not necessary, but it is **highly** encouraged. It helps anyone else (including future you) understand what your function does. With the docstring it becomes easy to understand what the argument *out_c* does. After you define a function in your script you can use it anywhere below the function definition in your script. For this reason, and for convention it is best to put function definitions at the top of any script. You can import your function to use in other scripts, but that will be covered in a future *lesson*

9.4 using *args and **kwargs

When you are looking at other peoples code you may find function calls that include one or more *s. This is simply using the *args and **kwargs format. let's look at an example:

```
In [11]: things_to_print = ['Spam', 'Spam', 'Spam', 'egg', 'and Spam'] # the args,␣
↪must be iterable, typically a tuple, though a list also works

In [12]: how_to_print = {'sep':'-', 'end':'!\n'} # the kwargs, must be dictionary

# simple printing
In [13]: print(things_to_print)
['Spam', 'Spam', 'Spam', 'egg', 'and Spam']

# using *args
In [14]: print(*things_to_print)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Spam Spam Spam egg and Spam

# using *args and **kwargs
In [15]: print(*things_to_print, **how_to_print)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Spam-Spam-
↪Spam-egg-and Spam!
```

So what is happening here? The * in front of things_to_print tells python to take all of the values out of the iterable (in this case a list) and pass them as positional arguments (e.g. the first item in the list becomes the first positional argument and so on). Note that you are no longer printing a list when using * in front of things_to_print. Remember that kwargs are defined by a keyword and a value, which is not so different than a dictionary. Here how_to_print has keys that exaclty match some of the keywords of the print function (*sep*, *end*) and has the values to be used as the values of those keyword arguments. the **how_to_print, simply tells python to use the keys and values of the dictionary to set the function kwargs.

CHAPTER 10

Using Packages in python

10.1 External packages

We've already talked about installing specific packages, but simply installing a package on your computer does not allow you to use the package in a script - you also need to import the package. This is very much in line with every other scripting language and it provides two main functions: 1) it is explicit what packages are used for a given script and 2) it lowers the overhead of any script (you don't need to load in everything you've installed). Before we explain how to import things, we need to have a quick discussion around namespaces.

A namespace is a naming convention that python uses to avoid ambiguity. Simply put, a namespace is a name given to a collection of functions and variables. You can imagine it, abstractly, as family names. Consider the Howard's and the Smiths

Howard	Smith
Larry	Bill
Moe	Larry
Curly	Carl

Both families ahve a member called Larry, if I just tell you I'm going to see Larry, you don't know who I mean. Instead if I say I'm going over to the Howard's then it's clear which Larry I mean. For a python example, both builtin python and the package Numpy have functions called min, max, and abs. Without a namespace there would be no way to distinguish between them, but if I say min vs numpy.min it's clear as day.

Let's look at some examples of how to import packages using the package math, which unsurprisingly has a whole host of mathematical functions and objects. There are three ways to import specific packages:

1. Importing selected items from a package to the current namespace

```
In [1]: from math import pi  # just import pi

In [2]: print(pi)
3.141592653589793
```

This is useful if you know you only need a few select functions from a package, but does run the risk of overwriting a function that is already present, for example *from numpy import min* would overwrite the builtin min function.

2. Importing everything from a package to the current namespace

```
In [3]: from math import *  # import everything

In [4]: print(pi)
3.141592653589793
```

This is really bad form in most scripts as you run a serious risk that you'll overwrite something you don't want to. Also if you import everything from multiple packages it is less clear where a given function or variable came from.

3. Importing a namespace (with or without a name change)

```
import math
```

```
In [5]: print(math.pi)
3.141592653589793

In [6]: import math as m  # import the namespace math and change the name␣
↪to m

In [7]: print(m.pi)
3.141592653589793
```

This is probably the most common and is more efficient (for the coder) if many functions are going to be used, it also eliminates the overwrite concern.

10.2 Importing from your own python scripts

You can also import functions and variables from scripts that you have already developed. Consider the following file tree:

```
project
├── functions
│   ├── __init__.py
│   ├── interpolation_techs
│   │   ├── __init__.py
│   │   └── geostatistical.py
│   └── lsr_calc.py
├── examples
│   ├── rainfall_interpolation.py
│   └── lsr.py
```

As long as the project folder is in your PYTHONPATH (more on this in a second) you can import objects from any script (.py) that is in any python module. A python module simply a folder that contains an *__init__.py* file. The file may be completely blank, or it can hold a set of imports for initialising the module. For our purposes we'll assume that the *__init__.py* file is blank, but you can find more information about what can be contained in inits here

Looking at the project tree above (remember the project folder is in your PYTHONPATH), you can import objects as follows:

```
# import a mythical function that converts potential evapotranspiration (et) to␣
↪actual et from lsr_calc.py
from functions.lsr_calc import pet_to_aet

# import a mythical function that does kriging interpolation from geostatistical.py
from functions.interpolation_techs.geostatistical import krig
```

Note that you cannot import anything from the examples folder as it does not have an *__init__.py* file.

10.3 Adding a folder to the PYTHONPATH

10.3.1 On Windows

1. Open Explorer.
2. Right-click *'Computer'* in the Navigation Tree Panel on the left.
3. Select *'Properties'* at the bottom of the Context Menu.
4. Select *'Advanced system settings'*
5. Click *'Environment Variables. . . '* in the Advanced Tab
6. **Under 'System Variables':**
    (a) If it does not exist add: **PYTHONPATH**
    (b) Append the path to your project separating paths with *;* as follows

```
C:\Users\Documents\project;C:\another-library
```

You will now be able to import objects from projects in every python script that you write on your computer.

10.3.2 Within python

You can also add a folder to your python path in a script before you import from that folder as follows:

```
folder_path = "C:/Users/Documents/project"  # path to the project folder
import sys  # a built in package which helps you access the pythonpath
sys.path.append(folder_path)  # adds the folder to the pythonpath
from functions.lsr_calc import pet_to_aet  # now you can import as usual
```

#this is coming # basically a discussion about mutability # introduce copy and deepcopy?

#this is coming

#this is coming

# list comprehension # also dictionary comprehension

#this is coming

#this is coming

CHAPTER 11

Practice Exercises

We have developed a set of practice exercises to give you a taste of doing your own scripting. These exercises are facillitated through Github Classroom, so we recommend that you wait until you have finished the lesson on version control before you begin to explore the exercises. If you want to get on to the exercises, this link will create a new repository with a copy of the exercises for you to begin working.

exercise	prerequisites / associated lessons
exercise 1	• *What is Python* • *Building Python on Your computer* • *Basic Python Objects, Variables, and Operators* • *The List* • *Dictionaries*
exercise 2	• *Conditional Statements* • *Loops in Python*
exercise 3	• *Functions in Python*

CHAPTER 12

Handy Builtin modules

There are a number of handy builtin modules in python. Rather than trying to cover them all in detail here, below is a a table of the most useful tools, what they do, and a link to some good learning materials:

Module with link	General description
time	Functions for manipulating clock time
math	Provides functions for specialized mathematical operations.
datetime	The datetime module includes functions and classes for doing date and time parsing, formatting, and arithmetic.
itertools	The itertools module includes a set of functions for working with sequence data sets.
glob	Use Unix shell rules to find filenames matching a pattern
os and os.path	portable access to operating system specific features and work on filenames and paths.
shutil	High-level file operations
pickle	Object serialization
timeit	Time the execution of small bits of Python code