Intro to GitHub ¶

Basic git commands:

- `git status`
- `git pull`
- `git add`
- `git commit`
- `git push`

Git Demonstration

Install and download Git (https://git-scm.com/downloads) and GitHub Desktop (https://desktop.github.com/).

1. How to `clone` a repository

You can use **Git Bash** to clone a repo or use GitHub Desktop. **I will demonstrate how to do it using Git Bash.** But all can also be easily performed using the interface GitHub Desktop. (Download and install GitHub Desktop (https://desktop.github.com/))

For example, let's clone the repository.

https://github.com/jiefu2017/eel3850S25 (https://github.com/jiefu2017/eel3850S25)

2. Getting the latest edits from a repository - use `git pull`

To `pull` from a repository, simply call `git pull` using Git Bash.

3. How to manage files within a repo

The 3 most used Git commands are: `git pull`, `git add`, `git commit` and `git push`. You can call these commands directly on the **Git Bash** console within the cloned repository on your machine.

This should be sufficient to get you started with Git and GitHub in this course. To learn more, watch the tutorials below:

- Git bootcamp: https://help.github.com/categories/bootcamp/ (https://help.github.com/categories/bootcamp/)
- Tutorials: https://www.atlassian.com/git/tutorials/ (https://www.atlassian.com/git/tutorials/)
- Interactive Introduction: https://try.github.io/ (https://try.github.io/)

The Curious git (https://matthew-brett.github.io/curious-git/curious_git.html) is also a great resource.

In [ ]:

Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This chapter outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this. Understanding this difference is fundamental to understanding much of the material throughout the rest of the book.

Users of Python are often drawn in by its ease of use, one piece of which is dynamic typing. While a statically typed language like C or Java requires each variable to be explicitly declared, a dynamically typed language like Python skips this specification. For example, in C you might specify a particular operation as follows:

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be written this way:

```
# Python code
result = 0
for i in range(100):
    result += i
```

Notice one main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred. This means, for example, that we can assign any kind of data to any variable:

```
# Python code
x = 4
x = "four"
```

Here we've switched the contents of `x` from an integer to a string. The same thing in C would lead (depending on compiler settings) to a compilation error or other unintended consequences:

```
/* C code */
int x = 4;
x = "four";  // FAILS
```

This sort of flexibility is one element that makes Python and other dynamically typed languages convenient and easy to use. Understanding *how* this works is an important piece of learning to analyze data efficiently and effectively with Python. But what this type flexibility also points to is the fact that Python variables are more than just their values; they also contain extra information about the *type* of the value. We'll explore this more in the sections that follow.

A Python List Is More Than Just a List

Let's consider now what happens when we use a Python data structure that holds many Python objects. The standard mutable multielement container in Python is the list. We can create a list of integers as follows:

In [2]:
```
?range
```

In [7]:
```
print(range(1, 11))
```

```
range(1, 11)
```

In [1]:
```
L = list(range(10))
L
```

Out[1]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [8]:
```
type(L[0])
```

Out[8]:
```
int
```

In [9]:
```
L[1]
```

Out[9]:
```
1
```

In [10]:
```
len(L)
```

Out[10]:
```
10
```

In [11]:
```
# find the last object in a list
L[-1]
```

Out[11]:
```
9
```

Or, similarly, a list of strings:

In [12]:
```
L2 = [str(c) for c in L]
L2
```

Out[12]:
```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

In [ ]:
```

```

In [13]:
```
type(L2[0])
```

Out[13]:
```
str
```

In [14]:
```python
L # 0,1,2,...9
L4 =[c*2 for c in L]
L4
```

Out[14]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Because of Python's dynamic typing, we can even create heterogeneous lists:

In [15]:
```python
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
```

Out[15]:
```
[bool, str, float, int]
```

Creating Arrays from Python Lists

We'll start with the standard NumPy import, under the alias `np` :

In [16]:
```python
import numpy as np
```

Now we can use `np.array` to create arrays from Python lists:

In [17]:
```python
# Integer array
np.array([1, 4, 2, 5, 3])
```

Out[17]:
```
array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy arrays can only contain data of the same type. If the types do not match, NumPy will upcast them according to its type promotion rules; here, integers are upcast to floating point:

In [18]:
```python
np.array([3.14, 4, 2, 3])
```

Out[18]:
```
array([3.14, 4.  , 2.  , 3.  ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

In [19]:
```python
np.array([1, 2, 3, 4], dtype=np.float32)
```

Out[19]:
```
array([1., 2., 3., 4.], dtype=float32)
```

Finally, unlike Python lists, which are always one-dimensional sequences, NumPy arrays can be multidimensional. Here's one way of initializing a multidimensional array using a list of lists:

In [20]:
```python
# Nested lists result in multidimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

Out[20]:
```
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

In [ ]:
```
# Create a length-10 integer array filled with 0s
```

In [ ]:
```
# Create a 3x5 floating-point array filled with 1s
```

In [ ]:
```
# Create a 3x5 array filled with 3.14
```

In [ ]:
```
# Create an array filled with a linear sequence
# starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range function)
```

In [ ]:
```
# Create an array of five values evenly spaced between 0 and 1
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in the following table. Note that when constructing an array, they can be specified using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)

Data type	Description
`int8`	Byte (–128 to 127)
`int16`	Integer (–32768 to 32767)
`int32`	Integer (–2147483648 to 2147483647)
`int64`	Integer (–9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`
`float16`	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`

More advanced type specification is possible, such as specifying big- or little-endian numbers; for more information, refer to the [NumPy documentation (http://numpy.org/)](http://numpy.org/). NumPy also supports compound data types, which will be covered in [Structured Data: NumPy's Structured Arrays (02.09-Structured-Data-NumPy.ipynb)](02.09-Structured-Data-NumPy.ipynb).

In [ ]:

```

```

Matplotlib

[Matplotlib (http://matplotlib.org/)](http://matplotlib.org/) is a plotting library. In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

You can read much more about the plot function [in the documentation (http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot)](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot).

Subplots

You can plot different things in the same figure using the `subplot` function. Here is an example:

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```

In [ ]:
```
```