
UM Carpentries Workshop - Python

2024-03-21

Pre-workshop survey:
https://carpentries.typeform.com/to/wi32rS?slug=2024-03-21-UMich-python

Workshop website:
https://umcarpentries.org/2024-03-21-UMich-python/

https://pad.carpentries.org/2024-03-21-umich

If you don't yet have the un-report.zip file, you can download it from here and extract it to your Desktop folder:
https://umcarpentries.org/intro-curriculum-python/setup.html

Python for Plotting

Launch jupyter lab, which is the software we will be using for teaching python. If you're here, you sent me your screenshot, and had already launched jupyter lab. Do the same thing.

Launch a new jupyter notebook (There are a few methods. Fred is showing the launcher.)
 Rename your notebook to give it a meaningful name, e.g. gdp_population.ipynb
 Extension ipynb = "i python notebok" and indicates that this file is a jupyter notebook

 Click the + sign in the top toolbar to insert additional cells

 We can do basic math in python

```
2+3
```

Click the run button (play/triangle symbol) to run code

In python 2^3 is not an exponent. The "carot" sign ^ in python means "exclusive OR" which we don't need to know about. Instead, in python, you use a double asterisk 2**3 for exponents.

Question: Does the space make any difference?
Answer: Great question! Here it doesn't matter; it's more for style. We get the same result. There are established styles, for example we add a single-whitespace on both sides of an equal sign. The program doesn't care [it's more for us to be able to read the code well].

In mathematics we also have orer of operations that we enforce with parantheses. To convert from fahrenheit to celcius:

- In [8]: 5 / 9 (24 - 32)
- Out[8]: -4.4444

We can save values to variables to use them latter. Create a variable age and assign a value to it.

- In [9]: age = 26
- Out[9]: [No output! We asked it to store the value]
- In [10]: print(age)
- Out[10]: 26

Another function that is useful is called type().

- In [11]: type(age)
- Out[11]: int

So the variable age has a type of integer. We can also have numbers with decimal places called floating point numbers:

- In [12]:
-   pi = 3.1415
-   print(pi)
- Out[12]:
-   3.1415

In the Jupyer Lab menu, go to Settings > Auto Close Brackets. This is very handy.

Question: Do I have to use print()?
Answer: You can just type pi to print out the value, but it only works in the Jupyter notebook [and not general python code]. It also only shows the value of the last variable and won't print other variables that we try to print:

- In [14]:
-   age
-   pi
- Out[14]:

- 3.1415

Let's use type() on pi

- In [15]:
- pi = 3.1415
- print(pi)
- print(type(pi))
- Out[15]:
- 3.1415
- <class 'float'>

Normally, code executes from top to bottom. But in jupyter you can technically execute cells in whatever order you want. Gives you flexibility, but can be chaotic! Good practices is to run top to bottom to not confuse other people. Everytime you run a cell, the number increases.

- In [19]: name = Ben
- Out[19]: ...
- NameError: name 'Ben' is not defined

Issue is we need to use quotation marks:

- In [20] name = "Ben"
- Out[20]: [Empty]

Doesn't matter if you use 'single quotation' or "double quotation" marks as long as you are consistent and close the opening with the same closing quotation mark.

- In [21]
- name = "Ben"
- **print(name)**
- **print(type(name))**
- Out[21]:
- Ben
- <class 'str'>

Let's assign another string:

- In [21]
- name = "Ben"
- print(name)
- print(type(name))
- **name = "Harry Potter"**
- **print(name)**
- Out[21]:
- Ben
- <class 'str'>
- **Harry Potter**

So the name gets reassigned.

Traditionally, we use shorter names for variables rather than longer names, but there are some restrictions

Question: What happened with the assignment?
Answer: It's overwriting the previous value of Ben with Harry Potter.

This will give an error:

```
• In [23]: 1number = 3
• Out[23]:
•   ...
•   SyntaxError: Invalid decimal literal
```

We can comment out the code with a pound sign #

```
• In [23]: # 1number = 3
• Out[23]: [No output]
```

Create another variable

```
• In [24]:
•   # 1number = 3
•   Flower = "marigold"
•   flower = "rose"
•   print(Flower)
•   print(flower)
• Out[24]:
•   marigold
•   rose
```

We can see the variable is not overwritten, because python is case-sensitive.

Let's create another variable called "favorite number"

```
• In [26]: favorite-number = 12
• Out[26]:
•   ...
•   SyntaxError: ...
```

Cannot use symbols like - & etc. Instead can use underscore _ to combine multiple words.

```
• In [29]: favorite_number = 12
• Out[29]: [No output]
```

Can store multiple values in the same variable. Lots of ways to do this with "Data structures". The first one we will talk about is a list. A list is indicated by a pair of square brackets and different items are separated by a comma.

```
• In [31]:
•   # data structures
•
•   # python list
•   squares = [1, 4, 9, 16, 25]
•   print(squares)
•   print(type(squares))
• Out[31:
•   [1, 4, 9, 16, 25]
•   <class 'list'>
```

We can store anything in a list; doesn't just have to be numbers.

```
• In [33]:
•   names = ['Sara', 'Tom', 'Jerry', 'Emma']
•   print(names)
•   print(type(names))
• Out[33]:
•   ['Sara', 'Tom', 'Jerry', 'Emma']
•   <class 'list'>
```

What if we wanted to get the first value from the list?

```
• In [34]: names[1]
• Out[34]: 'Tom'
```

What happened here? python uses zero-based indexing, so the first value is zero.

```
• In [34]: names[0]
• Out[34]: 'Sara'
```

To get the last name:

```
• In [34]: names[3]
• Out[34]: 'Emma'
```

python also supports negative indexing to count from the back:

```
• In [34]: names[-1]
• Out[34]: 'Emma'
```

For a very long list it's easier to count from the back.

So a list is a collection of items that are organized by position, e.g. first, second, third, ...
Another common data structure in python is accessing by key instead of position. That is called a python dictionary. Think about looking up the meaning of a word in a dictionary - hard to remember by page number, but easier to use the word name to find its meaning. Dictionary is indicated by a pair of curly brackets. Like a list, multiple items are connected with a comma:

```
• In [37]:
•   # python dictionary
•   capitals = {"France" : "Paris", "USA" : "Washington DC", "Germany" :
  "Berlin"}
•   print(capitals)
•   print(type(capitals))
• Out[37]:
•   {'France': 'Paris', 'USA': 'Washington DC', 'Germany': 'Berlin'}
•   <class 'dict'>
```

We can make these long lines easier to read with newlines that don't affect the code:

```
• In [40]:
•   # python dictionary
•   capitals = {"France" : "Paris",
•              "USA" : "Washington DC",
•              "Germany" : "Berlin"}
•   print(capitals)
•   print(type(capitals))
• Out[40]:
•   {'France': 'Paris', 'USA': 'Washington DC', 'Germany': 'Berlin'}
•   <class 'dict'>
```

Let's find one of the capitals:

```
• In [42]: capitals['France']
• Out[42]: 'Paris'
```

Calling functions:

```
• In [43]:
•   # calling functions
•
•   import os
•   os.getcwd()
• Out[43]: '/Users/pnanda/Desktop/un-report'
```

If we ignore the paranthesis we don't get the output we expect [looks similar to our type() output]:

```
• In [44]:
•   # calling functions
•
•   import os
•   os.getcwd
• Out[44]: <function posix.getcwd()>
```

Some functions in a module, are in a submodule:

```
• In [47]:
•   import datetime
•
•   datetime.date.today()
• Out[47]: datetime.date(2024, 3, 21)
```

Another function we will talk about is python's builtin function called round() for rounding numbers. A pro-tip is sometimes you may want to know what a function does or what argument it takes. If you type the name of a function, click on it, and use Shift + Tab you can get the documentation. So we can use this to see the round function takes 2 arguments, number and ndigits. Then there is a one-sentence summary of what the function does. You can also google the function to see what it does but this is quicker.

- In [48]: round(3.1415)
- Out[48]: 3

So far in jupyer lab we have clicked the run button. But in jupyter lab we can also use keyboard shortcuts. We can use Ctrl+Enter or Cmd+Enter to run a cell, which is quicker. Another keyboard shortcut I use everyday is to comment. You can temporarily disable multiple lines of code without deleting it. A quick way to do that is commenting it out. Keyboard shortcut is selecting lines then using Ctrl+/ or Cmd+/. For a single line you can move your mouse cursor without selecting and use the shortcut to comment it out.

The default value for ndigits in round is None. We can round to 3 digits:

- In [50]: round(3.1415, 3) # Quicker to write
- Out[50]: 3
- In [51]: round(**number=**3.1415, **ndigits=**3) # More clear
- Out[51]: 3

These were the basic things. Now we can more forward to making plots! Before making plots we need to first read in our data. We will read in gapminder_1997.csv In jupyter lab you can get a preview by double-clicking on a file like you would in a spreadsheet program. We see different countries of the world, popuilations, continent, life expectancy, and GDP per capita. For now we will be using this smaller dataset and later we will use gapminder_data.csv which has information across multiple years instead of just 1997 to see trends over time. For now, we will just look at data from 1997. For data analysis we will use extra libraries instead of basic python; here we will use the pandas library https://pandas.pydata.org

- In [52]:
-   import pandas as pd # Shorthand name to type 2 letters instead of 6.
-
-   # We can use the Tab key for autocompletion to choose pd.read_csv
-   pd.read_csv("./gapminder_1997.csv")
- Out[52]:
-   [Tabular preview of data]

--- Break ---

Python for Plotting (continued)

We can use the pandas function read_csv() to read out data into python. As expected, we see the same table that we saw in the jupyter preview. To analyze or plot the data, we need to assign the data table into a variable / object to refer to it later.

- In [56]:
-   import pandas as pd
-
-   **gapminder_1997 =** pd.read_csv("./gapminder_1997.csv")

- Out[56]: [No output]

- In [57]:
-   import pandas as pd
-
-   gapminder_1997 = pd.read_csv("./gapminder_1997.csv")
-   **gapminder_1997**
- Out[57]: [Table output]

- In [58]: type(gapminder_1997)
- Out[58]: pandas.core.frame.DataFrame

There are many libraries for plotting in python, unlike the R world where ggplot dominates. For this intro workshop we we focus on the seaborn plotting library. The library is similar to ggplot in R. We need to import the seaborn library. Note the color of "import" is green which tells you this is a python reserved keyword. We will use the idea of "method chaining" to create the plots using parentheses.

- In [59]:
-   import seaborn.objects as so
-
-   (
-       so.Plot(gapminder_1997)
-   )
- Out[58]: [Empty plot, because we haven't told it about axes or representation]

Question: why is seaborn so but pandas is pd? (i thought seaborn would be sb) :)
Answer: The "so" is shorthand for seaborn objects.

Make a scatterplot:

- In [60]:
-   import seaborn.objects as so
-
-   (
-       so.Plot(gapminder_1997**, x='gdpPercap', y='lifeExp'**)
-   )
- Out[60]: [Empty plot, but we have labels for X- and Y-axes]

Add the visual representation we want (dots, lines, bars, etc.):

- In [61]:
- import seaborn.objects as so
- (
-     so.Plot(gapminder_1997, x='gdpPercap', y='lifeExp')
-     **.add(so.Dot())**
-)
- Out[61]: [Scatterplot]

By default, the label of the axis is the column name from the dataset, but to publish we want to give a more detailed label:

```
• In [62]:
• import seaborn.objects as so
• (
•     so.Plot(gapminder_1997, x='gdpPercap', y='lifeExp')
•     .add(so.Dot())
•     .label(x='GDP per capita')
• )
• Out[62]: [Scatterplot with labels]
```

Exercise! Give a label to the y-axis for lifeExp.

```
• In [62]:
• import seaborn.objects as so
• (
•     so.Plot(gapminder_1997, x='gdpPercap', y='lifeExp')
•     .add(so.Dot())
•     .label(x='GDP per capita', y='Life expectancy')
• )
• Out[62]: [Scatterplot with both axes labelled]
```

Can also add a title:

```
• In [64]:
• import seaborn.objects as so
• (
•     so.Plot(gapminder_1997, x='gdpPercap', y='lifeExp')
•     .add(so.Dot())
•     .label(x='GDP per capita',
•            y='Life expectancy',
•            title='Do people in wealthy countries live longer?')
• )
• Out[64]: [Scatterplot with title]
```

Incorporate country distribution information into the plot with colors. Countries are categorical variables. Use Shift + Tab to see the arguments that the so.Plot() function can take.

```
• In [65]:
• import seaborn.objects as so
• (
•     so.Plot(gapminder_1997,
•             x='gdpPercap',
•             y='lifeExp',
•             color='continent')
•     .add(so.Dot())
•     .label(x='GDP per capita',
•           y='Life expectancy',
•           title='Do people in wealthy countries live longer?')
• )
• Out[65]: [Colored continent with legend]
```

You can control what colors you want to use with a color palette. The previous plot used the default palette. The default color palette looks like:

```
• In[68]:
• import seaborn as sns
• sns.color_palette()
• Out[68]: [Image with color blocks]
```

Try different palettes:

```
• In[69]:
• import seaborn as sns
• sns.color_palette('flare')
• sns.color_palette('Reds')
• sns.color_palette('Set1')  # Better for category data.
• Out[69]: [Image with color blocks]
```

We will use the Set1 color palette.

```
• In[72]:
• import seaborn.objects as so
• (
•    so.Plot(gapminder_1997,
•            x='gdpPercap',
•            y='lifeExp',
•            color='continent')
•    .add(so.Dot())
•    .label(x='GDP per capita',
•           y='Life expectancy',
•           title='Do people in wealthy countries live longer?')
•    .scale(color='Set1')
• )
• Out[72]: [Scatterplot with Set1 color palette]
```

[Demonstration of https://colorbrewer2.org color palettes and copying colors]

```
• In[73]:
• import seaborn.objects as so
• (
•    so.Plot(gapminder_1997,
•            x='gdpPercap',
•            y='lifeExp',
•            color='continent')
•    .add(so.Dot())
```

- .label(x='GDP per capita',
- y='Life expectancy',
- title='Do people in wealthy countries live longer?')
- .scale(color=**['#7fc97f','#beaed4','#fdc086']**)
-)
- Out[73]: [Color palette copied from JavaScript export of colorbrewer2 website]

What other information can we add to the plot? [Suggestion from student:] Population size.

- In[75]:
- import seaborn.objects as so
- (
- so.Plot(gapminder_1997,
- x='gdpPercap',
- y='lifeExp',
- color='continent',
- **pointsize='pop'**)
- .add(so.Dot())
- .label(x='GDP per capita',
- y='Life expectancy',
- title='Do people in wealthy countries live longer?')
- .scale(color=**'Set1'**)
-)
- Out[75]: [Now points are different sizes]

For finer control of point sizes, we can specify a minimum size because they're now harder to see.
The .scale() method helps us specify how data maps to different visual properties.

- In[76]:
- import seaborn.objects as so
- (
- so.Plot(gapminder_1997,
- x='gdpPercap',
- y='lifeExp',
- color='continent',
- pointsize='pop')
- .add(so.Dot())
- .label(x='GDP per capita',
- y='Life expectancy',
- title='Do people in wealthy countries live longer?')
- .scale(color='Set1',
- **pointsize=(2, 20)**)
-)
- Out[72]: [Now larger points]

<u>Question: pointsize in .scale() only seems to be changing legend size, but not the plot ifself.</u>
Answer: to change the size of the plot, you need to "chain" another method (called `layout`) that we didn't talk about. See the example code below:

- .layout(size=(8, 5)). # set the figure size to 8-by-5 inside the layout method

Reduce overplotting with transparency:

- In[77]:
- import seaborn.objects as so
- (
-     so.Plot(gapminder_1997,
-             x='gdpPercap',
-             y='lifeExp',
-             color='continent',
-             pointsize='pop')
-     .add(so.Dot(**alpha=0.5**))
-     .label(x='GDP per capita',
-            y='Life expectancy',
-            title='Do people in wealthy countries live longer?')
-     .scale(color='Set1',
-            pointsize=(2, 20))
-)

Can also add markers. Not necessarily a good idea, but good to see the possibilities:

- In[77]:
- import seaborn.objects as so
- (
-     so.Plot(gapminder_1997,
-             x='gdpPercap',
-             y='lifeExp',
-             color='continent',
-             pointsize='pop',
-             **marker='continent'**)
-     .add(so.Dot(alpha=0.5))
-     .label(x='GDP per capita',
-            y='Life expectancy',
-            title='Do people in wealthy countries live longer?')
-     .scale(color='Set1',
-            pointsize=(2, 20))
-)

Now we will use the full dataset:

- In[80]: gapminder = pd.read_csv("gapminder_data.csv")

As before, we have an empty plot without the plot layer:

- In[81]:
- (
-     so.Plot(gapminder, x='year', y='lifeExp', color='continent')

-)

The dot plot is hard to see the trend:

- In[82]:
- (
-    so.Plot(gapminder, x='year', y='lifeExp', color='continent')
-    **.add(so.Dot())**
-)

Let's use a line instead.

- In[83]:
- (
-    so.Plot(gapminder, x='year', y='lifeExp', color='continent')
-    .add(**so.Line()**)
-)

But still not what we want to see because it's grouping all countries. We want individual countries.

- In[84]:
- (
-    so.Plot(gapminder, x='year', y='lifeExp', color='continent', group='country')
-    .add(so.Line())
-)

Now one line per country.

<u>Question: What is this warning in red that appears:</u>

- /Users/pnanda/anaconda3/lib/python3.11/site-packages/seaborn/_core/plot.py:1491: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instead.
-   with pd.option_context("mode.use_inf_as_na", True):

Answer: You may see this warning in newer versions, but it's probably benign

- In[86]:
- (
-    so.Plot(gapminder_1997, x='continent', y='lifeExp')
-    .add(so.Dot())
-)

To deal with overplotting we can use jittering where we use noise to spread the dots a little bit.

- In[87]:
- (
-    so.Plot(gapminder_1997, x='continent', y='lifeExp')
-    .add(so.Dot()**, so.Jitter()**)
-)

Can use options to spread out the dots and make them bigger:

- In[90]:
- (
-     so.Plot(gapminder_1997, x='continent', y='lifeExp')
-     .add(so.Dot(**pointsize=10, alpha=0.5**), so.Jitter(**width=0.8**))
-)

We can map the continent information with the color:

- In[91]:
- (
-     so.Plot(gapminder_1997, x='continent', y='lifeExp'**, color='continent'**)
-     .add(so.Dot(pointsize=10, alpha=0.5), so.Jitter(width=0.8))
-)

<u>question: is it random how it spreads the jitter? his plot result is different from mine graphically : ok thanks</u>
Answer: Yes. His random seed is different than yours. The so.Jitter() has a seed= option that you can use for a consistent spread by specifying an integer number (I tend to use seed=123 for no particular reason).

Histogram plot with total number of bins:

- In[93]:
- (
-     so.Plot(gapminder_1997, x='lifeExp')
-     .add(**so.Bars(), so.Hist(bins=20)**)
-)

Set bins to a specific endpoint and with 5 year intervals:

- In[94]:
- (
-     so.Plot(gapminder_1997, x='lifeExp')
-     .add(so.Bars(), so.Hist(**binrange=(0, 120), binwidth=5**))
-)

<u>Question: why both Bars() and Hist()</u>
Answer: Bars() is the plot type, and Hist() is the summary to use. Can also use things like kernel density functions, etc. Seems tedious, but separates the representation from the statistical transformation.

Separate by continent, but this creates overplotting of bars on top of each other:

- In[95]:
- (
-     so.Plot(gapminder_1997, x='lifeExp'**, color='continent'**)
-     .add(so.Bars(), so.Hist(binrange=(0, 120), binwidth=5))
-)

Move the visual representation like we did with so.Jitter, but another way. We can use a stacked bar chart:

```
In[96]:
(
    so.Plot(gapminder_1997, x='lifeExp', color='continent')
    .add(so.Bars(), so.Hist(binrange=(0, 120), binwidth=5), so.Stack())
)
```

Similar to the histogram, we can create a kernel density estimate (KDE).

```
In[97]:
(
    so.Plot(gapminder_1997, x='lifeExp')
    .add(so.Line(), so.KDE())
)
```

For area we can instead use:

```
In[98]:
(
    so.Plot(gapminder_1997, x='lifeExp')
    .add(so.Area(), so.KDE())
)
```

For each continent:

```
In[99]:
(
    so.Plot(gapminder_1997, x='lifeExp', color='continent')
    .add(so.Area(), so.KDE())
)
```

We can add multiple layers to the same plot. But the KDE is too small to see because of the different Y-measure.

```
In[101]:
(
    so.Plot(gapminder_1997, x='lifeExp')
    .add(so.Bars(), so.Hist(binrange=(0, 120)), so.Stack())
    .add(so.Line(), so.KDE())
)
```

So we should use a different stat measure instead of count:

```
In[102]:
(
    so.Plot(gapminder_1997, x='lifeExp')
    .add(so.Bars(), so.Hist(stat='density', binrange=(0, 120)), so.Stack())
    .add(so.Line(), so.KDE())
• )
```

Can change properties and get this crazy plot:

```
• In[103]:
• (
•     so.Plot(gapminder_1997, x='lifeExp')
•     .add(so.Bars(color='green'), so.Hist(stat='density', binrange=(0, 120)),
  so.Stack())
•     .add(so.Line(color='red', linewidth=10), so.KDE())
• )
```

Last part I want to talk about is going back to our spaghetti plot from earlier.

```
• In[108]:
• (
•     so.Plot(gapminder,
•             x='year',
•             y='lifeExp',
•             color='continent',
•             group='country')
•     .add(so.Line())
•     .facet(col='continent', wrap=3)
• )
```

Can save with:

```
• In[109]:
• (
•     so.Plot(gapminder,
•             x='year',
•             y='lifeExp',
•             color='continent',
•             group='country')
•     .add(so.Line())
•     .facet(col='continent', wrap=3)
•     .save('awesome_plot.png')
• )
```

Fix legend being cut-off:

```
• In[109]:
• (
•     so.Plot(gapminder,
•             x='year',
•             y='lifeExp',
•             color='continent',
•             group='country')
•     .add(so.Line())
•     .facet(col='continent', wrap=3)
•     .save('awesome_plot.png', bbox_inches='tight')
• )
```

Also can save the image in higher resolution for publication:

```
• In[110]:
• (
•     so.Plot(gapminder,
•             x='year',
•             y='lifeExp',
•             color='continent',
•             group='country')
•     .add(so.Line())
•     .facet(col='continent', wrap=3)
•     .save('awesome_plot.png', bbox_inches='tight', dpi=200)
• )
```

5 minutes left! Last thing before lunch is another way to make visualizations. Python has lots of visualizations with their pros and cons. Seaborn is good for pretty plots. But they're static plots that we cannot interact with. Can make interactive plots to publish on your website or to make animations. We can use the plotly python library for that. Anaconda comes with plotly. Works differently from seaborn. See how the lifeExp changes from gdpPercap over the years and animate it.

```
• import plotly.express as px
• (
•     px.scatter(data_frame=gapminder,
•                x='gdpPercap',
•                y='lifeExp'
•               )
• )
```

Notice many more dots from the entire dataset.

```
• import plotly.express as px
• (
•     px.scatter(data_frame=gapminder,
•                x='gdpPercap',
•                y='lifeExp',
•                 animation_frame='year',
•               )
• )
```

Can drag the bar to see the distribution per year. Let's make it look nicer.

```
• import plotly.express as px
• (
•     px.scatter(data_frame=gapminder,
•                x='gdpPercap',
•                y='lifeExp',
•                animation_frame='year',
```

- **size='pop',**
- **color='continent',**
- **size_max=80,**
- **height=600,**
- **hover_name='country',**
-)
-)

Question: How do we resize the plot for later years so that we can see all the data?
Answer: On the top bar of plotly, click on "pan" and double-click on the plot. Or further right, click on "autoscale"

Question: How do we remove the blue background from the plot?
Answer: Use set_theme(). See https://seaborn.pydata.org/generated/seaborn.set_theme.html
Additional resources on setting theme:
https://seaborn.pydata.org/generated/seaborn.objects.Plot.theme.html

Question: can we save the animated plots as gifs like we can static as pngs?
Answer: Yes. You can even save the animation and controls with the HTML format. Something like the method `.to_html("file_name.html")`

UNIX Shell

PS1=$ # simplify your command prompt so you only see a dollar sign to enter commands.

pwd   # tells you the directory you are currently in. "print working directory"
ls   # shows you what is in your directory.
ks   # incorrect command "command not found"
man ls # does not work on Windows, so instead see
https://www.man7.org/linux/man-pages/

cd ~/Desktop
cd un-report/   # Can use the TAB key for autocomplete.
# What if we want to move backwards to the Desktop folder?
cd Desktop   # Does not work! Because it's not a folder in the current directory.
cd ..   # Goes back up a directory.
pwd

# How do we look inside a directory?
ls Desktop/un-report

- Question: Are commands sensitive to whitespaces?
- Answer: Yes, you need a whitespace between the command and arguments that follow the command.

cd Desktop/un-report
pwd

```
# Go to your home folder
cd ~
pwd

# Lets talk more about absolute and relative paths.
cd ./Desktop/un-report
# Where does this take us?
cd .     # It keeps us where we are because . means the current directory.
cd ..    # Takes us 1 back.
cd ../.. # Takes us 2 back.
pwd
cd ~     # Takes us back to our home directory.
cd ~/Desktop/..  # Effectively does nothing; goes to the desktop and back up 1 folder.

cd ./Desktop/un-report
ls      # Lots of files!  Not great data management.  Manage it better with folders.
mkdir code # make directory
ls code  # It's empty.
mv gdp_population.ipynb code/     # mv needs 2 things, what you're moving and where to.
# Alternatively could do
mv ~/Desktop/un-report/check_setup.ipynb ~/Desktop/un-report/code/
ls code/  # Shows our 2 files moved from our un-report/ folder.
mkdir data
ls
mv gapminder_1997.csv data/
mv gapminder_data.csv data/
ls data/
mkdir figures
ls *csv
ls d*
```

- Question: Can we use regular expressions in bash?
- Answer: Ways to do that but we will not cover that today.

```
# Besides moving things around, we can view files.
less data/gapminder_data.csv
# Can move down with the SPACE key.  To get out of the view, press q (for quit).
less -S data/gapminder_data.csv   # Keeps everything in a single line.

# What happens if we try to open a file that is not a text file?
less awesome_plot.png    # Complains about being a binary file.  Type y to see it.

# We can edit files with nano.
nano data/gapminder_data.csv  # Ignore changes with Control+X and n to not save.
```

- Question: How to open images?

- Answer: On your computer can use open / xdg-open. But on a cluster like Great Lakes not sure.

```
nano text.txt    # Type some text and save it.

# The next function "rm" deletes files and there is no way to recover it; does not go
to the trash directory and does not ask "are you sure?"  Be very careful with it!
rm text.txt

# Make a copy of the awesome_plot.png similar to mv.
cp awesome_plot.png ~/Desktop/un-report/figures/
ls          # Still shows awesome_plot.png here.
ls figures/  # Shows awesome_plot.png here as well.
```

- Question: List of useful bash commands?
- Answer: Not directly here, but resources out there like cheatsheets for common commands. UMich's advanced research computing (ARC) has cheatsheets.

--- Break ---

Git and GitHub

Slides link: https://umcarpentries.org/intro-git-cli/

- If you're working on a document such as your thesis, you go back and forth with your supervisor with lots of versions of the document. Or if you're working on revisions to a document with "track changes" you can see what was changed. Similarly, in a Google spreadsheet you can also see what changed and when.
- Some programs allow you to "Undo" changes. But if you close the document you can no longer access the changes you made before you closed the document. This is where version control comes in.
- If you're working on a project, in git terms, the "main" branch is where you put everything. You can work on separate features without affecting "main".

We assume you installed git during your setup and created a GitHub account. We will not talk about GitHub just yet. After newly installing git on your system, your collaborators should know who made the change. Launch the "Terminal" in Jupyer Lab and run:

```
# Configure git:
git config --global user.name "Your_First_Name Your_Last_Name"
git config --global user.email "your_email@umich.edu"
# Only if you're using Windows:
git config --global core.autocrlf true
# If you're using macOS or GNU/Linux:
```

```
git config --global core.autocrlf input
git config --global core.editor "nano -w"
git config --global init.defaultBranch main
git config --global credential.helper store
git config --list    # Check all our settings are correct.

# Use the shell commands we learned earler.
pwd
# Go to your un-report folder.
cd path/to/your/un-report/folder

# Check our git repository:
git status   # Error!  No git repository yet.
# Create an empty git repository:
git init
# Now we can use git with our current directory and its subdirectories:
git status
# The `git status` command output above shows "Untracked" files.
ls .git/     # Because `ls -a` does not work in Windows PowerShell.

# Add 2 files.
git add ./data/gapminder_1997.csv ./data/gapminder_data.csv
git status   # Note the new green text with new files staged to be committed.
git add ./data   # Add all the files in the data folder.
git status
# Good practice is to use a useful commit message.
git commit -m "Initial commit"
```

it's a commit comment! if our files were about comets it would be a comet commit comment lol

```
# In the jupyter lab interface click on the blue "+" button at the top-left, scroll to
the bottom, and choose "Markdown File".  Rename the file to README.md and add
some comments:
```

- ## Notes for UN Report
-
- We plotted life expetancy over time.

```
# Save your Markdown file and go back to the Terminal.

cat README.md    # Show what you saved in your Markdown file.
git status       # Out new README.md file is shown in red and is "Untracked".
git add README.md
git status       # Now the file is green and ready to be committed.
git commit -m "Start notes on analysis"
git status
# We committed our new README.md file.

# Add another line to the README.md file in your editor.
```

- ## Notes for UN Report
-
- We plotted life expetancy over time.
-
- Each point represents a country.

```
cat README.md
git status    # Shows README.md has been "modified".
git diff      # Shows what changed since our last commit.
# Previously, the add command both tracked and added the file to the stage.
# Now the add command only adds the file to the stage
# (because it's already tracked).
git add README.md
git commit -m "Add information on points"
git status

# Exercise: Add a another line to the README.md file as shown here:
# http://umcarpentries.org/intro-git-cli/#32

git log       # Show details of our commits.
git log -5    # Adding a number only shows those many recent commits.
git log --oneline   # Show only the short description in a single line.

# Now we will go to the GitHub website.
# We assume you have your GitHub account created.  Please sign-in to GitHub.
# Create your repository: http://umcarpentries.org/intro-git-cli/#36
# Copy the https URL of your new repository.
# Go back to your Terminal in Jupyer Lab.

git remote add origin https://github.com/YOUR_USER_NAME/un-report.git

# Now back on the GitHub website create a personal access token as described here: http://umcarpentries.org/intro-git-cli/#39

git push origin main
# For your password, paste your personal access token. It won't ask again for 30 days.

# Can look at commits similar to our `git log`: http://umcarpentries.org/intro-git-cli/#48
# Clicking on a commit shows what changed like `git diff`: http://umcarpentries.org/intro-git-cli/#49

# Let's create a file on GitHub itself: http://umcarpentries.org/intro-git-cli/#53
# Then copy that file back to our laptop:
git pull origin main

# Final Exercise: http://umcarpentries.org/intro-git-cli/#58
```

Python for Data Analysis

Questions

- How can I summarize my data in Python?
- How can Python help make my research more reproducible?
- How can I combine two datasets from different sources?
- How can data tidying facilitate answering analysis questions?

Objectives

- To become familiar with the common methods of the Python pandas library.
- To be able to use pandas to prepare data for analysis.
- To be able to combine two different data sources using joins.
- To be able to create plots and summary tables to answer analysis questions.

Some references we will consult later:
Pandas cheatsheet: https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf
Concept of tidy data: https://vita.had.co.nz/papers/tidy-data.pdf (ignore section 4 onwards)
Saving code-only jupyter notebooks with git + git pre-commit hooks: https://pypi.org/project/nbstripout/

Notes: it seems `! pwd` does not work for some people.

```
import pandas as pd
import numpy as np

gapminder = pd.read_csv("data/gapminder_data.csv")
```

Keyboard shortcut: In the viewing mode, press the "B" key to insert a new cell below.

You can also hide the output below a cell by clicking the left side blue bar next to the output.

```
gapminder.info()
```

Question: Is the object the same as string?
Answer: Not exactly. Object can be mixed type of strings and other types (e.g., integers).

```
gapminder.head(3)
```

```
gapminder.tail(3)
```

~~~
```
~~~

~~~
```
gapminder.tail()  # defaults to 5 rows
```
~~~

Question: can you do methed chaining with the info() and then head()?
Answer: Let's find out. Actually no because the output from the info() method is not a pandas dataframe but a "NoneType".

~~~
```
gapminder.describe()
```
~~~

Let's find out the mean value of the life expectancy:

~~~
```
(
    gapminder
    .agg({'lifeExp' : 'mean'})
)
```
~~~

Question: does it work if we use a capital letter for M when specifying the mean function?
Answer: No. Only the lowercase works. Python in general use lowercases, when there are multiple works, they are often connected via an underscore `_` (this naming style is called "snake_case"). You can check the pandas official website on what functions are available.

~~~
```
gapminder.max()['year']
```
~~~

Or

~~~
```
gapminder['year'].max()
```
~~~

~~~
```
# narrow down rows with query()

(
    gapminder
    .query('year == 2007')
)
```
~~~

~~~
```
(
    gapminder
    .query('year == 2007')
~~~

```
    .agg({'lifeExp' : 'mean'})
)
```

Question: can you round the mean lifeExp values?
Yes, you can chain the round() method at the bottom.

```
(
    gapminder
    .query('year == 2007')
    .agg({'lifeExp' : 'mean'})
    .round(decimals=2)
)
```

```
year_min = gapminder['year'].min()

(
    gapminder
    .query('year == @year_min')
    .agg({'gdpPercap' : 'mean'})
)
```

```
(
    gapminder
    .query('year == year.min()')
    .agg({'gdpPercap' : 'mean'})
)
```

```
(
    gapminder
    .query('year == year.min()')
    .agg({'gdpPercap' : ['mean', 'var']})
)
```

```
(
    gapminder
    .query('country == "United States"')
)
```

Note if the column name has spaces in it, we need to wrap the column name with backticks `` for it to work.

```
(
    gapminder
    .query('country == "United States" and year > 2000')
)
```

```
(
    gapminder
    .query('country == "United States"')
    .query('year > 2000')
)
```

The first way of query with two criteria is slightly more computationally effecient. You can put the Jupyter "magic command" below at the top row in a cell to time it.

```
%%timeit

(
    gapminder
    .query('country == "United States" and year > 2000')
)
```

It saved us a couple milli-seconds!! :)

```
(
    gapminder
    .query("country in ['United States', 'Canada']")
)
```

```
(
    gapminder
    .query("country in ['United States', 'Canada']")
)
```

```
country_list = ['United States', 'Canada']

(
    gapminder
    .query("country not in @country_list")
    .query("year == 2007")
    .query("continent == 'Americas'")
)
```

We can take a closer look at what happened underneath the covere after the groupby by checking the indices.

```
(
    gapminder
    .groupby('year')
    .indices
)
```

We can verify the indices output by checking

```
(
    gapminder
    .query('year == 1952')
)
```

```
(
    gapminder
    .groupby('year')
    .agg({'lifeExp' : 'mean'})
)
```

Exercise:
```
(
    gapminder
    .groupby('continent')
    .agg({'lifeExp' : ['mean', 'min']})
)
```

Question: why it doesn't work when have two agg methods one after another?
Answer: The data object is progressively reduced as the chain goes. The original data are lost after the first agg, so the second agg will not work.

```
```

```
(
    gapminder
    .query('year == 2007')
    .groupby('continent')
    .agg({'lifeExp' : 'mean'})
    .sort_values('lifeExp', ascending=False)

)
```

```
# make new variables with assign()

gapminder['gdpPercap'] * gapminder['pop']
```

```
(
    gapminder
    .assign(gdp=lambda df: df['gdpPercap'] * df['pop'])
)
```

Question: why we don't need to put quotes to the new column name in this case?
Answer: in this case the new column name is essentially the keyword argument inside the method call. And if you recall how to specify the arguments inside a function or method, we don't put quotes to them.

```
(
    gapminder
    .assign(gdp=lambda df: df['gdpPercap'] * df['pop'],
            popInMillion=lambda df: df['pop'] / 1_000_000)
)
```

Subset columns

```
gapminder['pop']
```

```
gapminder[['pop', 'year']]
```

```
(
    gapminder
    .filter(['pop', 'year'])
)
```

```

```
(
    gapminder
    .drop(columns=['pop', 'year'])
)
```

Changing the shape of data
```
(
    gapminder
    .filter(['continent', 'country', 'lifeExp', 'year'])
    .pivot(columns='year',
           values='lifeExp',
           index=['continent', 'country'])
)
```

```
(
    gapminder
    .filter(['continent', 'country', 'lifeExp', 'year'])
    .pivot(columns='year',
           values='lifeExp',
           index=['continent', 'country'])
    .reset_index()
    .melt(id_vars=['continent', 'country'],
          value_name='lifeExp')
)
```

```
# Exercise: Recreating the Americas 2007 gapminder dataset
# Choose to the year 2007
# Keep the continent 'Americas'
# Drop the year and continent
# Store the new dataframe into a variable gapminder_2007

gapminder_2007 = (
    gapminder
    .query("year == year.max() and continent == 'Americas'")
    .drop(columns=['year', 'continent'])

)
gapminder_2007
```
```

Commit the new notebook to git

```
git status
git add gapminder_data_analysis.ipynb
git status
git commit -m "Create data analysis file"
git log --oneline
git push
git push --set-upstream origin main
git status
```

You can use the keyboard shortcut Ctrl+C to get out of an ediit mode in a commit messege or other cases you are stuck in the edit mode (e.g., when using the `cat` command.)

```
pd.read_csv("data/co2-un-data.csv", skiprows=1)
```

Comment on git

Usually when you create a new git repo, you don't want it to be inside another git repo.

```
co2_emissions_dirty = (
    pd.read_csv("data/co2-un-data.csv", skiprows=2,
                names=['region', 'country', 'year', 'series',
                       'value', 'footnotes', 'source'
                      ])
)
co2_emissions_dirty
```

```
co2_emissions_dirty['series'].unique()
```

```
(
    co2_emissions_dirty
    .filter(['country', 'year', 'series', 'value'])
    .replace({
        'series' : {
            'Emissions (thousand metric tons of carbon dioxide)' : 'emissions_total',
            'Emissions per capita (metric tons of carbon dioxide)' : 'emissions_percap'
        }
    })
)
```

questoon: if we were only replacing one of the two values would we still need to use series or was that because we had two values?

```
co2_emissions = (
    co2_emissions_dirty
    .filter(['country', 'year', 'series', 'value'])
    .replace({
        'series' : {
            'Emissions (thousand metric tons of carbon dioxide)' : 'emissions_total',
            'Emissions per capita (metric tons of carbon dioxide)' : 'emissions_percap'
        }
    })
    .pivot(index=['country', 'year'],
           columns='series',
           values='value')
    .reset_index()
    # .value_counts(['year'])
    # .sort_index()
    .query('year == 2005')
    .drop(columns='year')
)
```

```
cco2_emissions = (
    co2_emissions_dirty
    .filter(['country', 'year', 'series', 'value'])
    .replace({
        'series' : {
            'Emissions (thousand metric tons of carbon dioxide)' : 'emissions_total',
            'Emissions per capita (metric tons of carbon dioxide)' : 'emissions_percap',
        }
    })
    .pivot(index=['country', 'year'], columns='series', values='value')
    .reset_index()
    # .value_counts(['year'])
    # .sort_index()
    .rename_axis(columns=None)
    .query('year == 2005')
    .drop(columns='year')
)

co2_emissions
```

# LUNCH

# Exercise: Recreating the Americas 2007 gapminder dataset
# Choose the year 2007
# Keep the continent 'Americas'
# Drop the year and continent

```
# Store the new DataFrame into a variable gapminder_2007.
gapminder_2007 = (
    gapminder
    .query('year == year.max() and continent == "Americas"')
    .drop(columns=['year', 'continent'])
)
gapminder_2007
```

```
(
    gapminder_2007
    .merge(co2_emissions, how='inner', on='country')
)
```

```
(
    gapminder_2007
    .merge(co2_emissions, how='outer', on='country', indicator=True)
    .query("_merge == 'left_only'")
)
```

```
(
    co2_emissions
    .query("country.str.contains('Bolivia|Puerto Rico|United States|Venezuela')")
)
```

```
(
    gapminder_2007
    .merge(co2_emissions, how='outer', on='country', indicator=True)
    .query("_merge == 'left_only'")
)

(
    co2_emissions
    .query("country.str.contains('Bolivia|Puerto Rico|United States|Venezuela')")
)

co2_fixed = (
    co2_emissions
    .replace({
        'country' : {
            'Bolivia (Plurin. State of)' : 'Bolivia',
            'United States of America' : 'United States',
            'Venezuela (Boliv. Rep. of)' : 'Venezuela'
        }
    })
)

(
    gapminder_2007
    .merge(
        co2_emissions
        .replace({
            'country' : {
                'Bolivia (Plurin. State of)' : 'Bolivia',
                'United States of America' : 'United States',
                'Venezuela (Boliv. Rep. of)' : 'Venezuela'
            }
        }),
        how='outer', on='country', indicator=True
    )
    .query('_merge == "left_only"')
)
```

```
# We want to combine PR and US

gapminder_fixed = (
    gapminder_2007
    .replace({'country' : {'Puerto Rico' : 'United States'}})
    .groupby('country')
    .apply(
        lambda df:
        pd.Series({
            'pop' : np.sum(df['pop']),
            'gdpPercap' : np.sum(df['gdpPercap'] * df['pop']) / np.sum(df['pop']),
            'lifeExp' : np.sum(df['lifeExp'] * df['pop']) / np.sum(df['pop'])
        })
    )
)
```

Question: What does the `apply' method do?
Use the specfied function and apply it to the entire dataframe.

comment: I had thought the NaN values were just o

```
gapminder_co2 = (
    gapminder_fixed
    .merge(co2_fixed, how='inner', on='country')
)
```

```
gapminder_co2
```

```
(
    gapminder_co2
    .assign(region=lambda df: np.where(
        # Condition:
        df['country'].isin(['Canada', 'United States', 'Mexico']),
        # If true,
        'north',
        # else
        'south'
    ))
    .to_csv("data/gapminder_xo2.csv")
)
```

```
(
    so.Plot(gapminder_co2,
            x='gdpPercap',
            y='emissions_percap',
            text='country',
           )
    .add(so.Dot())
    .add(so.Line(color='red'), so.PolyFit(order=1), text=None)
    .add(so.Text(valign='bottom', fontsize=10))
    .label(x="GDP (per capita)",
           y="GCO2 emitted (per capita)",
           title="Association between GDP and CO2",
          )
    .limit()
    .scale(x='log', y='log')
)
```

```
(
    pd.read_csv("data/gapminder_xo2.csv")
    .filter(['region', 'emissions_total', 'pop'])
    .groupby('region')
    .sum()
    .assign(emissions_perc=lambda df: df['emissions_total'] / np.sum(df['emissions_total']),
            pop_perc=lambda df: df['pop'] / df['pop'].sum(),
           )
)
```

```
gapminder_1997 = pd.read_csv("data/gapminder_1997.csv")

import plotly.express as px

(
    gapminder_1997
    .replace({
        'country' : {
            'United States' : 'United States of America',
            'United Kingdom' : 'United Kingdom of Great Britain and Northern Ireland',
        }
    })
    .merge(pd.read_csv("data/country-iso.csv")
           .rename(columns={'name' : 'country'}),
           on='country', how='inner'
          )
    .pipe(px.choropleth,
          locations='alpha-3',
          color='lifeExp',
          hover_name='country',
          hover_data=['lifeExp', 'pop']
         )
)
```

--- Break ---

Jupyter Notebooks and Markdown

Full curriculum:
https://umcarpentries.org/intro-curriculum-python/

Post-workshop survey:
https://carpentries.typeform.com/to/UgVdRQ?slug=2024-03-21-UMich-python