Python: A Crash Course in Programming

Slides: http://matthewja.com/misc/python

A brief introduction

Programming:

- I have a computer
- I want to make the computer do things for me
    - I need to use a programming language to tell it how to do the things

A brief introduction

Programming:

- I have a computer
- I want to make the computer do things for me
    - I need to use a programming language to tell it how to do the things

Python:

- General-purpose programming language
- *De facto* standard language used for machine learning, statistics, physics, biology, AI, ...
- Powers Reddit, Dropbox, BitTorrent, Blender, YouTube...

What we're going to do

We're going to learn how to do basic things in Python:

- Make decisions
- Do things repeatedly
- Use NumPy to work with data better
- Use MatPlotLib to make pretty pictures of data
- Where to go next

After this, you should have some background to help you out when you need to solve a programming task. You won't learn everything here, but you'll get some very useful exposure.

Outline

- Fundamentals
- Types, lists, strings, and methods
- Expressions, `print`
- `if`, `while`, and `for`
- Files and data
- NumPy
- MatPlotLib

Fundamentals

Installing Python

You should have Python 3.5 already installed. If you don't:

- Windows: [Download and run this installer.]
- Mac OS X: [Download and run this installer.]
- Linux: You probably already have Python!

Fundamentals

Installing libraries

Libraries are code that other people have written that give Python more features. If you're dealing with numbers, you're probably going to want **NumPy** and **MatPlotLib** (we'll look at these later), and if you're doing anything with scientific data or statistics you'll also want **SciPy** (which we won't look at).

Fundamentals

Installing libraries

NumPy:

- On Mac OS X, `sudo pip3 install numpy` should work.
- On Windows, download this, save it to your C: drive and rename it to `numpy.whl`, then open command prompt and run `C:\Python35\Scripts\pip.exe install C:\numpy.whl`.
- On Ubuntu or Debian, `sudo apt-get install python3-numpy` should work.

MatPlotLib:

- On Mac OS X, `sudo pip3 install matplotlib`.
- On Windows, `C:\Python35\Scripts\pip.exe install matplotlib`.
- On Ubuntu or Debian, `sudo apt-get install python3-matplotlib`.

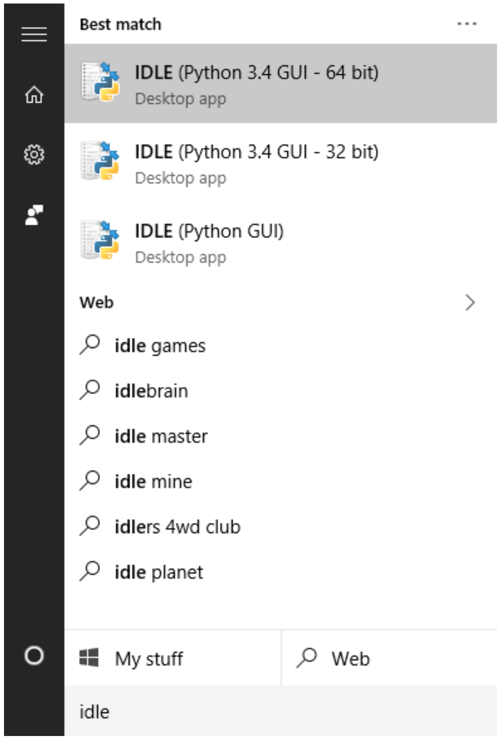
(This is a one-time setup thing...)

Dead link? Try this one.

Fundamentals

Running Python

1. Open IDLE3 or IDLE
2. You're running Python



Fundamentals

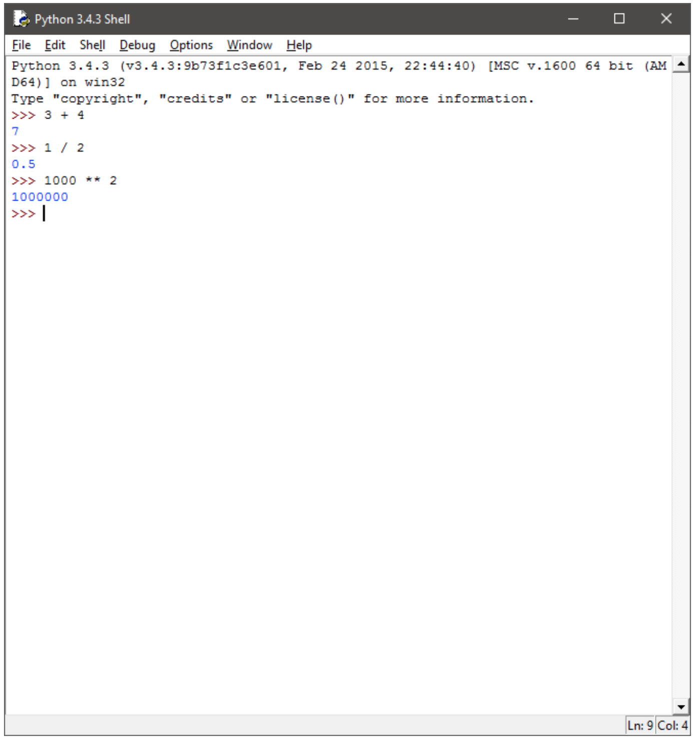
Getting around IDLE

Shell

This is the shell. You can use it to *interact* with Python code that you've already written.

How do you interact with it? ...With more Python code. **:D**

You can enter **expressions** into it and Python will evaluate them, then tell you the answer (in blue).



```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 3 + 4
7
>>> 1 / 2
0.5
>>> 1000 ** 2
1000000
>>> 
```

Fundamentals

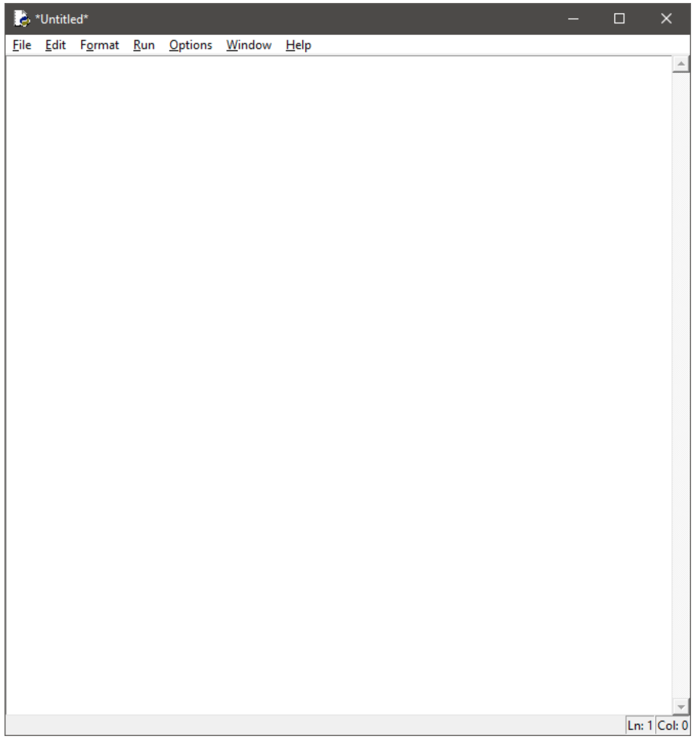
Getting around IDLE

File editor

In the shell window: File > New File

This is where you write code. Save files with a `.py` file extension (you'll have to type it in yourself) and you'll get **syntax highlighting**.

To run code, a) save it, b) Run > Run Module. If you're on Windows, you can also hit F5.



Comments

Lines that start with a # are ignored by Python. You can (and should) use this to describe how your code works to anyone reading your code.

```
# This is a comment
# This is another comment
# Python is pretty good

some_code_here()  # I can also put comments at the end of lines of code
```

Basics

Programming is a glorified way of manipulating **values**. Different kinds of values have different **types** and are represented in different ways.

```
1   # int
1.0   # float
"hello"   # str
True   # bool
```

Basics

You can combine values with **operators** to make new values.

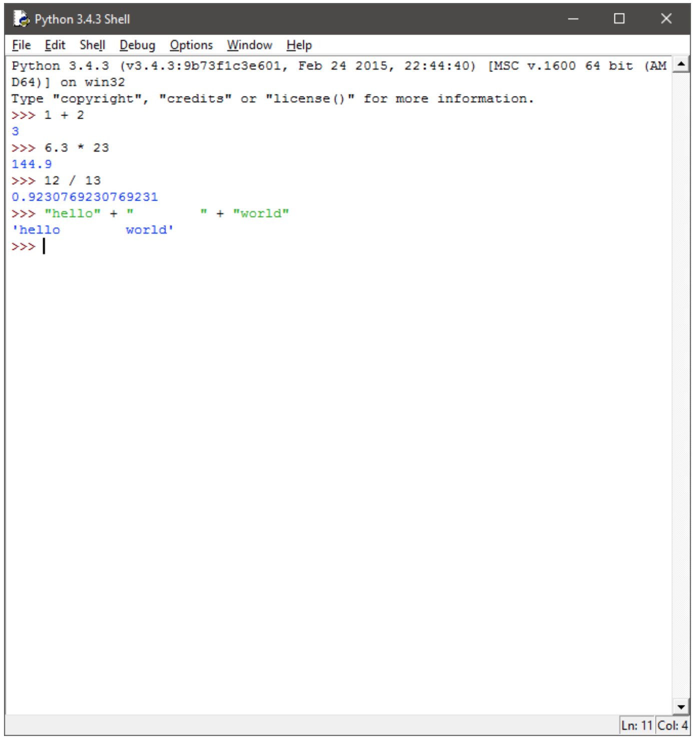
```
1 + 2  # is 3
1.0 + 2  # is 3.0
1 / 2  # is 0.5
"hello" + "world"  # is "helloworld"
(4 < 5) and (3 > 2)  # is True
(4 < 5) and (3 < 2)  # is False
```

Operators work on different types. `+`, `-`, `*`, and `/` work on numbers in ways you would expect; `and` and `or` work on `bool`s and let you do logic.

(also there's heaps more operators)

Basics

You can try this out in the shell.



```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 1 + 2
3
>>> 6.3 * 23
144.9
>>> 12 / 13
0.9230769230769231
>>> "hello" + "        " + "world"
'hello        world'
>>> 
```

Strings

Strings are sequences of characters.

```
"hello"
"world"
"this is me"
```

You can add them together, e.g. `"hello" + "world"` gives `"helloworld"`.

You can also slice them:

```
"this is me"[0]  # "t"
"this is me"[1]  # "h"
"this is me"[1:3]  # "hi"
```

If you just put a single number in the brackets, you will get a single character back. Note that `0` is the first character.

If you put two numbers separated by `:`, you will get multiple characters back. If you put in `[i:j]`, you will get back the `i`th character up to (but *not including*) the `j`th character.

Lists

Lists are sequences of values. They can have whatever you want in them and be as long as you like. They are a bunch of values surrounded by square brackets.

```
[1, 2, 3]
["hello", "world", "this is me"]
[1, 2, "hello", "world"]
[True, True, False]
[]
```

Just like strings, they can be sliced.

```
[1, 2, "hello", "world"][0]  # 1
[1, 2, "hello", "world"][1]  # 2
[1, 2, "hello", "world"][1:3]  # [2, "hello"]
```

Printing

If you try adding some numbers in Python's shell (or whatever), Python tells you what the result is. But if you type code into its own file, and then run that with Python, Python won't tell you anything. You have to explicitly tell Python to output information. You do this with `print`.

```
print("Hello world!")
print(1 + 2)
```

Variables

We want to store our values somewhere, so that we can reuse them (and also so that we don't have to jam all our maths onto one line). **Variables** are how we do this:

```
x = 1
result = x + 2
print(result)  # prints 3
```

Variables can be named any sequence of letters and underscores. Note that you can change their values later:

```
x = 1
# x is 1
x = 2
# x is now 2
```

You can even change their values based on their current values.

```
x = 1
x = x + 1
# x is now 2
```

Making decisions

You can compare values with **boolean operators**:

```
# here, x and y are variables
x == y  # equality
x != y  # inequality
x > y  # greater than
x < y  # less than
x >= y  # greater than or equal to
x <= y  # less than or equal to
```

Python turns these into `True` or `False`.

```
x = 1
y = 2

print(x != y)  # prints True
print(y < x)  # prints False
z = y > x  # z is now True
```

Making decisions

You can make Python do different things depending on whether something is `True` or `False` using an **if statement**.

```
x = True

if x:
  print("x is True!")
else:
  print("x is False!")
  # you can have lots of lines inside if/else
  # just keep indenting them
```

All the code that runs inside the `if` statement must be indented — this just means you should put a tab or two spaces in front of each line.

Making decisions

Inside the `if` statements you can put whatever code you want (including more `if` statements).

```
y = 78

x = 0

if y < 80:
  z = 10 * y
  if z > 770:
    x = z + 1
    # note that we've now indented *twice*
else:
  x = 4

print(x)  # what does this print?
```

Making decisions (repeatedly)

If you want to do something over and over again until something happens, you need a `while` loop. `while` loops are exactly like `if` statements, except that when Python gets to the end of the code inside them, it jumps back up to the start, checks whether the condition is `True` again, and then maybe runs the whole loop again (and repeat).

```
x = 0
while x < 10:
  x = x + 1
  print(x)
```

```
while True:
  # this runs forever!
  print("hello")
```

Just like with `if`s, you need to indent all the code that goes inside a `while` loop. You can put whatever code you want inside `while` loops (including more `while` loops or `if` statements... etc).

Fun computer science fact

You can now write Python programs that compute *literally any computable values*!

- variables
- `if`
- `while`

Technically all you need!

Fun computer science fact

You can now write Python programs that compute *literally any computable values*!

- variables
- `if`
- `while`

Technically all you need!

...though we should do more real-world stuff.

Converting between types

```
"10" + "10"  # "1010" D:
```

Use `float(string)` to convert `string` into a number. Use `str(number)` to convert a number into a string.

```
float("10") + float("10")  # 20 :D
```

Adding items to a list

If you have a list stored in a variable, you can add items to that list:

```
cool_list = [1, 2, 3]

cool_list.append(5)

# cool_list is now [1, 2, 3, 5]
```

This is particularly useful for storing results if you do lots of repeated calculations (like you might do in a loop!).

for loops

If you have a list of things, and you want to run some code for each thing in the list, you use a `for` loop.

```
my_cool_list = [1, 2, 3, 4, 5]

for item in my_cool_list:
  print(item * item)
```

In the above code, `item` is a variable. You can call it whatever you want.

Nested for loops

Maybe you have a list of lists.

```
z = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

And you want to do something to all of the elements in all of the lists.

```
for sublist in z:
  for item in sublist:
    print(item)
```

Nested for loops

Maybe you have a list of lists.

```
z = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

And you want to do something to all of the elements in all of the lists.

```
for sublist in z:
  for item in sublist:
    print(item)
```

Of course, when would we ever have a list of lists...?

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Working with data

To work with data, we could do things the hard way, or we could not do things the hard way. For the purposes of this workshop, we are not doing things the hard way.

NumPy is the Python community's answer to not doing things the hard way. It deals with data using types of values called **arrays**, which are like lists of lists (or lists of lists of lists, or...).

NumPy

Put this at the top of your code:

```
import numpy
```

This tells Python that we want to use NumPy. Let's load some data. Specifically, this data, via data.act.gov.au.

```
Year,Projected Population - Male,Projected Population - Female,Total Persons,Aged
2009,174999,177190,352189,18.4%,71.4%,10.2%,33.7,,
2010,178055,179903,357958,18.3%,71.3%,10.4%,33.8,5769,1.6%
```

Loading it is straightforward:

```
# 0, 3, and 7 are year, total persons, and median age, respectively
population_data = numpy.genfromtxt("population_data.csv", delimiter=",",
                                   names=True, usecols=[0, 3, 7])
```

This assumes that your code is in the same directory as the data file, and that the data file is a nice CSV (like you might get out of Excel).

NumPy

Since all our data is numeric, we can convert `population_data` into an array, which lets us do neat things like pull out specific columns and transform lots of data at once.

```
population_data = numpy.array(population_data.tolist())
```

This works because magic (outside the scope of this workshop).

NumPy

Can we use loops with it?

You bet. It's basically a list of lists — one list of rows, and then each row is a list itself.

```
for row in population_data:
  year = row[0]
  pop = row[1]
  age = row[2]

  print(year)
```

NumPy

How do we get a column by itself?

Remember slicing lists?

```
my_list[1:10]  # gets the 2nd item to the 11th item
               # (but not the 11th item)
```

NumPy arrays can also be sliced, but they can be sliced in multiple directions.

To get the (0, 2)th item (1st row, 3rd item — in our case, the first row, and the "median age" column):

```
population_data[0, 2]
```

If you want *all* items along a row or a column, use `:` instead of the number.

```
population_data[:, 2]  # all median ages!
```

NumPy

Can we make plots with it?

No.

NumPy

Can we make plots with it?

No.

We need MatPlotLib for that!

MatPlotLib

Tell Python that you want to use MatPlotLib:

```
import matplotlib.pyplot
```

(MatPlotLib is actually lots of libraries all put together, which is why we need the dot — this tells Python that we want the part of MatPlotLib that plots things.)

Then we can start to plot things. We need to have a list of all the `x` values and a list of all the `y` values. Say we want the `x` values to be the years (1st column of our data), and the `y` values to be the population (2nd column).

```
years = population_data[:, 0]  # all rows, 1st column
population = population_data[:, 1]  # all rows, 2nd column
```

MatPlotLib

To plot, you write `matplotlib.pyplot.plot(x_values, y_values)`. After you've plotted, you have to tell Python to show you the results. (Why doesn't it show automatically? Because sometimes you might want to plot lots of things before you show the results.)

```
matplotlib.pyplot.plot(years, population)
matplotlib.pyplot.show()
```

MatPlotLib

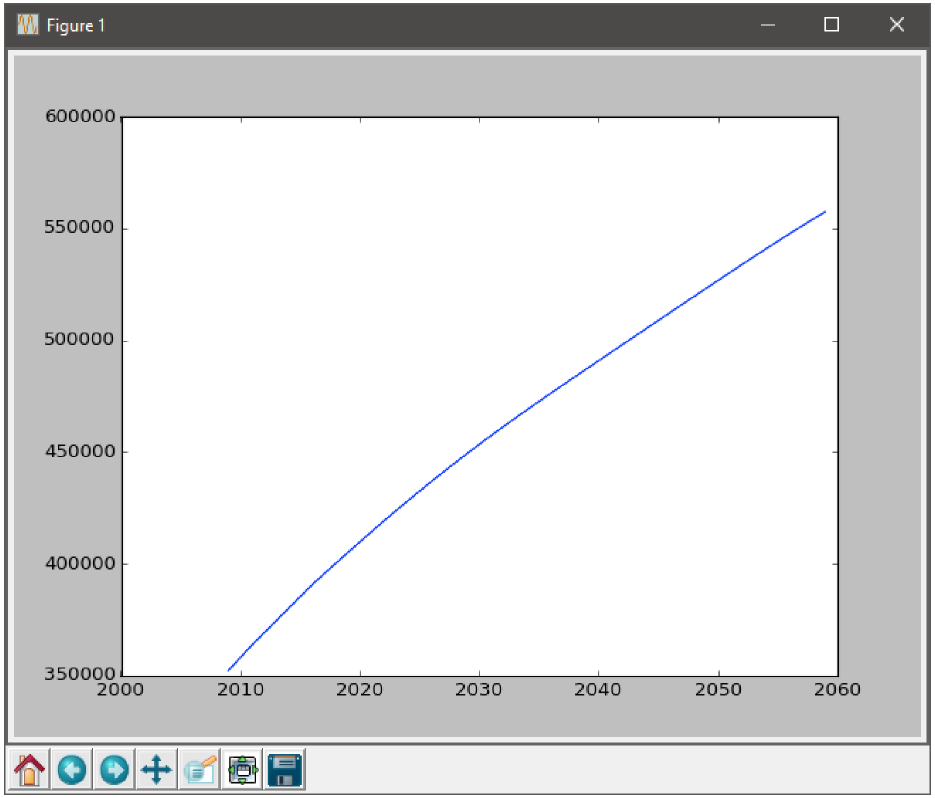


Figure 1
600000
550000
500000
450000
400000
350000
2000
2010
2020
2030
2040
2050
2060

MatPlotLib

I want axis labels! Titles!

Okay.

```
matplotlib.pyplot.plot(years, population)
matplotlib.pyplot.xlabel("Year")
matplotlib.pyplot.ylabel("Population")
matplotlib.pyplot.title("ACT Population Growth")
matplotlib.pyplot.show()
```

(Now you can see why we have to explicitly tell Python to `show`.)

MatPlotLib

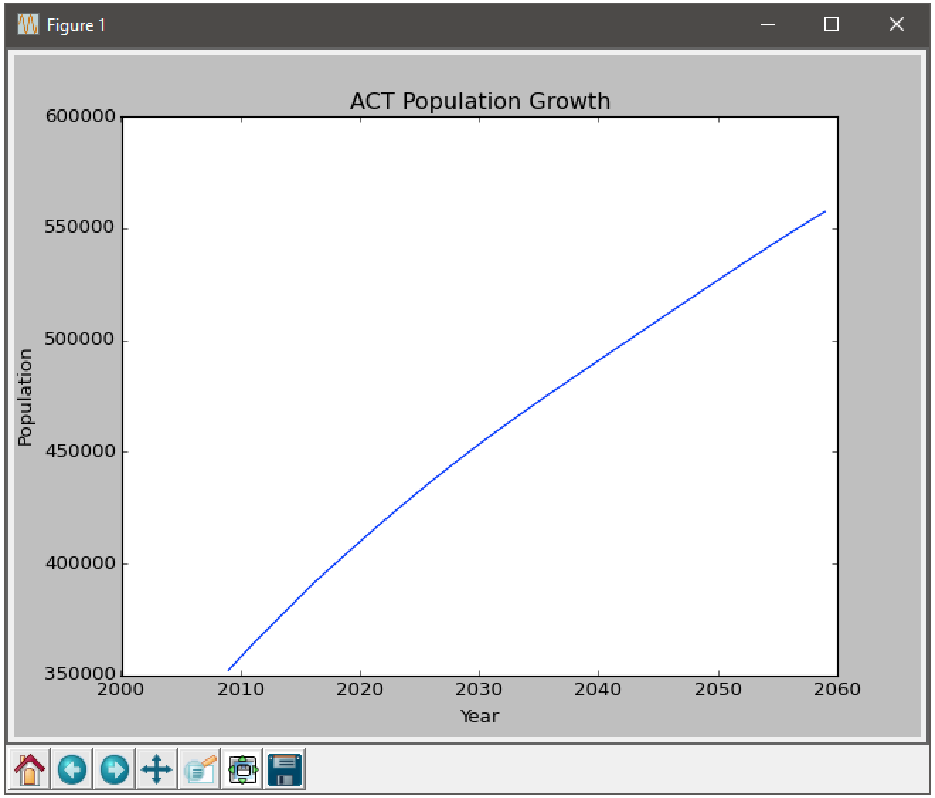


Figure 1
ACT Population Growth
Population
Year
600000
550000
500000
450000
400000
350000
2000
2010
2020
2030
2040
2050
2060

A simple model

I think that the population is totally a linear function.

$$population = 4057 \times year - 7796730$$

Let's see how close I am to being right. First, I want to plot this model against the real data. So we need to find all the populations predicted by this model.

A simple model

$$population = 4057 \times year - 7796730$$

```
years = population_data[:, 0]
populations = population_data[:, 1]

predicted_populations = []

for year in years:
  # figure out what the model population is
  model_population = 4057 * year - 7796730

  # add it onto the predictions list
  predicted_populations.append(model_population)
```

A simple model

$$population = 4057 \times year - 7796730$$

Plotting is then pretty straightforward. If you tell Python to `plot` two different things before `show`ing the plot, it will put them on the same graph.

```
years = population_data[:, 0]
populations = population_data[:, 1]

predicted_populations = []

for year in years:
  # figure out what the model population is
  model_population = 4057 * year - 7796730

  # add it onto the predictions list
  predicted_populations.append(model_population)

matplotlib.pyplot.plot(years, populations)
matplotlib.pyplot.plot(years, predicted_populations)
matplotlib.pyplot.xlabel("Year")
matplotlib.pyplot.ylabel("Population")
matplotlib.pyplot.title("Linear model of ACT population growth")
matplotlib.pyplot.legend(["data", "model"])
matplotlib.pyplot.show()
```

A simple model

$$population = 4057 \times year - 7796730$$

Now, let's find the residuals and plot those instead. To get the residuals, we have to subtract the model.

```
residuals = []
for row in population_data:
  year = row[0]
  population = row[1]

  # figure out what the model population is
  model_population = 4057 * year - 7796730

  # calculate and store the residual
  residual = population - model_population
  residuals.append(residual)
```

A simple model

$$population = 4057 \times year - 7796730$$

Then we can plot it.

```
residuals = []
for row in population_data:
  year = row[0]
  population = row[1]

  # figure out what the model population is
  model_population = 4057 * year - 7796730

  # calculate and store the residual
  residual = population - model_population
  residuals.append(residual)

years = population_data[:, 0]

matplotlib.pyplot.plot(years, residuals)
matplotlib.pyplot.xlabel("Year")
matplotlib.pyplot.ylabel("Residual")
matplotlib.pyplot.title("Residuals for linear model of ACT population growth")
matplotlib.pyplot.show()
```

A simple model

$$population = 4057 \times year - 7796730$$

If you don't want to recalculate all the residuals (we already stored them all in a list called `residuals` anyway), we can `zip` different lists of data together and loop through all of them in one hit.

```
years = population_data[:, 0]
populations = population_data[:, 1]
predicted_populations = [...]  # calculate as before...

residuals = []
for year, population, predicted in zip(years, populations, predicted_populations):
  # calculate and store the residual
  residual = population - predicted
  residuals.append(residual)
```

Summary

- `if`/`else` for doing different things based on conditions
- `while` to loop until a condition is false
- `for` to loop through data
- `import numpy` for data magic
- `import matplotlib.pyplot` for plotting

Next steps and other resources

- The fairly comprehensive official Python tutorial
- The kinda slow but more structured Codecademy Python course
- COMP1730
- Computer science academic mentors (most of us know Python)
- The documentation! Python docs, NumPy docs, MatPlotLib docs — if you have some exposure to Python (which you do), you can learn basically everything from examples in the docs
- PEP 0008, the Python style guide (for learning how to write good-looking Python code)