Python 03

Elias Wachmann

2024

Content

1. Variables
2. Python files
3. Functions
4. Fitting Data

Variables

Variable Names

From PEP-8 Style Guide:

- must start with letter or _
- can only contain alpha-numeric values (a-z, A-Z, 0-9) and _
- variable names are case sensitive
- do not overwrite built-in functions

Naming conventions

According to PEP-8 Style Guide:

- CONST_NAMES are all caps with _ between words
- function_names are all lower case with _ between words
- variables follow the same rules as functions
- ClassNames are CamelCase

Python files

Python files and usage

- Python files are called `.py`
- Python files can be run from the command line
  - `python3 file.py`
- Python files can be imported into other Python files
  - `import numpy`
  - or with an alias `import numpy as np`
- Python files can be run as scripts or imported as modules

Python files run as scripts

I can simply run this file and it will print the sinus values from y for the indices 5 through 10 (excluding 10).

```
# myscript.py
import numpy as np

x = np.arange(0, 10, 0.1)
y = np.sin(x)
print(y[5:10])
```

What happens if I import this file in another file?

Python files imported as modules

I can now import myscript.py in another file using the `import` statement:

```
import myscript
print("This is import.py")
```

What happens if I run this file?

Python files imported as modules

```
lides] % /bin/python3 /home/etschgi1/Desktop/REPOS/exercises-python/slide
s24/03/examples/import.py
[0.47942554 0.56464247 0.64421769 0.71735609 0.78332691]
This is import.py
```

It not only prints the *This is import.py* line, but also the sinus values from y for the indices 5 through 10 (excluding 10).
These are printed because the `import` statement runs the file as a script.
How to avoid this?

`__name__ == '__main__'`

By using the `__name__` variable, we can avoid running the file as a script when it is imported as a module.

```python
# myscript2.py
import numpy as np

if __name__ == "__main__":
    x = np.arange(0, 10, 0.1)
    y = np.sin(x)
    print(y[5:10])
```

Importing this file in another file will not print the sinus values.

Functions

Functions

- Functions are defined with `def`
- Functions can have arguments
- Functions can return values
- Functions can have default arguments
- Functions can have variable number of arguments

Some examples

```python
def hello():
    print("Hello, world!")


def arguments(a, b, c):
    print("Got arguments:", a, b, c)


def return_value():
    return 42
```

Some examples

```python
def add(a, b=2):
    return a + b


add_two_arguments = add(30, 12)
add_one_argument = add(40)
```

Multi returns

Multiple returns from the same funciton

```python
def multireturn(a, b):
    sum_ = a + b
    product = a * b
    return sum_, product


my_sum, my_product = multireturn(3, 4)
print(my_sum)  # 7
print(my_product)  # 12
```

Import functions from other files

```python
import random


def random_add(a, b):
    return a + b + random.random()
```

Import the functions from myfunc.py in another file (just like we did with numpy):

```python
from myfunc import random_add

print(random_add(1, 2))
```

Fitting Data

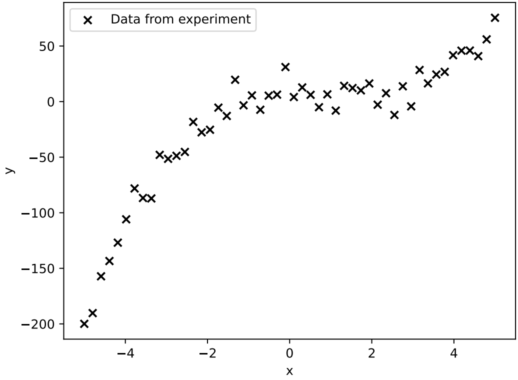
Fitting Data

In physics we often have to fit data to a function.
There are many ways to fit a function to given data.
For now we will use numpy's `polyfit()` function.

It fits a polynomial of degree `deg` to the data reducing the sum of squared residuals.
It returns the coefficients of the polynomial in decreasing powers.

Fitting Data - Experimental Data

From our measurements we get the following data:



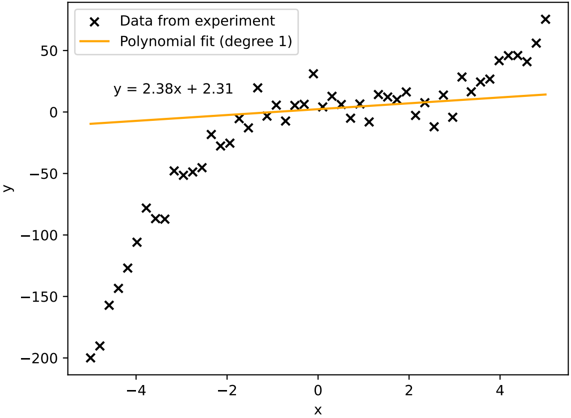
Fitting Data - Linear Fit

```
x_data = [...]
y_data = [...]
# plot data
plt.scatter(x_data, y_data, label="Data")
# Fit a line to the data (deg=1)
coefficients1 = np.polyfit(x_data, y_data,
    1)
# calculate the fitted values
y_fit1 = np.polyval(coefficients1, x_data)
plt.plot(x_data, y_fit1, label='Fit deg 1')
```

`polyval()` evaluates the polynomial at the given points.

Fitting - Linear Fit

A linear fit on the data works well for $x \in [-2, 3]$, but not outside this range.



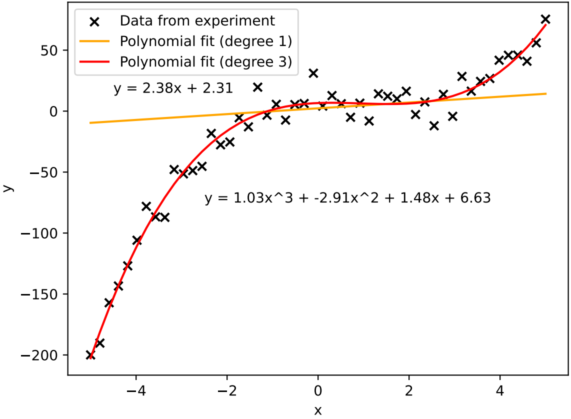
Fitting Data - Cubic Fit

We can now add another fit, this time a cubic fit:

```
...
# Fit a polynomial of degree 3 to the data
coefficients3 = np.polyfit(x_data, y_data,
    3)
y_fit3 = np.polyval(coefficients3, x_data)
plt.plot(x_data, y_fit3, label='Fit deg 3')
```

Fitting - Cubic Fit

A cubic fit describes the data better than a linear fit.



Fitting Data - Uncertainty

Every measurement has it’s own uncertainty.
We add uncertainty-bars to the data points using the `errorbar()` function:

```
plt.errorbar(x, y, yerr=y_err, xerr=x_err, fmt='o',
             color='blue', ecolor='red', capsize=5)
```

First two arguments are the data, `yerr` and `xerr` are the uncertainties. `color` specifies the color of the datapoints and `ecolor` the color of the errorbars.

Fitting Data - Uncertainty