Python 08

Elias Wachmann

2024

Content

1. Lambda functions
2. Curve fitting
3. Interpolation
4. Input / Output (I/O)

Lambda functions

Lambda functions

Lambda functions are small anonymous functions. They can take any number of arguments, but can only have one expression.
They are useful for short functions that are only used once.

```
x = np.linspace(0, 2*np.pi, 100)
my_lambda = lambda x: np.sin(x)
y = my_lambda(x)
```

Lambda functions inside other functions (closure)

Maybe you want to create functions at runtime. This can be done using lambda functions inside other functions.

```python
def create_gaussian(mu, sigma):
    return lambda x: np.exp(-(x-mu)**2/(2*sigma**2))
x = np.linspace(-5, 10, 1000)
my_lambda = create_gaussian(0, 1)
y = my_lambda(x)
```

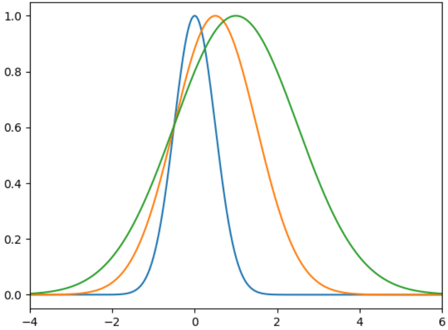
Lambda functions inside other functions (closure)

Create multiple functions using lists and a closure.

```python
mus, sigmas = [0, 0.5, 1], [0.5, 1, 1.5]
ys = [create_gaussian(mu, sigma)(x) for mu, sigma in zip(mus, sigmas)]
for y in ys:
    plt.plot(x, y)
plt.xlim(-4, 6)
plt.show()
```

Lambda functions inside other functions (closure)

Create multiple functions using lists and a closure.



Lambda using multiple arguments

Lambda functions can take multiple arguments, just like normal functions.

```python
adder = lambda x, y: x + y
print(adder(1, 2)) # 3
adder_default = lambda x, y=1: x + y
print(adder_default(1)) # 2
adder_lambda = lambda x: lambda y: x + y
print(adder_lambda(1)(2)) # 3
```

You can even specify default values for the arguments and nest lambda functions.

Lambda & if / else

Compact way to find prime numbers.

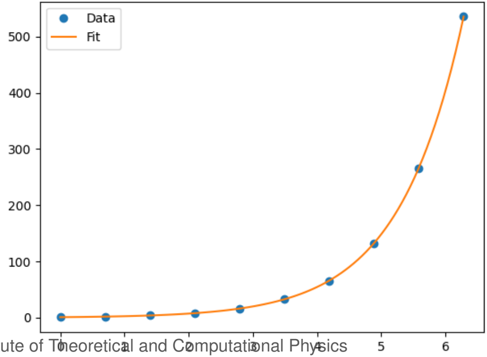
```python
import numpy as np
prime_test = lambda x,y: True if x % y else False
x = np.arange(1, 101)
primes = [i for i in x if all(prime_test(i,j) for j in range(2, i))]
print(primes[:5]) # [1, 2, 3, 5, 7]
```

Note: The `<arg> if <condition> else <arg>` syntax is called a conditional expression which can also be used inside list/dict comprehensions and in returns.

Curve fitting

Curve fitting using `scipy`

With the `curve_fit` function from the `scipy` module you can fit a function to data.



Curve fitting using scipy

```python
from scipy.optimize import curve_fit

x_samples = np.linspace(0, 2 * np.pi, 10)
sample_data = np.exp(x_samples)
sample_data += np.random.normal(0, 0.1, 10)

def fit(x, a, b): return a * np.exp(b * x)

popt, pcov = curve_fit(fit, x_samples, sample_data)
a_guess, b_guess = popt
```

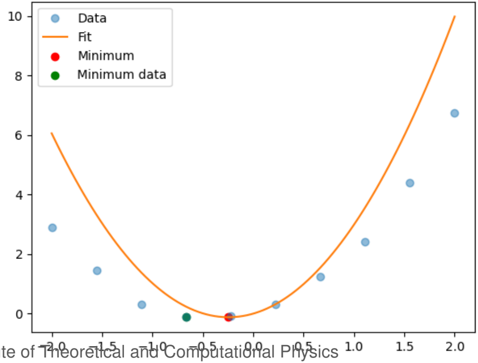
Curve fitting using `scipy`

```python
a_guess, b_guess = popt

x_smooth = np.linspace(0, 2 * np.pi, 100)
plt.plot(x_samples, sample_data, 'o', label='Data')
plt.plot(x_smooth, fit(x_smooth, a_guess, b_guess), label='Fit')
plt.legend()
plt.show()
```

Finding a minima using `scipy`

With the `fmin` function you can find the minima of a function.



Finding a minima using `scipy`

```python
import matplotlib.pyplot as plt
x_samples = np.linspace(-2, 2, 10)
sampled_data = np.random.uniform(-2, 2)*x_samples**2 + \
    x_samples + np.random.normal(0, 0.1, 10)

def sample_data(a, b, x): return a*x**2+b*x

popt, pcov = curve_fit(sample_data, x_samples, sampled_data)
a_guess, b_guess = popt
fmin_ = fmin(lambda x: sample_data(a_guess, b_guess, x), 0)
```

Finding a minima using `scipy`

```python
x_smooth = np.linspace(-2, 2, 100)
y_smooth = sample_data(a_guess, b_guess, x_smooth)
plt.plot(x_samples, sampled_data, 'o', label='Data', alpha=0.5)
plt.plot(x_smooth, y_smooth, label='Fit')
plt.scatter(fmin_, sample_data(a_guess, b_guess, fmin_),
            c='r', label='Minimum')
min_data = min(sampled_data)
```

Finding a minima using `scipy`

```
plt.scatter(x_samples[np.where(sampled_data == min_data)],
            min_data, c=’g’, label=’Minimum data’)

plt.legend()
plt.show()
```

Interpolation

Difference between interpolation and curve fitting

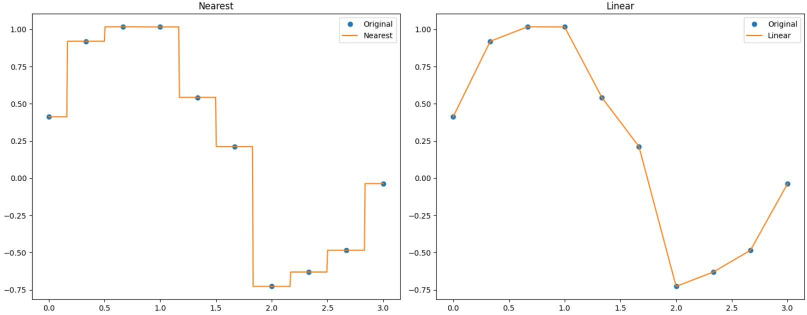
Interpolation constructs new data points within the range of a discrete set of known data points.

Curve fitting is the process of constructing a curve, or mathematical function, that has the best fit to a series of data points.

While interpolation requires the function to **go through the data points**, curve fitting does not.

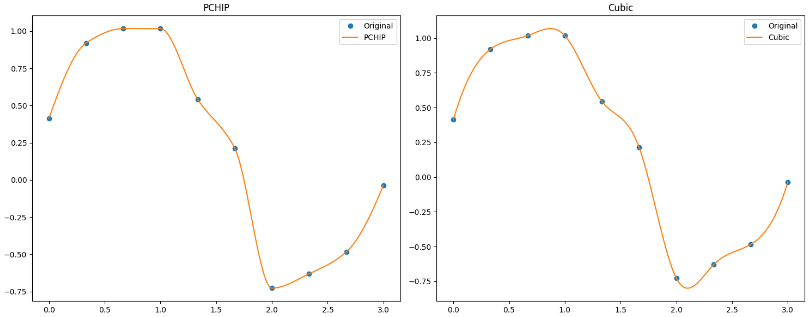
Interpolation using `scipy`

Scipy has a suite of functions for interpolation.



Interpolation using `scipy`

Scipy has a suite of functions for interpolation.



Interpolation using `scipy`

Using the `interp1d` function you can interpolate data.

- Nearest-neighbor interpolation ('nearest')
- Linear interpolation ('linear')
- Piecewise polynomial interpolation (PchipInterpolator)
- Cubic spline interpolation ('cubic')

For piecewise polynomial interpolation use `PchipInterpolator`.

Interpolation using scipy

```python
from scipy.interpolate import interp1d, PchipInterpolator

x_data = np.arange(0, 3.1, 1/3)
y_data = np.sin(2*x_data) + np.random.rand(len(x_data))/2

x_int = np.linspace(x_data[0], x_data[-1], 500)

interpolation_methods = ['nearest', 'linear', 'cubic']
```

Interpolation using scipy

```python
y_interpolated = {}
for method in interpolation_methods:
    f = interp1d(x_data, y_data, kind=method)
    y_interpolated[method] = f(x_int)

y_nearest = y_interpolated['nearest']
y_linear = y_interpolated['linear']
y_pchip = PchipInterpolator(x_data, y_data)(x_int)
y_spline = y_interpolated['cubic']
#... plotting
```

Input / Output (I/O)

Input / Output (I/O)

We have considered the `print` function to output text to the console.

Input can be read from the console using the `input` function.

Users can also input arguments when calling a script from the command line. These arguments are stored in the `sys.argv` list.

Input – sys.argv

Let’s assume we have a script with the following content:

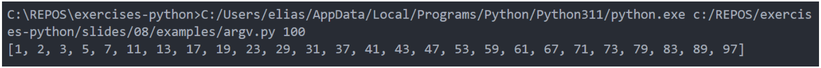
```python
import sys
def calcprimes(n):
    return [i for i in range(1, n+1) if all(i % j for j in range(2, i))]

if __name__ == ’__main__’:
    print(calcprimes(int(sys.argv[1])))
```

We can now call this script from the command line and pass the number of iterations as an argument.

Input – `sys.argv`

We can now call this script from the command line and pass the number of iterations as an argument.



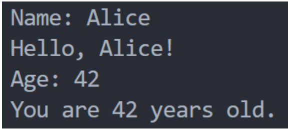
```
C:\REPOS\exercises-python>C:/Users/elias/AppData/Local/Programs/Python/Python311/python.exe c:/REPOS/exercis
es-python/slides/08/examples/argv.py 100
[1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Entering 100 as an argument will return and print all prime numbers up to 100.

Input – input

The input function can be used to read user input from the console.

```python
name = input("Name: ")
print("Hello, " + name + "!")
age = int(input("Age: "))
print("You are " + str(age) + " years old.")
```



```
Name: Alice
Hello, Alice!
Age: 42
You are 42 years old.
```

Input – check it before you wreck it!

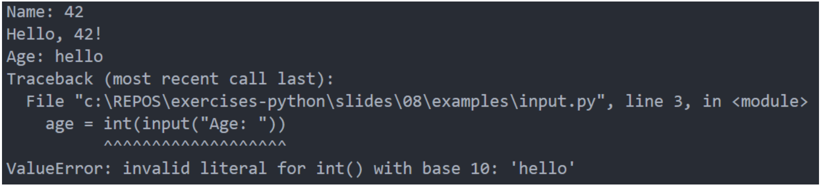
User input is always read as a string and can be used as it or converted to other types.

Never trust user input!

Always check if the input is valid and convert it to the desired type. Otherwise, it might lead to unexpected behaviour, error or even security issues in the worst case.

Input – check it before you wreck it!

Not checking input leads to unexpected behavior:



```
Name: 42
Hello, 42!
Age: hello
Traceback (most recent call last):
  File "c:\REPOS\exercises-python\slides\08\examples\input.py", line 3, in <module>
    age = int(input("Age: "))
          ^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'hello'
```

We can fix this by converting to the desired type in a try-except block.

Input – check it before you wreck it!

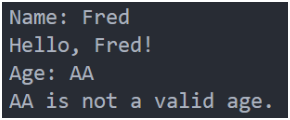
We can fix this by converting to the desired type in a try-except block.

```python
name = input("Name: ")
print("Hello, " + name + "!")
age = input("Age: ")
try:
    age = int(age)
except ValueError:
    print(f"{age} is not a valid age.")
    quit()
print("You are " + str(age) + " years old.")
```

Input – check it before you wreck it!

We can fix this by converting to the desired type in a try-except block.

This way the user gets useful feedback and the program does not crash.



```
Name: Fred
Hello, Fred!
Age: AA
AA is not a valid age.
```

Input – check it before you wreck it!

Some code may be malicious and try to exploit your program.

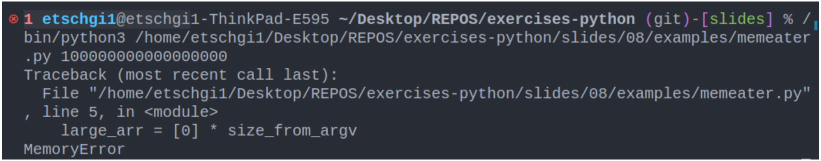
```
size_from_argv = int(sys.argv[1])
large_arr = [0] * size_from_argv

time.sleep(5)
```

If you input a large enough number the program will consume a lot of your RAM and the system may slow down or crash.

Input – check it before you wreck it!

Python has some check which prevent this from happening → `MemoryError`.



```
⊗ 1 etschgi1@etschgi1-ThinkPad-E595 ~/Desktop/REPOS/exercises-python (git)-[slides] % /
  bin/python3 /home/etschgi1/Desktop/REPOS/exercises-python/slides/08/examples/memeater
  .py 100000000000000000
  Traceback (most recent call last):
    File "/home/etschgi1/Desktop/REPOS/exercises-python/slides/08/examples/memeater.py"
  , line 5, in <module>
      large_arr = [0] * size_from_argv
  MemoryError
```

But it is still good practice to check user input and prevent this from happening.

Input – eval

The `eval` function can be used to evaluate a string as a Python expression.

```python
expr = sys.argv[1]
print(eval(expr))
print(eval("1 + 2 * 3"))
```

As you should know by now, this is dangerous and you shouldn’t evaluate user input!
Sometimes it may be useful though to dynamically evaluate equations (just like with closures).

Output – a closer look at `print`

The `print` function has a lot of options.

```python
print("Multiple", "arguments", "to", "print", sep=" | ", end="!\n")
print("Countdown: ")
for i in reversed(range(10)):
    print(i, end="\r")
    time.sleep(0.5)
```

The `sep` argument can be used to specify the separator between the arguments and the `end` argument can be used to specify the end of the line.
`\n` … newline, `\t` … tabulator, `\r` … carriage return.

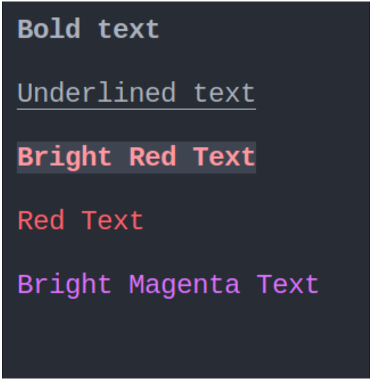
Output – a closer look at colors

How to print colored text to the console?
Use escape sequences!

```
print("\033[4mUnderlined text\033[0m\n")
print("\033[1;31;40mBright Red Text\033[0m\n")
print("\033[91mRed Text\033[0m\n")
print("\033[95mBright Magenta Text\033[0m\n")
print("\033^Invisible\033[0m\n")
```

Output – a closer look at colors

Output in the console:



```
Bold text

Underlined text

Bright Red Text

Red Text

Bright Magenta Text
```

Import csv-files

`csv` is a module that allows you to read and write csv-files.

`csv` is a text format, so you have to open the file in text mode (`w` or `r`).

To read a file use the `reader()` method, to write a file use the `writer()` method.

Import csv-files – Example (csv)

```python
import csv
import matplotlib.pyplot as plt
lists = []
header = []
with open(’slides/08/examples/example.csv’, ’r’) as f:
    reader = csv.reader(f)
    header = reader.__next__()[0].split(’;’)
    lists = [[] for _ in range(len(header))]
    for row in reader:
        for i, item in enumerate(row[0].split(’;’)):
            lists[i].append(int(item))
plt.plot(lists[0], lists[1])
plt.show()
```

Import csv-files – Example (csv)

Wow, that was a lot of code!
Isn't there a better way?

Maybe do everything in one line?

Use pandas inbuilt csv-import function instead:

```python
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('slides/08/examples/example.csv', sep=';')
plt.plot(df['x'], df['y'])
plt.show()
```