Python - Vorkurs

Caner Ates

University of Basel

Introduction to Scientific Computing

- Packages
- Numpy
- Scipy
- Matplotlib
- An Exercise

What's Python?

- General-purpose programming language
- Supports most programming paradigms
- Free and open source
- One of the most popular programming languages

Installation

- To install the "core Python package" you can go to `https://www.python.org/`
- As we want to use Python for scientific programming, you only have to install "Anaconda": `https://www.anaconda.com/`

→ Anaconda is a free distibution for Python which provides the core Python package and the most popular scientific libraries

Setting up your Environment

We will use the Spyder IDE which is already included with Anaconda

- Spyder is split into different "panes" which are sections providing us with information or access to certain features
- You can add or remove panes by going to "View" $\rightarrow$ "Panes"
- The panes can be moved ba dragging, they can be either docked or used in a different window
- You can change the fonts by going to "Tools" $\rightarrow$ "Preferences" $\rightarrow$ "General" $\rightarrow$ "Fonts"

Variables

- Variables store data in our programs
- Using the assignment operator "=", we give them names and values

→ all lower case, elements are seperated by an underscore

- A variable name in Python can only start with a letter

→ `number_1 = 15` / `my_name = "Caner"` / `num_list = [2,5]`

In Python you can assign multiple values to do different variables in one line. Instead of

```
number_1 = 15
my_name = "Caner"
num_list = [2, 5]
```

You can write the following code

```
number_1, my_name, num_list = 15, "Caner", [2,5]
```

This is called **multiple assignment**. Now run the code above and use the function `type()` with your freshly made variables as arguments in your console. What happens?

Numbers

- There are two different types of data representing numbers

→ **integers** and **floats**

- An integer is a whole number (no decimal point)
- A float is a number with decimal point (used for more precision)

Arithmetic Operators

	Operator	Example
Addition	+	`10 + 5 = 15`
Subtraction	−	`30 − 20 = 10`
Multiplication	*	`2 * 5 = 10`
Division	/	`6 / 2 = 3.0`
Modulus	%	`10 % 4 = 2`
Exponent	**	`2 ** 3 = 8`
Floor Division	//	`9 // 4 = 2`

Note: ^ is the bitwise operator "xor" (exclusive or)!

Strings

- A string is a series of characters
- In Python anything inside single or double quotes are strings

→ `"My name is..."`

→ `'Python is fun!'`

- We can also use quotes and apostrophes within our strings

→ `'He said, "I love my dog."'`

→ `"'Python' is dynamically typed."`

→ `"This is Caner's imaginary dog"`

Booleans

- The Boolean is a data type that has two possible values

→ `True` or `False`

- Booleans are often used to keep track of conditions
- You can write booleans as follows

→ `Bool = True`

- But usually we get them from doing logical comparisons

Comparison Operators

Operator	Description	Example
==	equal	`4 == 3` → `False`
!=	not equal	`4 != 3` → `True`
>	greater than	`6 > 10` → `False`
<	less than	`2 < 5` → `True`
>=	greater than or equal	`8 >=3` → `False`
<=	less than or equal	`5 <= 5` → `True`

Boolean Operations

Operation	Result
`x or y`	if `x` is false, then `y`, else `x`
`x and y`	if `x` is false, then `x`, else `y`
`not x`	if `x` is false, then `True`, else `False`

What do we need boolean operations for? Let's try out on Spyder!

If Statements

By using if statements, we can execute a piece of code only if a certain condition is `True`.

A simple if statement...

```
if condition :
    execute this code
```

- What if you have to test more than two possible situations?

→ If-elif-else chain

- The **`elif`** test is another **`if`** test, which runs if previous test has failed
- If the **`if`** and **`elif`** tests fail python runs the **`else`** block

```
if condition_1:
    code_1
elif condition_2:
    code_2
elif condition_3:
    code_3
else:
    code_4
```

Note: You can use as many **elif** tests as you like. Also, an **else** block is not required.

Lists

- A list is a sequence of elements in a particular order
- It is mutable
- The list elements are also called items

```
# create a list with integers
list_int = [1,4,5,8]
# access the third item in the list an print it
print(list_int[2])
```

- The code above returns the third item in the list

→ The index position in Python starts at 0 not 1

- You can modify an element of a list by accessing it and using the assignment operator "="

→ `list_int[2] = 7` #modified list: [1,4,7,8]

- The following table shows the most important list methods

Method	Description
`list.append(x)`	Add an Item to the end of the list
`list.insert(i,x)`	Insert an item at a given position
`list.pop(x)`	Remove item at given position and return it
`list.copy(x)`	Return a copy of the list
`list.sort()`	Sort the items

Examples	Outcome
`a = [1,2]; a.append(3)`	`# a = [1,2,3]`
`a = [1,2]; a.insert(1,3)`	`# a = [1,3,2]`
`a = [1,2,3]; popped = a.pop(1)`	`# a = [1,3]; popped = 2`
`a = [1,2]; b = a.copy()` `# a = [1,2]; b = [1,2]`	`# a = [1,2]; b = [1,2]`
`a = [4,1,5,3]; b = a.copy();` `a.sort(); b.sort(reverse = True)`	`# a = [1, 3, 4, 5]; b = [5, 4, 3, 1]`

For Loops

- Often, we want to perform the same task repeatedly or with each item in a list
- A **for** loop simplifies this procedure
- The **for** statement in Python iterates over items in any sequence in the order that they have in the sequence
- Iterating does not make a copy

→ If you want to modify the sequence in the loop, you should first make a copy

```
numbers = [4,34,2]
for number in numbers:
    print(number)
# 4
# 34
# 2
```

- The **`range`**`()` function generates arithmetic progressions
- It is commonly used to loop a specific number of time in **`for`** loops
- The **`len`**`()` function gives you the length of a list
- Combining **`len`**`()` and **`range`**`()`, you can over the indices o fa sequence

Note: The object returned by **`range`**`()` behaves like a list, but it doesn't make the list to save space.

Example

```
floats = [1.2, 2.343, 0.44]
for i in range(3):
    print(i)
# 0
# 1
# 2

for i in range(len(floats)): # len(floats) = 3
    print(i, floats[i])
# 0 1.2
# 1 2.343
# 2 0.44

# you can also loop within a list
list_loop = [2*i for i in range(5)]
```

While Loops

A while loop executes a task repeatedly *while* an expression is true.

```
i = 1
while i < 3:
    print(i)
    i += 1 # equivalent to i = i + 1
```

```
i = 1
while i < 10:
    print(i)
    if i == 2:
        break # stops the loop
    i += 1 # equivalent to i = i + 1
```

Functions

- A function is a block of code that is named
- It is written to do a specific task
- To perform the task, call the function name

→ Python runs the code inside the function

- By using the keyword **`def`** we tell python that we are defining a function
- It is followed by the function name and a list of parameters in parentheses

Fibonacci Examples

```
# Examples are taken from the python docs:
# https://docs.python.org/3/tutorial/controlflow.html

def fib(n):
    """Print a Fibonacci series up to n"""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()
```

If we want the function to return a list of the Fibonacci series...

```
def fib2(n):  # return Fibonacci series up to n
    """
    Return a list containing
    the Fibonacci series up to n.
    """

    result = [] # create an empty list
    a, b = 0, 1
    while a < n:
        result.append(a) # fill your list
        a, b = b, a+b
    return result
```

- Functions save time as we don't have to repeat code for specific tasks
- They also make our code more readable and organized
- We can save function in modules (modules are python files; for example: your_module.py) and import those modules in order to use our functions

→ Make sure that the module is in the current working directory

→ Then, you can import your module by using an **import** statement, usually declared on top of the code and under general comments

→ `import your_module`

Packages

- Packages are imported by using an **import** statement
- Generally packages are directories that contain...
- ... Python files (modules)
- ... a **__init__**.py file specifiying what will be executed when running "**import** some_package"
- Subpackages are packages hierarchically below a package and can be imported as follows

→ **import** package.sub_package

- If the module/package name after the **import** statement is followed by **as**, the name after **as** is bound to the imported module/package
- You can access names directly by using the **from** clause

Examples

```
import numpy.random # only import
                    # the random subpackage

numpy.random.seed(123) # set seed to have same numbers
numpy.random.randn(2)
# array([-1.0856306 ,  0.99734545])
```

```
import numpy as np # name the numpy as np
from numpy import cos, pi

np.sin(np.pi) # 1.2246467991473532e-16
np.round(np.sin(np.pi)) # 0.0
cos(pi) # -1.0  you can save code
        #       by using the from clause
```

Numpy

- Basic package for scientific computing
- Provides N-dimensional arrays (ndarrays) and useful mathematical operations
- ndarrays are flexible, efficient and also faster than lists
- Their size can be dynamically modified

- A ndarray object is a collection of elements that have the same type.
- The data type of the elements are called `dtype`
- The shape is a tuple of the length N
- Each element of the tuple defines number of elements in corresponding dimension
- You can create arrays by using `numpy.array()` with lists, tuples or arrays as an argument
- `numpy.zero()` returns a zero-array of given shape

Examples

```
import numpy as np

N_dim = (2,3)    # two dimensions
                 # first dimension 2 elements
                 # second dimension 3 elements
# create a 2x3 zero array resp. 'matrix'
zeros_23 = np.zeros(N_dim); print(zeros_23)
#[[0. 0. 0.]
# [0. 0. 0.]]
```

```
# create an one dimensional array
array_1d = np.array([2,51,6]); print(array_1d)
# [ 2 51  6]
# create a two dimensional array
array_2d = np.array([[2,51,6],[4,3,9]])
print(array_2d)
#[[ 2 51  6]
# [ 4  3  9]]
```

ARRAY INDEXING AND SLICING

- Square brackets `[]` are used to index array values

→ `array_1d[0]` #first element: 2

→ `array_2d[1, 0]` #first element of second row: 4

- Array slicing for producing subarrays; Also enables us to get entire rows or columns

→ Basic syntax for slicing: `start:stop:step`

Examples

```
# 1 dimensional slicing example
a1d = np.linspace(1,11,10)  # create a sequence from
                            # from 1 to 11 in 11 steps
sliced_a1d = a1d[2:10:2]
# take every second element starting from the third
# element stop before the eleventh element
sliced_a1d # array([3., 5., 7., 9.])
```

```
# create a 2 dimensional 6x3 array
zeros_6x3 = np.zeros((6,3))
# us numpy.copy() if you don't want to zeros_6x3
array_6x3 = np.copy(zeros_6x3)
# create an array with 18 values
# use random module or linspace
# array_vals = np.random.randn(zeros_6x3.size)
array_vals = np.linspace(1,100,zeros_6x3.size)
n = array_6x3.shape[0] # elements first dim
m = array_6x3.shape[1] # elements second dim
for i in range(n): # loop through first dim
    for j in range(m): # loop through second dim
        vals_index = m*i + j # index array_vals
        # print(vals_index) <- test it!
        #fill in array_6x3 with array_vals
        array_6x3[i,j] = array_vals[vals_index]
print(array_6x3)
```

```
# another way to create a 2 dim 6x3 array:
array_6x3 = np.linspace(1,100,6*3);
array_6x3.shape = (6,3)

# in one line:
array_6x3 = np.linspace(1,100,6*3).reshape(6,3)
#[[  1.          6.82352941  12.64705882]
# [ 18.47058824  24.29411765  30.11764706]
# [ 35.94117647  41.76470588  47.58823529]
# [ 53.41176471  59.23529412  65.05882353]
# [ 70.88235294  76.70588235  82.52941176]
# [ 88.35294118  94.17647059 100.        ]]
```

Another Slicing Example

```
# fetch the second row
array_6x3[1,:]
# array([12.64705882, 30.11764706, 47.58823529])

# fetch the first 3 elements of the third column
array_6x3[0:3,2]
# array([12.64705882, 30.11764706, 47.58823529])
```

Array Manipulation

Funcitons	Description
`numpy.append()`	Add an element to the end of the array
`numpy.insert()`	Insert an element(s) at a given position(s)
`numpy.delete()`	Remove elements at given position and return it
`numpy.resize()`	Returns new array with specified shape
`numpy.reshape()`	Gives new shape to an array
`numpy.copy()`	Return a copy of the array
`numpy.sort()`	Sort the array elements
`numpy.tranpose()`	Permute dimensions of array
`numpy.dot()`	Returns dot product of two arrays
`numpy.linspace()`	Return evenly spaced numbers over an interval
`numpy.arange()`	Similiar to `linspace`; steps are different

Let's try out some!

```
# create 2 dimensional array
# elements from 0 to 17
array_2d = np.arange(18).reshape(6,3)
# matrix mulitplikation
array_2d_T = np.transpose(array_2d)
np.dot(array_2d_T, array_6x3)
# or
mat_mult = array_2d_T @ array_6x3
```

```
# copy array
mmult_copy = np.copy(mat_mult)
# reverse multidim array elements
# along the second axis
# first make it 1 dimensional
# then reverse it using the slicing syntax
m1d = mmult_copy.flatten()[::-1]
mmult_copy_rev = m1d.reshape(3,3)
# or
np.rot90(np.rot90(mmult_copy))
for i in range(2):
    mmult_copy = np.rot90(mmult_copy)
mmult_copy
```

SciPy

The SciPy library "...provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization." - SciPy Docs

- SciPy has subpackages that are very useful in scientific computing
- such as `scipy.integrate`, `scipy.optimize`, `scipy.stats`, `scipy.interpolate`, `scipy.linalg`

EXAMPLE scipy.linalg

```
import numpy as np
from scipy import linalg
# you can compute the determinant
# or the inverse of a matrix
array_3x3 = np.random.randint(1,100,3*3).reshape(3,3)
linalg.det(array_3x3)
linalg.inv(array_3x3)
```

```
# or solve linear equations
# 200x + 6y = 8
#  14x + 0.5z = 3
#  4y + 300z = 190
M = np.array([[200.0, 6.0, 0.0],
              [14.0, 0.0, 0.5],
              [0.0, 4.0, 300.0]])
B = np.array([[8],[3],[190]])
x = linalg.solve(M,B)
print(200.0*x[0] + 6*x[1])
```

Matplotlib

- Most popular 2D plotting package in Python
- seperated in three layers

→ backend (bottom), artist (middle), scripting (top)

- We are going to look at the scripting layer (pyplot) because...

"For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users." - matplotlib.org

USEFUL PYPLOT FUNCTIONS

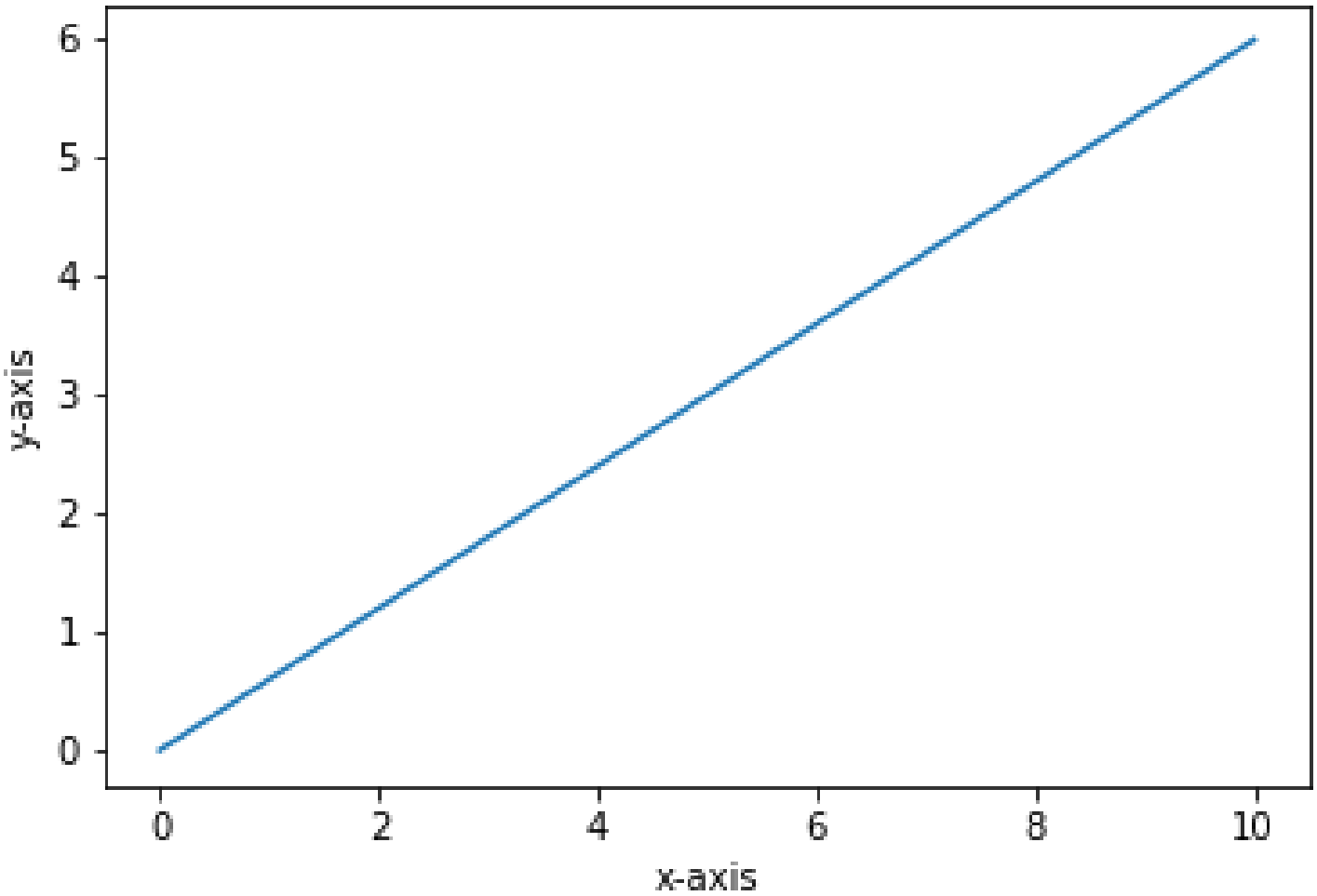
```
import matplotlib.pyplot as plt
```

Functions	Description (matplotlib.org)
`plt.plot()`	Plot y versus x as lines and/or markers
`plt.ylabel()`	Set the label for the y-axis
`plt.xlabel()`	Set the label for the x-axis
`plt.axis()`	Method to get or set some axis properties
`plt.title()`	Set a title for the axes
`plt.scatter()`	A scatter plot of y vs x
`plt.bar()`	Make a bar plot
`plt.figure()`	Create a new figure
`plt.suptitle()`	Add a centered title to the figure
`plt.subplot()`	Add a subplot to the current figure
`plt.show()`	Display a figure

- The easiest way to plot is by using the `plot()` function

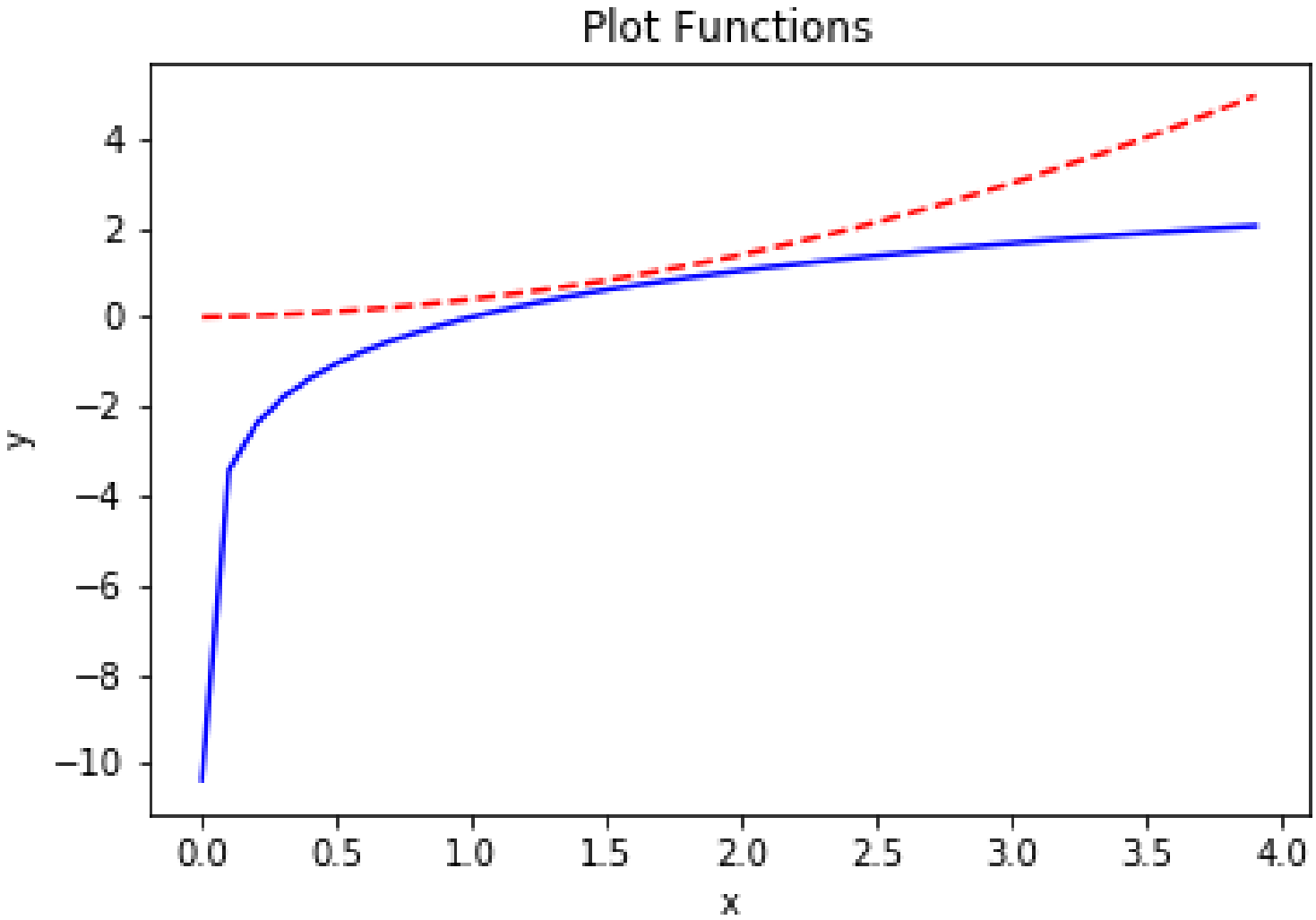
→
```
x_vals = np.linspace(0,10,10)
y_vals = np.linspace(0,6,10)
```

→
```
plt.plot(x_vals, y_vals)
plt.ylabel("y-axis"); plt.xlabel("x-axis")
plt.show()
```


y-axis
x-axis
0
1
2
3
4
5
6
0
2
4
6
8
10

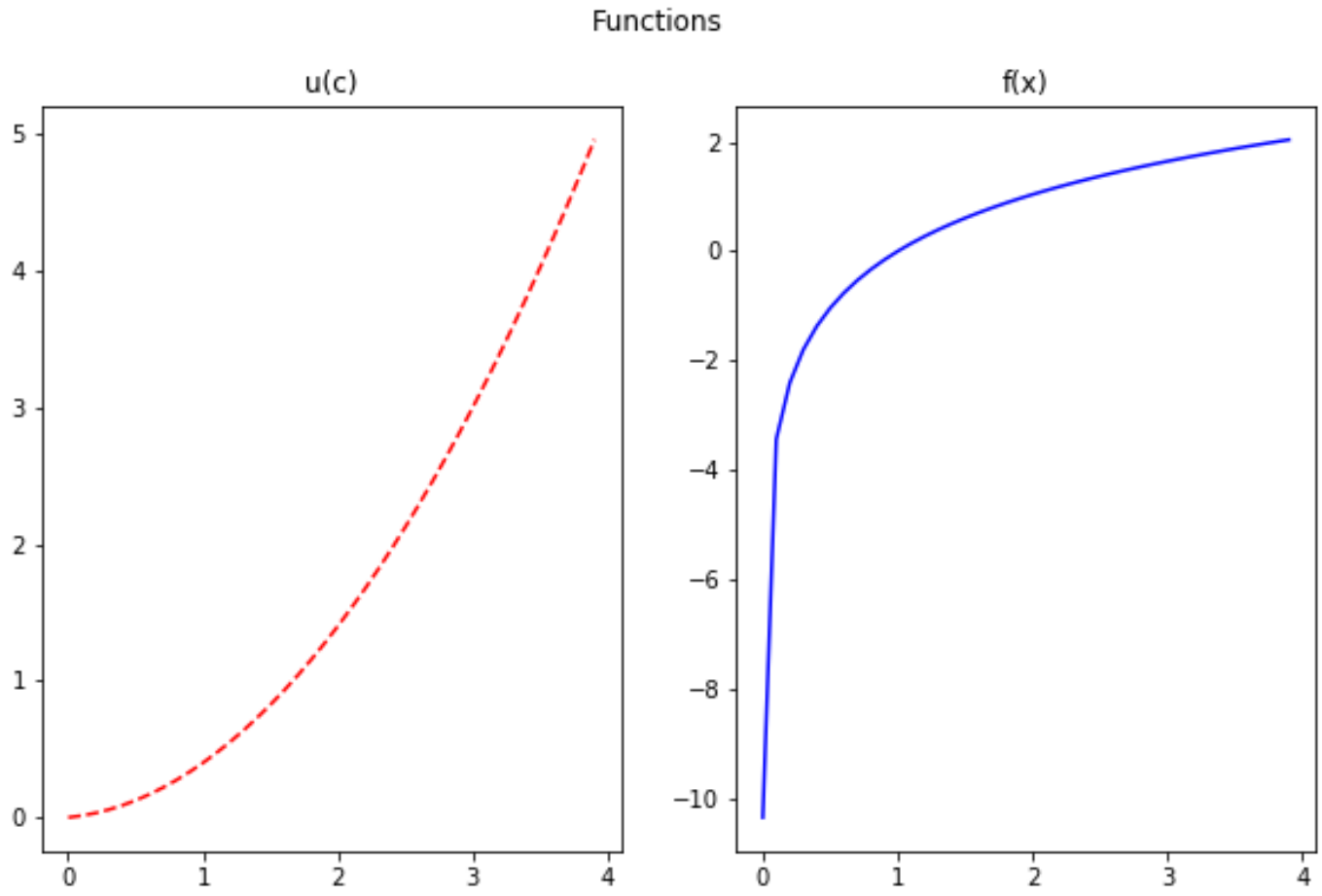
Let's try out more functions and options

```
# define functions
func = lambda x : 0.3*x**2 + 0.1*x
util = lambda x : 1.5*np.log(x)

# create x values
x_vals = np.arange(0.001, 4, 0.1)

# get function output
f_vals = np.array([func(x_val) for x_val in x_vals])
u_vals = np.array([util(x_val) for x_val in x_vals])

plt.clf() # clear current figure
plt.plot(x_vals, f_vals, 'r--', x_vals, u_vals, 'b-')
plt.ylabel('y'); plt.xlabel('x')
plt.title('Plot Functions'); plt.show()
```


Plot Functions
4
2
0
−2
−4
−6
−8
−10
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x

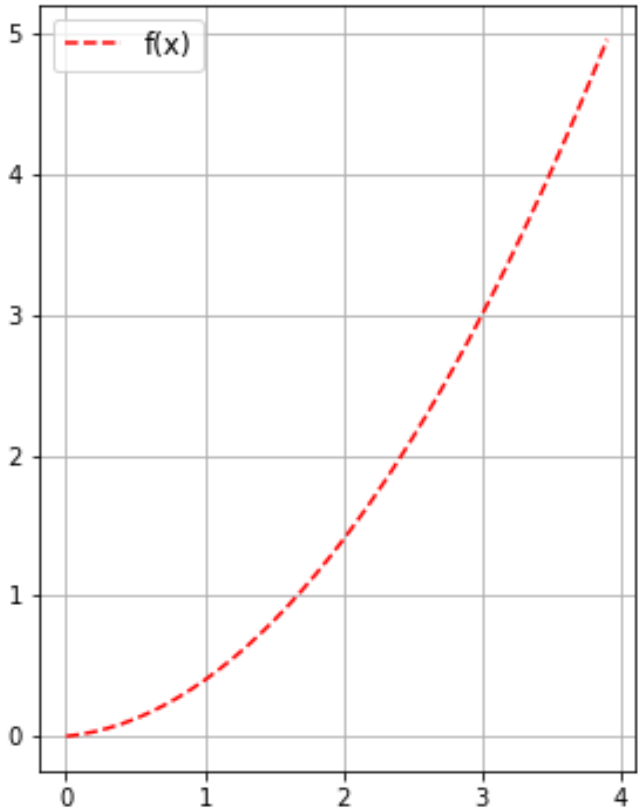
Multiple Subplots

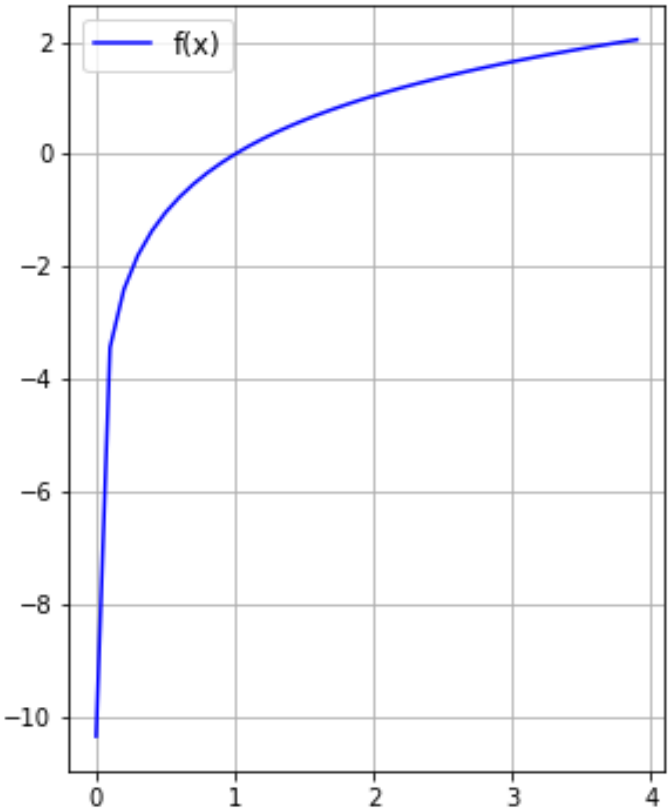
```
# plot the two functions in multiple subplots
plt.figure(2, figsize=(15, 9))
plt.subplot(121);plt.title("u(c)")
plt.plot(x_vals, f_vals, 'r--')
plt.subplot(122);plt.title("f(x)")
plt.plot(x_vals, u_vals, 'b-')
plt.suptitle('Functions')
plt.show()
```


Functions
u(c)
f(x)
0
1
2
3
4
5
−10
−8
−6
−4
−2

```
# another way is to use "subplots"
fig, (ax1, ax2) = plt.subplots(1, 2,
                               figsize=(15, 6))
ax1.plot(x_vals, f_vals, 'r--', label="f(x)");
ax2.plot(x_vals, u_vals, 'b-', label="f(x)");
ax1.grid(True), ax2.grid(True)
ax1.legend(loc='upper left', fontsize='large')
ax2.legend(loc='upper left', fontsize='large')
fig.suptitle('Functions'); plt.show()
```

Functions





Exercise

We have the following (CRRA) utility function

$$U(c) = \frac{c^{1-\eta} - 1}{1 - \eta}$$

1. Plot $U(c)$ for $0 \leq c \leq 10$ and $\eta = 0.5$
2. Now do the same for $\eta = [0.2 \quad 0.4 \quad 0.8 \quad 1]$

$\rightarrow$ What's the problem with $c = 0$ in the second task?

Recommended Tutorials and Links

Here are some links to **great content** for improving your Python skills! And it's all for free.

- https://lectures.quantecon.org/py/
- https://www.tutorialspoint.com/python/index.htm
- https://docs.python.org/3/tutorial/
- https://matplotlib.org/users/pyplot_tutorial.html
- https://www.kevinsheppard.com/Python_for_Econometrics
- https://docs.scipy.org/doc/numpy-1.15.1/user/quickstart.html
- https://docs.scipy.org/doc/scipy/reference/tutorial/
- http://treyhunner.com/

References

Matplotlib documentation. https://matplotlib.org/.

Python documentation. https://docs.python.org/3.6/.

Scipy documentation. https://docs.scipy.org/doc/.

Matthes, E. (2015). *Python Crash Course*. No Starch Press.

Mehta, H. K. (2015). *Mastering Python scientific computing*. Birmingham : Packt Publishing.

Sargent, T. J. and Stachurski, J. (2017). Quantecon lectures. https://lectures.quantecon.org/.