intro_to_python_w_solutions

February 13, 2020

1 Introduction to Python programming

Welcome to the Intorduction to Python workshop. Today we will introduce the main concepts of Python3 programming using this Jupyter (IPython) notebook. We will overview the some elementary topics that have been covered in the AM10IM Introduction to MATLAB module in the first year at Aston Univeristy. As we go through today's session, we will highlight the similarities and differences between Python and MATLAB, but emphaise that the programming principles are almost identical. Hopefully, by the end of the session, you will have mastered the basic Python syntax and some important Python modules used in elementary scientific programming.

1.1 Contents

In today's workshop, we plan to cover the following content

- 1 The Python language
  - 1.1 What is Python?
  - 1.2 Python programming files
  - 1.3 Jupyter notebooks
  - 1.4 Modules
  - 1.5 References
- 2 Elementary scientific programming in Python
  - 2.1 The Math module
  - 2.2 Variables and types
  - 2.3 Operations and comparisons
  - 2.4 Compound types: strings and lists
  - 2.5 Condtional statement
  - 2.6 Loops
  - 2.7 Functions
- 3 Arrays in Python
  - 3.1 The Numpy module
  - 3.2 Array generating functions
  - 3.3 Files read/write commands
  - 3.4 Manipulating arrays
  - 3.5 Linear algebra of arrays
- 4 Plotting in Python
  - 4.1 The Matplotlib module
  - 4.2 The MATLAB and object-orientated API
  - 4.3 2D plotting styles

2 1. The Python Langauge

2.1 1.2 What is Python

Python is a modern, general-purpose, object-oriented, high-level programming language.

General characteristics of Python:

- **clean and simple language:** Easy-to-read and intuitive code, easy-to-learn minimalistic syntax, maintainability scales well with size of projects.
- **expressive language:** Fewer lines of code, fewer bugs, easier to maintain.

Technical details:

- **dynamically typed:** No need to define the type of variables, function arguments or return types.
- **automatic memory management:** No need to explicitly allocate and deallocate memory for variables and data arrays. No memory leak bugs.
- **interpreted:** No need to compile the code. The Python interpreter reads and executes the python code directly.

Advantages:

- The main advantage is ease of programming, minimizing the time required to develop, debug and maintain the code.
- Well designed language that encourage many good programming practices:
- Modular and object-oriented programming, good system for packaging and re-use of code. This often results in more transparent, maintainable and bug-free code.
- Documentation tightly integrated with the code.
- A large standard library, and a large collection of add-on packages.

Disadvantages:

- Since Python is an interpreted and dynamically typed programming language, the execution of python code can be slow compared to compiled statically typed programming languages, such as C and Fortran.
- Somewhat decentralized, with different environment, packages and documentation spread out at different places. Can make it harder to get started.

2.1.1 What makes python suitable for scientific computing?

- Python has a strong position in scientific computing:
  - Large community of users, easy to find help and documentation.
- Extensive ecosystem of scientific libraries and environments
  - numpy: http://numpy.scipy.org - Numerical Python
  - scipy: http://www.scipy.org - Scientific Python
  - matplotlib: http://www.matplotlib.org - graphics library
- Great performance due to close integration with time-tested and highly optimized codes written in C and Fortran:
  - blas, atlas blas, lapack, arpack, Intel MKL, …

- Good support for
  - Parallel processing with processes and threads
  - Interprocess communication (MPI)
  - GPU computing (OpenCL and CUDA)
- Readily available and suitable for use on high-performance computing clusters.
- No license costs, no unnecessary use of research budget.

2.1.2 Python environments

Python is not only a programming language, but often also refers to the standard implementation of the interpreter (technically referred to as CPython) that actually runs the python code on a computer.

There are also many different environments through which the python interpreter can be used. Each environment has different advantages and is suitable for different workflows. One strength of python is that it is versatile and can be used in complementary ways, but it can be confusing for beginners so we will start with a brief survey of python environments that are useful for scientific computing.

2.1.3 Python interpreter

The standard way to use the Python programming language is to use the Python interpreter to run python code. The python interpreter is a program that reads and executes the python code in files passed to it as arguments. At the command prompt, the command `python` is used to invoke the Python interpreter.

For example, to run a file `my-program.py` that contains python code from the command prompt, use:

```
$ python my-program.py
```

We can also start the interpreter by simply typing `python` at the command line, and interactively type python code into the interpreter.

This is often how we want to work when developing scientific applications, or when doing small calculations. But the standard python interpreter is not very convenient for this kind of work, due to a number of limitations.

2.1.4 Versions of Python

There are currently two versions of python: Python 2 and Python 3. Python 3 will eventually supercede Python 2, but it is not backward-compatible with Python 2. A lot of existing python code and packages has been written for Python 2, and it is still the most wide-spread version. For these lectures either version will be fine, but it is probably easier to stick with Python 2 for now, because it is more readily available via prebuilt packages and binary installers.

To see which version of Python you have, run

```
$ python2 --version
Python 2.7.17
$ python3 --version
```

```
Python 3.7.6
```

Several versions of Python can be installed in parallel, as shown above.

2.2 1.1 Python program files

- Python code is usually stored in text files with the file ending "`.py`":

  ```
  myprogram.py
  ```

- Every line in a Python program file is assumed to be a Python statement, or part thereof.
  - The only exception is comment lines, which start with the character `#` (optionally preceded by an arbitrary number of white-space characters, i.e., tabs or spaces). Comment lines are usually ignored by the Python interpreter.
- To run our Python program from the command line we use:

  ```
  $ python myprogram.py
  ```

2.3 1.2 Jupyter notebooks

This file - a Jupyter notebook - does not follow the standard pattern with Python code in a text file. Instead, an Jupyter notebook is stored as a file in the JSON format. The advantage is that we can mix formatted text, Python code and code output. It requires the Jupyter notebook server to run it though, and therefore isn't a stand-alone Python program as described above. Other than that, there is no difference between the Python code that goes into a program file or an Jupyter notebook.

2.4 1.3 Modules

Most of the functionality in Python is provided by *modules*. The Python Standard Library is a large collection of modules that provides *cross-platform* implementations of common facilities such as access to the operating system, file I/O, string management, network communication, and much more.

2.5 1.4 References

- The Python Language Reference: http://docs.python.org/3/reference/index.html
- The Python Standard Library: http://docs.python.org/3/library/

To use a module in a Python program it first has to be imported. A module can be imported using the `import` statement. For example, to import the module `math`, which contains many standard mathematical functions, we can do:

3 2. Elementary Scientific Programming in Python

3.1 2.1. The Math module

[3]:
```
import math
```

This includes the whole module and makes it available for use later in the program. For example, we can do:

[4]:
```python
import math

x = math.cos(2 * math.pi)

print(x)
```

```
1.0
```

In Python, the function `print(...)` is used to output the value of the variable to the command window.

Alternatively, we can chose to import all symbols (functions and variables) in a module to the current namespace (so that we don't need to use the prefix "`math.`" every time we use something from the `math` module:

[5]:
```python
from math import *

x = cos(2 * pi)

print(x)
```

```
1.0
```

This pattern can be very convenient, but in large programs that include many modules it is often a good idea to keep the symbols from each module in their own namespaces, by using the `import math` pattern. This would elminate potentially confusing problems with name space collisions.

As a third alternative, we can chose to import only a few selected symbols from a module by explicitly listing which ones we want to import instead of using the wildcard character `*`:

[6]:
```python
from math import cos, pi

x = cos(2 * pi)

print(x)
```

```
1.0
```

3.1.1 Looking at what a module contains, and its documentation

Once a module is imported, we can list the symbols it provides using the `dir` function:

[7]:
```python
import math

print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

And using the function `help` we can get a description of each function (almost .. not all functions have docstrings, as they are technically called, but the vast majority of functions are documented this way).

[8]:
```
help(math.log)
```

```
Help on built-in function log in module math:

log(…)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base not specified, returns the natural logarithm (base e) of x.
```

[9]:
```
math.log(10)
```

[9]:
```
2.302585092994046
```

[10]:
```
math.log(10, 2)
```

[10]:
```
3.3219280948873626
```

We can also use the `help` function directly on modules: Try

```
help(math)
```

Some very useful modules form the Python standard library are `os`, `sys`, `math`, `shutil`, `re`, `subprocess`, `multiprocessing`, `threading`.

A complete lists of standard modules for Python 3 are available at http://docs.python.org/3/library/, respectively.

3.2 2.1 Variables and types

3.2.1 Symbol names

Variable names in Python can contain alphanumerical characters `a-z`, `A-Z`, `0-9` and some special characters such as `_`. Normal variable names must start with a letter.

By convention, variable names start with a lower-case letter, and Class names start with a capital letter.

In addition, there are a number of Python keywords that cannot be used as variable names. These keywords are:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Note: Be aware of the keyword `lambda`, which could easily be a natural variable name in a scientific program. But being a keyword, it cannot be used as a variable name.

3.2.2 Assignment

The assignment operator in Python is `=`. Python is a dynamically typed language, so we do not need to specify the type of a variable when we create one.

Assigning a value to a new variable creates the variable:

[11]:
```
# variable assignments
x = 1.0
my_variable = 12.2
```

Although not explicitly specified, a variable does have a type associated with it. The type is derived from the value that was assigned to it.

[12]:
```
type(x)
```

[12]:
```
float
```

If we assign a new value to a variable, its type can change.

[13]:
```
x = 1
```

[14]:
```
type(x)
```

[14]:
```
int
```

If we try to use a variable that has not yet been defined we get an `NameError`:

[15]:
```
print(y)
```

```
       ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-15-d9183e048de3> in <module>
   ----> 1 print(y)


       NameError: name 'y' is not defined
```

3.2.3 Fundamental types

[ ]:
```
# integers
x = 1
type(x)
```

[16]:
```
# float
x = 1.0
type(x)
```

[16]:
```
float
```

[17]:
```
# boolean
b1 = True
b2 = False

type(b1)
```

[17]:
```
bool
```

[18]:
```
# complex numbers: note the use of `j` to specify the imaginary part
x = 1.0 - 1.0j
type(x)
```

[18]:
```
complex
```

[19]:
```
print(x)
```

```
(1-1j)
```

[20]:
```
print(x.real, x.imag)
```

```
1.0 -1.0
```

3.2.4 Type utility functions

The module `types` contains a number of type name definitions that can be used to test if variables are of certain types:

[21]:
```
import types

# print all types defined in the `types` module
print(dir(types))
```

```
['AsyncGeneratorType', 'BuiltinFunctionType', 'BuiltinMethodType',
'ClassMethodDescriptorType', 'CodeType', 'CoroutineType',
'DynamicClassAttribute', 'FrameType', 'FunctionType', 'GeneratorType',
'GetSetDescriptorType', 'LambdaType', 'MappingProxyType',
'MemberDescriptorType', 'MethodDescriptorType', 'MethodType',
'MethodWrapperType', 'ModuleType', 'SimpleNamespace', 'TracebackType',
'WrapperDescriptorType', '_GeneratorWrapper', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', '_calculate_meta', 'coroutine', 'new_class', 'prepare_class',
'resolve_bases']
```

[22]:
```
x = 1.0
# check if the variable x is a float
type(x) is float
```

[22]:
```
True
```

[23]:
```
# check if the variable x is an int
type(x) is int
```

[23]:
```
False
```

We can also use the `isinstance` method for testing types of variables:

[24]:
```
isinstance(x, float)
```

[24]:
```
True
```

3.2.5 Type casting

[25]:
```
x = 1.5

print(x, type(x))
```

```
1.5 <class 'float'>
```

[26]:
```
x = int(x)

print(x, type(x))
```

```
1 <class 'int'>
```

[27]:
```
z = complex(x)

print(z, type(z))
```

```
(1+0j) <class 'complex'>
```

[28]:
```
x = float(z)
```

```
          ␣
↪---------------------------------------------------------------------------
```

```
        TypeError                                 Traceback (most recent call␣
↪last)

        <ipython-input-28-19c840f40bd8> in <module>
    ----> 1 x = float(z)


        TypeError: can't convert complex to float
```

Complex variables cannot be cast to floats or integers. We need to use `z.real` or `z.imag` to extract the part of the complex number we want:

[108]:
```
y = bool(z.real)

print(z.real, " -> ", y, type(y))

y = bool(z.imag)

print(z.imag, " -> ", y, type(y))
```

```
1.0  ->  True <class 'bool'>
0.0  ->  False <class 'bool'>
```

3.3 2.3 Operators and comparisons

Most operators and comparisons in Python work as one would expect:

- Arithmetic operators `+`, `-`, `*`, `/`, `//` (integer division), `**` power, `%` modulo division

[109]:
```
1 + 2, 1 - 2, 1 * 2, 1 / 2
```

[109]:
```
(3, -1, 2, 0.5)
```

[110]:
```
1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0
```

[110]:
```
(3.0, -1.0, 2.0, 0.5)
```

[111]:
```
# Integer division of float numbers
3.0 // 2.0
```

[111]:
```
1.0
```

[112]:
```
# Note! The power operators in python isn't ^, but **
2 ** 4
```

[112]:
```
16
```

[113]:
```
# modulo division
14 % 3
```

[113]:
```
2
```

Note: The `/` operator always performs a floating point division in Python 3.x. This is not true in Python 2.x, where the result of `/` is always an integer if the operands are integers. to be more specific, `1/2 = 0.5` (`float`) in Python 3.x, and `1/2 = 0` (`int`) in Python 2.x (but `1.0/2 = 0.5` in Python 2.x).

- The boolean operators are spelled out as the words `and`, `not`, `or`.

[114]:
```
True and False
```

[114]:
```
False
```

[115]:
```
not False
```

[115]:
```
True
```

[116]:
```
True or False
```

[116]:
```
True
```

- Comparison operators `>`, `<`, `>=` (greater or equal), `<=` (less or equal), `==` equality, `!=` not equal to, `is` identical.

[117]:
```
2 > 1, 2 < 1
```

[117]:
```
(True, False)
```

[118]:
```
2 > 2, 2 < 2
```

[118]:
```
(False, False)
```

[119]:
```
2 >= 2, 2 <= 2
```

[119]:
```
(True, True)
```

[120]:
```
# equality
[1,2] == [1,2]
```

[120]:
```
True
```

[121]:
```
# not equal to
[3,4] != [6,7]
```

[121]:
```
True
```

[122]:
```
# objects identical?
l1 = l2 = [1,2]

l1 is l2
```

[122]:
```
True
```

3.4 2.4 Compound types: strings and Lists

3.4.1 Strings

Strings are the variable type that is used for storing text messages.

[123]:
```
s = "Hello world"
type(s)
```

[123]:
```
str
```

[124]:
```
# length of the string: the number of characters
len(s)
```

[124]:
```
11
```

[125]:
```
# replace a substring in a string with something else
s2 = s.replace("world", "test")
print(s2)
```

```
Hello test
```

We can index a character in a string using `[]`:

[126]:
```
s[0]
```

[126]:
```
'H'
```

Heads up MATLAB users: Indexing start at 0

We can extract a part of a string using the syntax `[start:stop]`, which extracts characters between index `start` and `stop` -1 (the character at index `stop` is not included):

[127]:
```
s[0:5]
```

[127]:
```
'Hello'
```

[128]:
```
s[4:5]
```

[128]:
```
'o'
```

If we omit either (or both) of `start` or `stop` from `[start:stop]`, the default is the beginning and the end of the string, respectively:

[129]:
```
s[:5]
```

[129]:
```
'Hello'
```

[130]:
```
s[6:]
```

[130]:
```
'world'
```

[131]:
```
s[:]
```

[131]:
```
'Hello world'
```

We can also define the step size using the syntax `[start:end:step]` (the default value for `step` is 1, as we saw above):

[132]:
```
s[::1]
```

[132]:
```
'Hello world'
```

[133]:
```
s[::2]
```

[133]:
```
'Hlowrd'
```

This technique is called *slicing*. Read more about the syntax here: http://docs.python.org/release/3.8.0/library/functions.html?highlight=slice#slice

Python has a very rich set of functions for text processing. See for example http://docs.python.org/3/library/string.html for more information.

String formatting examples

[134]:
```
print("str1", "str2", "str3")  # The print statement concatenates strings with␣
 ↪a space
```

```
str1 str2 str3
```

[135]:
```
print("str1", 1.0, False, -1j)  # The print statements converts all arguments␣
 ↪to strings
```

```
str1 1.0 False (-0-1j)
```

[136]:
```
print("str1" + "str2" + "str3") # strings added with + are concatenated without␣
 ↪space
```

```
str1str2str3
```

[137]:
```
value1=1.0
print("value = " + str(value1))
```

```
value = 1.0
```

3.4.2 List

Lists are very similar to strings, except that each element can be of any type.

The syntax for creating lists in Python is `[...]`:

```
[138]: l = [1,2,3,4]

       print(type(l))
       print(l)
```

```
<class 'list'>
[1, 2, 3, 4]
```

We can use the same slicing techniques to manipulate lists as we could use on strings:

```
[139]: print(l)

       print(l[1:3])

       print(l[::2])
```

```
[1, 2, 3, 4]
[2, 3]
[1, 3]
```

Heads up MATLAB users: Indexing starts at 0

```
[140]: l[0]
```

```
[140]: 1
```

Elements in a list do not all have to be of the same type:

```
[141]: l = [1, 'a', 1.0, 1-1j]

       print(l)
```

```
[1, 'a', 1.0, (1-1j)]
```

Python lists can be inhomogeneous and arbitrarily nested:

```
[142]: nested_list = [1, [2, [3, [4, [5]]]]]

       nested_list
```

```
[142]: [1, [2, [3, [4, [5]]]]]
```

Lists play a very important role in Python. For example they are used in loops and other flow control structures. There are a number of convenient functions for generating lists of various types, for example the `range` function:

```
[143]: # in python 3 range generates an iterator, which can be converted to a list␣
        ↪using 'list(...)'.

       start = 10
       stop = 30
       step = 2

       list(range(start, stop, step))
```

```
[143]: [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

```
[144]: list(range(-10, 10))
```

```
[144]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[145]: s
```

```
[145]: 'Hello world'
```

```
[146]: # convert a string to a list by type casting:
       s2 = list(s)

       s2
```

```
[146]: ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

```
[147]: # sorting lists
       s2.sort()

       print(s2)
```

```
[' ', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
```

Adding, inserting, modifying, and removing elements from lists

```
[148]: # create a new empty list
       l = []

       # add an elements using `append`
       l.append("A")
       l.append("d")
       l.append("d")

       print(l)
```

```
['A', 'd', 'd']
```

We can modify lists by assigning new values to elements in the list. In technical jargon, lists are *mutable*.

[149]:
```
l[1] = "p"
l[2] = "p"

print(l)
```

```
['A', 'p', 'p']
```

[150]:
```
l[1:3] = ["d", "d"]

print(l)
```

```
['A', 'd', 'd']
```

Insert an element at an specific index using `insert`

[151]:
```
l.insert(0, "i")
l.insert(1, "n")
l.insert(2, "s")
l.insert(3, "e")
l.insert(4, "r")
l.insert(5, "t")

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'A', 'd', 'd']
```

Remove first element with specific value using 'remove'

[152]:
```
l.remove("A")

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'd', 'd']
```

Remove an element at a specific location using `del`:

[153]:
```
del l[7]
del l[6]

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't']
```

See `help(list)` for more details, or read the online documentation

3.5 2.5 Conditional Statements

3.5.1 Conditional statements: if, elif, else

The Python syntax for conditional execution of code uses the keywords `if`, `elif` (else if), `else`:

[154]:
```
statement1 = False
statement2 = False

if statement1:
    print("statement1 is True")

elif statement2:
    print("statement2 is True")

else:
    print("statement1 and statement2 are False")
```

```
statement1 and statement2 are False
```

For the first time, here we encounted a peculiar and unusual aspect of the Python programming language: Program blocks are defined by their indentation level.

The extent of a code block is defined by the indentation level (usually a tab or say four white spaces). This means that we have to be careful to indent our code correctly, or else we will get syntax errors.

Examples:

[155]:
```
statement1 = statement2 = True

if statement1:
    if statement2:
        print("both statement1 and statement2 are True")
```

```
both statement1 and statement2 are True
```

[156]:
```
# Bad indentation!
if statement1:
    if statement2:
    print("both statement1 and statement2 are True")  # this line is not␣
 ↪properly indented
```

```
      File "<ipython-input-156-ac4109c9123a>", line 4
    print("both statement1 and statement2 are True")  # this line is not␣
↪properly indented
        ^
IndentationError: expected an indented block
```

[267]:
```
statement1 = False

if statement1:
    print("printed if statement1 is True")

    print("still inside the if block")
```

[268]:
```
if statement1:
    print("printed if statement1 is True")

print("now outside the if block")
```

```
now outside the if block
```

3.6 2.6 Loops

In Python, loops can be programmed in a number of different ways. The most common is the `for` loop, which is used together with iterable objects, such as lists. The basic syntax is:

3.6.1 `for` loops:

[269]:
```
for x in [1,2,3]:
    print(x)
```

```
1
2
3
```

The `for` loop iterates over the elements of the supplied list, and executes the containing block once for each element. Any kind of list can be used in the `for` loop. For example:

[270]:
```
for x in range(4): # by default range start at 0
    print(x)
```

```
0
1
2
3
```

Note: `range(4)` does not include 4 !

[271]:
```
for x in range(-3,3):
    print(x)
```

```
-3
-2
-1
0
1
2
```

[272]:
```
for word in ["scientific", "computing", "with", "python"]:
    print(word)
```

```
scientific
computing
with
python
```

Sometimes it is useful to have access to the indices of the values when iterating over a list. We can use the `enumerate` function for this:

[273]:
```
for idx, x in enumerate(range(-3,3)):
    print(idx, x)
```

```
0 -3
1 -2
2 -1
3 0
4 1
5 2
```

3.6.2 List comprehensions: Creating lists using `for` loops:

A convenient and compact way to initialize lists:

[274]:
```
l1 = [x**2 for x in range(0,5)]

print(l1)
```

```
[0, 1, 4, 9, 16]
```

3.6.3 `while` loops:

[275]:
```
i = 0

while i < 5:
    print(i)

    i = i + 1

print("done")
```

```
0
1
2
3
4
done
```

Note that the `print("done")` statement is not part of the `while` loop body because of the difference in indentation.

3.7 Problem (15 minutes)

Instructions: Write a python code that prints all the primes between 1 and 100. [**Hint:** use two *for loops*, the first iterating through the numbers 1 to 100, and the second through possible divisors: 1 to 10.]

[276]:
```python
import math
N=100

for i in range(2,N+1):
    isprime = True
    sqi=int(math.sqrt(i))
    for j in range(2,sqi+1):
                if i%j==0:
                     isprime = False
                     break
    if(isprime is True):
        print(i)
```

```
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
```

3.8 2.7 Functions

A function in Python is defined using the keyword `def`, followed by a function name, a signature within parentheses `()`, and a colon `:`. The following code, with one additional level of indentation, is the function body.

[277]:
```
def func0():
    print("test")
```

[278]:
```
func0()
```

```
test
```

Optionally, but highly recommended, we can define a so called "docstring", which is a description of the functions purpose and behaviour. The docstring should follow directly after the function definition, before the code in the function body.

[279]:
```
def func1(s):
    """
    Print a string 's' and tell how many characters it has
    """

    print(s + " has " + str(len(s)) + " characters")
```

[280]:
```
help(func1)
```

```
Help on function func1 in module __main__:

func1(s)
    Print a string 's' and tell how many characters it has
```

[281]:
```
func1("test")
```

```
test has 4 characters
```

Functions that returns a value use the `return` keyword:

[282]:
```
def square(x):
    """
    Return the square of x.
    """
    return x ** 2
```

[283]:
```
square(4)
```

[283]:
```
16
```

We can return multiple values from a function using tuples (see above):

[284]:
```
def powers(x):
    """
    Return a few powers of x.
    """
    return x ** 2, x ** 3, x ** 4
```

[285]:
```
powers(3)
```

[285]:
```
(9, 27, 81)
```

[286]:
```
x2, x3, x4 = powers(3)

print(x3)
```

```
27
```

3.8.1 Default argument and keyword arguments

In a definition of a function, we can give default values to the arguments the function takes:

[287]:
```
def myfunc(x, p=2, debug=False):
    if debug:
        print("evaluating myfunc for x = " + str(x) + " using exponent p = " +␣
 ↪str(p))
    return x**p
```

If we don't provide a value of the `debug` argument when calling the the function `myfunc` it defaults to the value provided in the function definition:

[288]:
```
myfunc(5)
```

[288]:
```
25
```

[289]:
```
myfunc(5, debug=True)
```

```
evaluating myfunc for x = 5 using exponent p = 2
```

[289]:
```
25
```

If we explicitly list the name of the arguments in the function calls, they do not need to come in the same order as in the function definition. This is called *keyword* arguments, and is often very useful in functions that takes a lot of optional arguments.

[290]:
```
myfunc(p=3, debug=True, x=7)
```

```
evaluating myfunc for x = 7 using exponent p = 3
```

[290]:
```
343
```

4 Problem (10 minutes)

Instructions: Write a python function called *mycomplex* that takes in a complex number z and outputs both the amplitude r and argument θ of the complex number where $z = r\exp(i\theta)$. Ensure that you include a docstring in the function. **Hint:** use the function `phase` found in the library `cmath`.

[291]:
```python
import cmath
def mycomplex(z):
    """
    function that reads in a complex number and outputs both the amplitude and␣
 ↪argument of the number.
    """
    return abs(z), cmath.phase(z)

mycomplex(1-3j)
```

[291]:
```
(3.1622776601683795, -1.2490457723982544)
```

5 3 Arrays in Python

5.1 3.1 The Numpy module

The `numpy` package (module) is used in almost all numerical computation using Python. It is a package that provide high-performance vector, matrix and higher-dimensional data structures for Python. It is implemented in C and Fortran so when calculations are vectorized (formulated with vectors and matrices), performance is very good.

To use `numpy` you need to import the module, using for example:

[292]:
```python
from numpy import *
```

In the `numpy` package the terminology used for vectors, matrices and higher-dimensional data sets is *array*.

5.2 Creating `numpy` arrays

There are a number of ways to initialize new numpy arrays, for example from

- a Python list or tuples
- using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc.
- reading data from files

5.2.1 From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function.

[293]:
```
# a vector: the argument to the array function is a Python list
v = array([1,2,3,4])

v
```

[293]:
```
array([1, 2, 3, 4])
```

[294]:
```
# a matrix: the argument to the array function is a nested Python list
M = array([[1, 2], [3, 4]])

M
```

[294]:
```
array([[1, 2],
       [3, 4]])
```

The `v` and `M` objects are both of the type `ndarray` that the `numpy` module provides.

[295]:
```
type(v), type(M)
```

[295]:
```
(numpy.ndarray, numpy.ndarray)
```

The difference between the `v` and `M` arrays is only their shapes. We can get information about the shape of an array by using the `ndarray.shape` property.

[296]:
```
v.shape
```

[296]:
```
(4,)
```

[297]:
```
M.shape
```

[297]:
```
(2, 2)
```

The number of elements in the array is available through the `ndarray.size` property:

[298]:
```
M.size
```

[298]:
```
4
```

Equivalently, we could use the function `numpy.shape` and `numpy.size`

[299]:
```
shape(M)
```

[299]:
```
(2, 2)
```

[300]:
```
size(M)
```

[300]:
```
4
```

So far the `numpy.ndarray` looks awefully much like a Python list (or nested list). Why not simply use Python lists for computations instead of creating a new array type?

There are several reasons:

- Python lists are very general. They can contain any kind of object. They are dynamically typed. They do not support mathematical functions such as matrix and dot multiplications, etc. Implementing such functions for Python lists would not be very efficient because of the dynamic typing.
- Numpy arrays are **statically typed** and **homogeneous**. The type of the elements is determined when the array is created.
- Numpy arrays are memory efficient.
- Because of the static typing, fast implementation of mathematical functions such as multiplication and addition of `numpy` arrays can be implemented in a compiled language (C and Fortran is used).

Using the `dtype` (data type) property of an `ndarray`, we can see what type the data of an array has:

[301]:
```
M.dtype
```

[301]:
```
dtype('int64')
```

We get an error if we try to assign a value of the wrong type to an element in a numpy array:

[302]:
```
M[0,0] = "hello"
```

```
          ␣
↪---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call␣
↪last)

        <ipython-input-302-e1f336250f69> in <module>
    ----> 1 M[0,0] = "hello"


        ValueError: invalid literal for int() with base 10: 'hello'
```

If we want, we can explicitly define the type of the array data when we create it, using the `dtype` keyword argument:

[376]:
```
M = array([[1, 2], [3, 4]], dtype=complex)

M
```

[376]:
```
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Common data types that can be used with `dtype` are: `int`, `float`, `complex`, `bool`, `object`, etc.

We can also explicitly define the bit size of the data types, for example: `int64`, `int16`, `float128`, `complex128`.

5.3 3.2 Array generating functions

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in `numpy` that generate arrays of different forms. Some of the more common are:

arange

```
[377]: # create a range

       x = arange(0, 10, 1) # arguments: start, stop, step

       x
```

```
[377]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[378]: x = arange(-1, 1, 0.25)

       x
```

```
[378]: array([-1.  , -0.75, -0.5 , -0.25,  0.  ,  0.25,  0.5 ,  0.75])
```

linspace and logspace

```
[379]: # using linspace, both end points ARE included
       linspace(0, 10, 25)
```

```
[379]: array([ 0.        ,  0.41666667,  0.83333333,  1.25      ,  1.66666667,
               2.08333333,  2.5       ,  2.91666667,  3.33333333,  3.75      ,
               4.16666667,  4.58333333,  5.        ,  5.41666667,  5.83333333,
               6.25      ,  6.66666667,  7.08333333,  7.5       ,  7.91666667,
               8.33333333,  8.75      ,  9.16666667,  9.58333333, 10.        ])
```

```
[380]: logspace(0, 10, 10, base=e)
```

```
[380]: array([1.00000000e+00, 3.03773178e+00, 9.22781435e+00, 2.80316249e+01,
              8.51525577e+01, 2.58670631e+02, 7.85771994e+02, 2.38696456e+03,
              7.25095809e+03, 2.20264658e+04])
```

mgrid

```
[381]: x, y = mgrid[0:5, 0:5] # similar to meshgrid in MATLAB
```

```
[382]: x
```

[382]:
```
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]])
```

[383]:
```
y
```

[383]:
```
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

random data

[384]:
```
from numpy import random
```

[385]:
```
# uniform random numbers in [0,1]
random.rand(5,5)
```

[385]:
```
array([[0.4017048 , 0.54656593, 0.21730189, 0.61637805, 0.17892493],
       [0.85903299, 0.41178259, 0.22532383, 0.97964565, 0.60760309],
       [0.21395884, 0.96363992, 0.83357632, 0.53894336, 0.67481238],
       [0.95343869, 0.70192051, 0.70449315, 0.92257619, 0.03963254],
       [0.05244862, 0.2106985 , 0.73197306, 0.91687787, 0.80276846]])
```

[386]:
```
# standard normal distributed random numbers
random.randn(5,5)
```

[386]:
```
array([[ 1.11363355, -1.32649996,  0.47628888,  0.16851162,  0.24595966],
       [-0.90875181, -0.13754911,  0.3032839 ,  1.24606049,  0.26763849],
       [-1.08508818,  0.59254788,  2.10363205,  0.03121841,  0.63522355],
       [ 1.19759995,  1.02440203, -0.46499566,  1.14765173,  0.0648197 ],
       [ 0.20503559,  1.07045946,  0.94072048,  0.04531143, -0.07529438]])
```

diag

[387]:
```
# a diagonal matrix
diag([1,2,3])
```

[387]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

[388]:
```
# diagonal with offset from the main diagonal
diag([1,2,3], k=1)
```

[388]:
```
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
```

zeros and ones

[389]:
```
zeros((3,3))
```

[389]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[390]:
```
ones((3,3))
```

[390]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

5.4 3.3 File read/write commands

[391]:
```
# These lines are need for plotting, more explanation later.
%matplotlib inline
import matplotlib.pyplot as plt
```

5.4.1 Comma-separated values (CSV)

A very common file format for data files is comma-separated values (CSV), or related formats such as TSV (tab-separated values). To read data from such files into Numpy arrays we can use the `numpy.genfromtxt` function. For example,

[392]:
```
!head ./data/stockholm_temp_data.dat
```

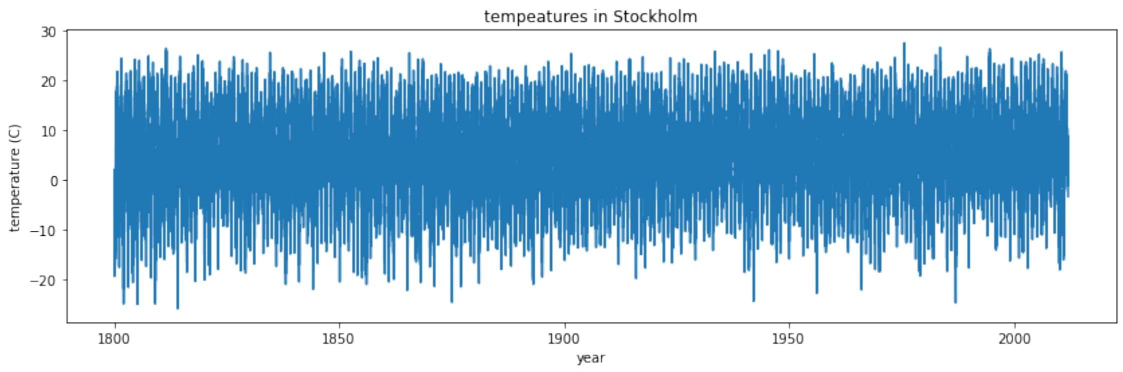
```
# year  month   day     avg temp
1800    1       1       -6.1
1800    1       2       -15.4
1800    1       3       -15
1800    1       4       -19.3
1800    1       5       -16.8
1800    1       6       -11.4
1800    1       7       -7.6
1800    1       8       -7.1
1800    1       9       -10.1
```

[393]:
```
data = genfromtxt('./data/stockholm_temp_data.dat')
```

[394]:
```
data.shape
```

[394]: (77431, 4)

[395]:
```
fig, ax = plt.subplots(figsize=(14,4))
ax.plot(data[:,0]+data[:,1]/12.0+data[:,2]/365, data[:,3])
ax.axis('tight')
ax.set_title('tempeatures in Stockholm')
ax.set_xlabel('year')
ax.set_ylabel('temperature (C)');
```



Using `numpy.savetxt` we can store a Numpy array to a file in CSV format:

[396]:
```
M = random.rand(3,3)

M
```

[396]:
```
array([[0.03144127, 0.11983682, 0.50825929],
       [0.43428231, 0.63446744, 0.24636   ],
       [0.63991427, 0.47929593, 0.72539592]])
```

[397]:
```
savetxt("random-matrix.csv", M)
```

[398]:
```
!cat random-matrix.csv
```

```
3.144126633947319505e-02 1.198368201178855452e-01 5.082592943857739964e-01
4.342823074314311471e-01 6.344674436355274283e-01 2.463600018721932017e-01
6.399142669270552197e-01 4.792959254651455447e-01 7.253959196013546329e-01
```

[399]:
```
savetxt("random-matrix.csv", M, fmt='%.5f') # fmt specifies the format

!cat random-matrix.csv
```

```
0.03144 0.11984 0.50826
0.43428 0.63447 0.24636
0.63991 0.47930 0.72540
```

5.4.2 Numpy's native file format

Useful when storing and reading back numpy array data. Use the functions `numpy.save` and `numpy.load`:

```
[400]: save("random-matrix.npy", M)

       !file random-matrix.npy
```

```
random-matrix.npy: data
```

```
[401]: load("random-matrix.npy")
```

```
[401]: array([[0.03144127, 0.11983682, 0.50825929],
              [0.43428231, 0.63446744, 0.24636   ],
              [0.63991427, 0.47929593, 0.72539592]])
```

5.5 More properties of the numpy arrays

```
[402]: M.itemsize # bytes per element
```

```
[402]: 8
```

```
[403]: M.nbytes # number of bytes
```

```
[403]: 72
```

```
[404]: M.ndim # number of dimensions
```

```
[404]: 2
```

5.6 3.4 Manipulating arrays

5.6.1 Indexing

We can index elements in an array using square brackets and indices:

```
[405]: # v is a vector, and has only one dimension, taking one index
       v[0]
```

```
[405]: matrix([[1]])
```

```
[406]: # M is a matrix, or a 2 dimensional array, taking two indices
       M[1,1]
```

```
[406]: 0.6344674436355274
```

If we omit an index of a multidimensional array it returns the whole row (or, in general, a N-1 dimensional array)

[407]:
```
M
```

[407]:
```
array([[0.03144127, 0.11983682, 0.50825929],
       [0.43428231, 0.63446744, 0.24636   ],
       [0.63991427, 0.47929593, 0.72539592]])
```

[408]:
```
M[0] # row 0
```

[408]:
```
array([0.03144127, 0.11983682, 0.50825929])
```

The same thing can be achieved with using : instead of an index:

[409]:
```
M[1,:] # row 1
```

[409]:
```
array([0.43428231, 0.63446744, 0.24636   ])
```

[410]:
```
M[:,1] # column 1
```

[410]:
```
array([0.11983682, 0.63446744, 0.47929593])
```

We can assign new values to elements in an array using indexing:

[411]:
```
M[0,0] = 1
```

[412]:
```
M
```

[412]:
```
array([[1.        , 0.11983682, 0.50825929],
       [0.43428231, 0.63446744, 0.24636   ],
       [0.63991427, 0.47929593, 0.72539592]])
```

[413]:
```
# also works for rows and columns
M[1,:] = 0
M[:,2] = -1
```

[414]:
```
M
```

[414]:
```
array([[ 1.        ,  0.11983682, -1.        ],
       [ 0.        ,  0.        , -1.        ],
       [ 0.63991427,  0.47929593, -1.        ]])
```

5.6.2 Index slicing

Index slicing is the technical name for the syntax `M[lower:upper:step]` to extract part of an array:

[415]:
```
A = array([1,2,3,4,5])
A
```

[415]:
```
array([1, 2, 3, 4, 5])
```

```
[416]: A[1:3]
```

```
[416]: array([2, 3])
```

Array slices are *mutable*: if they are assigned a new value the original array from which the slice was extracted is modified:

```
[417]: A[1:3] = [-2,-3]

       A
```

```
[417]: array([ 1, -2, -3,  4,  5])
```

We can omit any of the three parameters in `M[lower:upper:step]`:

```
[418]: A[::] # lower, upper, step all take the default values
```

```
[418]: array([ 1, -2, -3,  4,  5])
```

```
[419]: A[::2] # step is 2, lower and upper defaults to the beginning and end of the␣
        ↪array
```

```
[419]: array([ 1, -3,  5])
```

```
[420]: A[:3] # first three elements
```

```
[420]: array([ 1, -2, -3])
```

```
[421]: A[3:] # elements from index 3
```

```
[421]: array([4, 5])
```

Negative indices counts from the end of the array (positive index from the begining):

```
[422]: A = array([1,2,3,4,5])
```

```
[423]: A[-1] # the last element in the array
```

```
[423]: 5
```

```
[424]: A[-3:] # the last three elements
```

```
[424]: array([3, 4, 5])
```

Index slicing works exactly the same way for multidimensional arrays:

```
[425]: A = array([[n+m*10 for n in range(5)] for m in range(5)])

       A
```

[425]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

[426]:
```
# a block from the original array
A[1:4, 1:4]
```

[426]:
```
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

[427]:
```
# strides
A[::2, ::2]
```

[427]:
```
array([[ 0,  2,  4],
       [20, 22, 24],
       [40, 42, 44]])
```

5.7 Functions for extracting data from arrays and creating arrays

5.7.1 diag

With the diag function we can also extract the diagonal and subdiagonals of an array:

[428]:
```
diag(A)
```

[428]:
```
array([ 0, 11, 22, 33, 44])
```

[429]:
```
diag(A, -1) # lower off diagonal
```

[429]:
```
array([10, 21, 32, 43])
```

5.7.2 take

The `take` function permits taking custom indices

[430]:
```
v2 = arange(-3,3)
v2
```

[430]:
```
array([-3, -2, -1,  0,  1,  2])
```

[431]:
```
row_indices = [1, 3, 5]
v2[row_indices] # fancy indexing
```

[431]:
```
array([-2,  0,  2])
```

```
[432]: v2.take(row_indices)
```

```
[432]: array([-2,  0,  2])
```

But `take` also works on lists and other objects:

```
[433]: take([-3, -2, -1,  0,  1,  2], row_indices)
```

```
[433]: array([-2,  0,  2])
```

5.7.3 choose

Constructs an array by picking elements from several arrays:

```
[434]: which = [1, 0, 1, 0]
       choices = [[-2,-2,-2,-2], [5,5,5,5]]

       choose(which, choices)
```

```
[434]: array([ 5, -2,  5, -2])
```

5.8 3.5 Linear algebra of arrays

Vectorizing code is the key to writing efficient numerical calculation with Python/Numpy. That means that as much as possible of a program should be formulated in terms of matrix and vector operations, like matrix-matrix multiplication.

5.8.1 Scalar-array operations

We can use the usual arithmetic operators to multiply, add, subtract, and divide arrays with scalar numbers.

```
[435]: v1 = arange(0, 5)
       v1
```

```
[435]: array([0, 1, 2, 3, 4])
```

```
[436]: v1 * 2
```

```
[436]: array([0, 2, 4, 6, 8])
```

```
[437]: v1 + 2
```

```
[437]: array([2, 3, 4, 5, 6])
```

```
[438]: A
```

```
[438]: array([[ 0,  1,  2,  3,  4],
              [10, 11, 12, 13, 14],
              [20, 21, 22, 23, 24],
              [30, 31, 32, 33, 34],
              [40, 41, 42, 43, 44]])
```

```
[439]: A * 2
```

```
[439]: array([[ 0,  2,  4,  6,  8],
              [20, 22, 24, 26, 28],
              [40, 42, 44, 46, 48],
              [60, 62, 64, 66, 68],
              [80, 82, 84, 86, 88]])
```

```
[440]: A + 2
```

```
[440]: array([[ 2,  3,  4,  5,  6],
              [12, 13, 14, 15, 16],
              [22, 23, 24, 25, 26],
              [32, 33, 34, 35, 36],
              [42, 43, 44, 45, 46]])
```

5.8.2 Element-wise array-array operations

When we add, subtract, multiply and divide arrays with each other, the default behaviour is **element-wise** operations:

```
[441]: A * A # element-wise multiplication
```

```
[441]: array([[   0,    1,    4,    9,   16],
              [ 100,  121,  144,  169,  196],
              [ 400,  441,  484,  529,  576],
              [ 900,  961, 1024, 1089, 1156],
              [1600, 1681, 1764, 1849, 1936]])
```

```
[442]: v1 * v1
```

```
[442]: array([ 0,  1,  4,  9, 16])
```

If we multiply arrays with compatible shapes, we get an element-wise multiplication of each row:

```
[443]: A.shape, v1.shape
```

```
[443]: ((5, 5), (5,))
```

```
[444]: A * v1
```

```
[444]: array([[  0,   1,   4,   9,  16],
              [  0,  11,  24,  39,  56],
              [  0,  21,  44,  69,  96],
              [  0,  31,  64,  99, 136],
              [  0,  41,  84, 129, 176]])
```

5.8.3 Matrix algebra

What about matrix mutiplication? There are two ways. We can either use the `dot` function, which applies a matrix-matrix, matrix-vector, or inner vector multiplication to its two arguments:

```
[445]: dot(A, A)
```

```
[445]: array([[ 300,  310,  320,  330,  340],
              [1300, 1360, 1420, 1480, 1540],
              [2300, 2410, 2520, 2630, 2740],
              [3300, 3460, 3620, 3780, 3940],
              [4300, 4510, 4720, 4930, 5140]])
```

```
[446]: dot(A, v1)
```

```
[446]: array([ 30, 130, 230, 330, 430])
```

```
[447]: dot(v1, v1)
```

```
[447]: 30
```

Alternatively, we can cast the array objects to the type `matrix`. This changes the behavior of the standard arithmetic operators `+, -, *` to use matrix algebra.

```
[448]: M = matrix(A)
       v = matrix(v1).T # make it a column vector
```

```
[449]: v
```

```
[449]: matrix([[0],
               [1],
               [2],
               [3],
               [4]])
```

```
[450]: M * M
```

```
[450]: matrix([[ 300,  310,  320,  330,  340],
               [1300, 1360, 1420, 1480, 1540],
               [2300, 2410, 2520, 2630, 2740],
               [3300, 3460, 3620, 3780, 3940],
               [4300, 4510, 4720, 4930, 5140]])
```

```
[451]: M * v
```

```
[451]: matrix([[ 30],
                [130],
                [230],
                [330],
                [430]])
```

```
[452]: # inner product
       v.T * v
```

```
[452]: matrix([[30]])
```

```
[453]: # with matrix objects, standard matrix algebra applies
       v + M*v
```

```
[453]: matrix([[ 30],
                [131],
                [232],
                [333],
                [434]])
```

If we try to add, subtract or multiply objects with incomplatible shapes we get an error:

```
[454]: v = matrix([1,2,3,4,5,6]).T
```

```
[455]: shape(M), shape(v)
```

```
[455]: ((5, 5), (6, 1))
```

```
[456]: M * v
```

```
       ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-456-e8f88679fe45> in <module>
   ----> 1 M * v


       /opt/local/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.
↪7/site-packages/numpy/matrixlib/defmatrix.py in __mul__(self, other)
       218         if isinstance(other, (N.ndarray, list, tuple)) :
       219             # This promotes 1-D vectors to row vectors
   --> 220             return N.dot(self, asmatrix(other))
```

```
    221         if isscalar(other) or not hasattr(other, '__rmul__') :
    222             return N.dot(self, other)


    <__array_function__ internals> in dot(*args, **kwargs)


    ValueError: shapes (5,5) and (6,1) not aligned: 5 (dim 1) != 6 (dim 0)
```

See also the related functions: `inner`, `outer`, `cross`, `kron`, `tensordot`. Try for example `help(kron)`.

5.8.4 Array/Matrix transformations

Above we have used the `.T` to transpose the matrix object `v`. We could also have used the `transpose` function to accomplish the same thing.

Other mathematical functions that transform matrix objects are:

```
[457]: C = matrix([[1j, 2j], [3j, 4j]])
       C
```

```
[457]: matrix([[0.+1.j, 0.+2.j],
               [0.+3.j, 0.+4.j]])
```

```
[458]: conjugate(C)
```

```
[458]: matrix([[0.-1.j, 0.-2.j],
               [0.-3.j, 0.-4.j]])
```

Hermitian conjugate: transpose + conjugate

```
[459]: C.H
```

```
[459]: matrix([[0.-1.j, 0.-3.j],
               [0.-2.j, 0.-4.j]])
```

We can extract the real and imaginary parts of complex-valued arrays using `real` and `imag`:

```
[460]: real(C) # same as: C.real
```

```
[460]: matrix([[0., 0.],
               [0., 0.]])
```

```
[461]: imag(C) # same as: C.imag
```

```
[461]: matrix([[1., 2.],
               [3., 4.]])
```

Or the complex argument and absolute value

```
[462]: angle(C+1) # heads up MATLAB Users, angle is used instead of arg
```

```
[462]: matrix([[0.78539816, 1.10714872],
                [1.24904577, 1.32581766]])
```

```
[463]: abs(C)
```

```
[463]: matrix([[1., 2.],
                [3., 4.]])
```

5.8.5 Matrix computations

```
[464]: C = matrix([[1, 2], [3, 4]])
       C
```

```
[464]: matrix([[1, 2],
                [3, 4]])
```

Inverse

```
[465]: linalg.inv(C) # equivalent to C.I
```

```
[465]: matrix([[-2. ,  1. ],
                [ 1.5, -0.5]])
```

```
[466]: C.I * C
```

```
[466]: matrix([[1.00000000e+00, 0.00000000e+00],
                [2.22044605e-16, 1.00000000e+00]])
```

Determinant

```
[467]: linalg.det(C)
```

```
[467]: -2.0000000000000004
```

```
[468]: linalg.det(C.I)
```

```
[468]: -0.49999999999999967
```

5.8.6 Data processing

Often it is useful to store datasets in Numpy arrays. Numpy provides a number of functions to calculate statistics of datasets in arrays.

For example, let's calculate some properties from the Stockholm temperature dataset used above.

[469]:
```
# reminder, the tempeature dataset is stored in the data variable:
shape(data)
```

[469]:
```
(77431, 4)
```

mean

[470]:
```
# the temperature data is in column 3
mean(data[:,3])
```

[470]:
```
5.785461895106611
```

The daily mean temperature in Stockholm over the last 200 years has been about 5.8 C.

standard deviations and variance

[471]:
```
std(data[:,3]), var(data[:,3])
```

[471]:
```
(8.195919794033603, 67.17310127023183)
```

min and max

[472]:
```
# lowest daily average temperature
data[:,3].min()
```

[472]:
```
-25.8
```

[473]:
```
# highest daily average temperature
data[:,3].max()
```

[473]:
```
27.5
```

sum, prod, and trace

[474]:
```
d = arange(0, 10)
d
```

[474]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[475]:
```
# sum up all elements
sum(d)
```

[475]:
```
45
```

[476]:
```
# product of all elements
prod(d+1)
```

[476]:
```
3628800
```

```
[477]: # cummulative sum
       cumsum(d)
```

```
[477]: array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45])
```

```
[478]: # cummulative product
       cumprod(d+1)
```

```
[478]: array([      1,       2,       6,      24,     120,     720,    5040,
                 40320,  362880, 3628800])
```

```
[479]: # same as: diag(A).sum()
       trace(A)
```

```
[479]: 110
```

5.8.7 Calculations with higher-dimensional data

When functions such as `min`, `max`, etc. are applied to a multidimensional arrays, it is sometimes useful to apply the calculation to the entire array, and sometimes only on a row or column basis. Using the `axis` argument we can specify how these functions should behave:

```
[480]: m = random.rand(3,3)
       m
```

```
[480]: array([[0.98874807, 0.35087561, 0.22418821],
              [0.74518137, 0.25036921, 0.09063451],
              [0.19577825, 0.27507988, 0.89384394]])
```

```
[481]: # global max
       m.max()
```

```
[481]: 0.9887480684567426
```

```
[482]: # max in each column
       m.max(axis=0)
```

```
[482]: array([0.98874807, 0.35087561, 0.89384394])
```

```
[483]: # max in each row
       m.max(axis=1)
```

```
[483]: array([0.98874807, 0.74518137, 0.89384394])
```

Many other functions and methods in the `array` and `matrix` classes accept the same (optional) `axis` keyword argument.

5.9 Reshaping, resizing and stacking arrays

The shape of an Numpy array can be modified without copying the underlaying data, which makes it a fast operation even for large arrays.

```
[484]: A
```

```
[484]: array([[ 0,  1,  2,  3,  4],
              [10, 11, 12, 13, 14],
              [20, 21, 22, 23, 24],
              [30, 31, 32, 33, 34],
              [40, 41, 42, 43, 44]])
```

```
[485]: n, m = A.shape
```

```
[486]: B = A.reshape((1,n*m))
       B
```

```
[486]: array([[ 0,  1,  2,  3,  4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30,
               31, 32, 33, 34, 40, 41, 42, 43, 44]])
```

```
[487]: B[0,0:5] = 5 # modify the array

       B
```

```
[487]: array([[ 5,  5,  5,  5,  5, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30,
               31, 32, 33, 34, 40, 41, 42, 43, 44]])
```

```
[488]: A # and the original variable is also changed. B is only a different view of␣
        ↪the same data
```

```
[488]: array([[ 5,  5,  5,  5,  5],
              [10, 11, 12, 13, 14],
              [20, 21, 22, 23, 24],
              [30, 31, 32, 33, 34],
              [40, 41, 42, 43, 44]])
```

We can also use the function `flatten` to make a higher-dimensional array into a vector. But this function create a copy of the data.

```
[489]: B = A.flatten()

       B
```

```
[489]: array([ 5,  5,  5,  5,  5, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
              32, 33, 34, 40, 41, 42, 43, 44])
```

```
[490]: B[0:5] = 10

       B
```

```
[490]: array([10, 10, 10, 10, 10, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
              32, 33, 34, 40, 41, 42, 43, 44])
```

```
[491]: A # now A has not changed, because B's data is a copy of A's, not refering to␣
        ↪the same data
```

```
[491]: array([[ 5,  5,  5,  5,  5],
              [10, 11, 12, 13, 14],
              [20, 21, 22, 23, 24],
              [30, 31, 32, 33, 34],
              [40, 41, 42, 43, 44]])
```

6 Problem (10 minutes)

Instructions: Find the solution to the following problem by writing a Python code.

$$\begin{aligned}
2x_1 + x_2 - x_3 + x_4 - 3x_5 &= 7, && (1)\\
x_1 + 2x_3 - x_4 + x_5 &= 2, && (2)\\
-2x_2 - x_3 + x_4 - x_5 &= -5, && (3)\\
3x_1 + x_2 - 4x_3 + 5x_5 &= 6, && (4)\\
x_1 - x_2 - x_3 - x_4 + x_5 &= 3. && (5)\\
& && (6)
\end{aligned}$$

```
[492]: import numpy as np
       A=np.
        ↪array([[2,1,-1,1,-3],[1,0,2,-1,1],[0,-2,-1,1,-1],[3,1,-4,0,5],[1,-1,-1,-1,1]])
       b=np.array([7,2,-5,6,3])
       A=matrix(A)
       b=matrix(b).T
       A.I*b
```

```
[492]: matrix([[ 1.91812865],
               [ 1.96491228],
               [-0.98830409],
               [-3.19298246],
               [-1.13450292]])
```

7 4 Plotting in Python

[493]:
```
# This line configures matplotlib to show figures embedded in the notebook,
# instead of opening a new window for each figure. More about that later.
%matplotlib inline
```

7.1 4.1 The Matplotlib module

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library include:

- Easy to get started
- Support for LaTeX formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

One of the key features of matplotlib that I would like to emphasize, and that I think makes matplotlib highly suitable for generating figures for scientific publications is that all aspects of the figure can be controlled *programmatically*. This is important for reproducibility and convenient when one needs to regenerate the figure with updated data or change its appearance.

More information at the Matplotlib web page: http://matplotlib.org/

To get started using Matplotlib in a Python program, either include the symbols from the `pylab` module (the easy way):

[494]:
```
from pylab import *
```

or import the `matplotlib.pyplot` module under the name `plt` (the tidy way):

[495]:
```
import matplotlib
import matplotlib.pyplot as plt
```

[496]:
```
import numpy as np
```

7.2 4.2 MATLAB-like and object orientiated API

The easiest way to get started with plotting using matplotlib is often to use the MATLAB-like API provided by matplotlib.

It is designed to be compatible with MATLAB's plotting functions, so it is easy to get started with if you are familiar with MATLAB.

To use this API from matplotlib, we need to include the symbols in the `pylab` module:

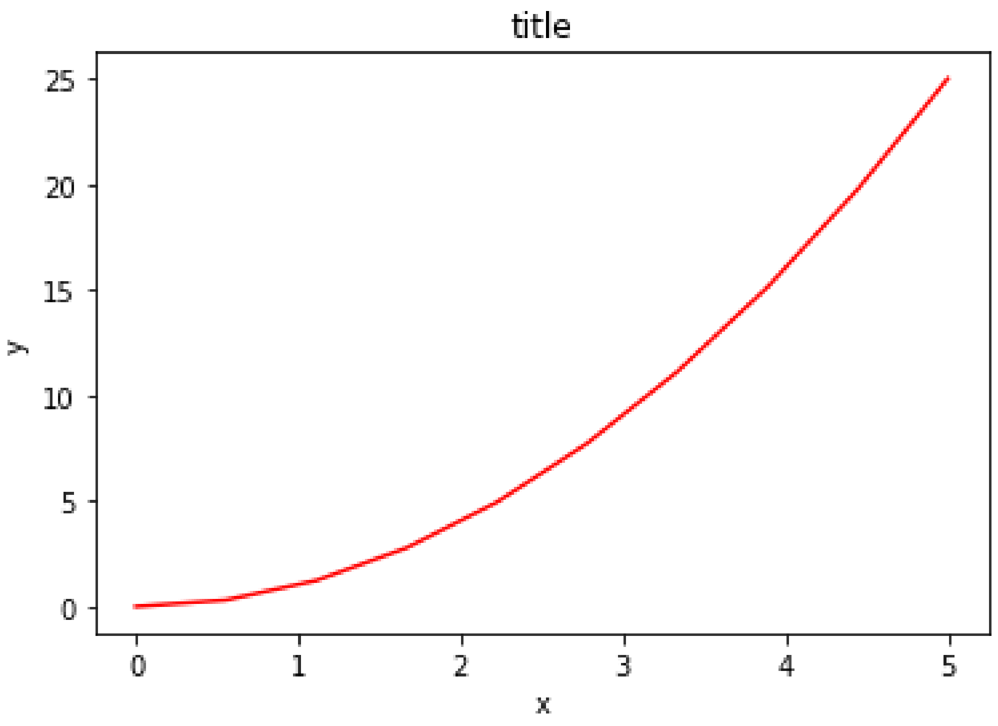
[497]:
```
from pylab import *
```

7.2.1 Example

A simple figure with MATLAB-like plotting API:

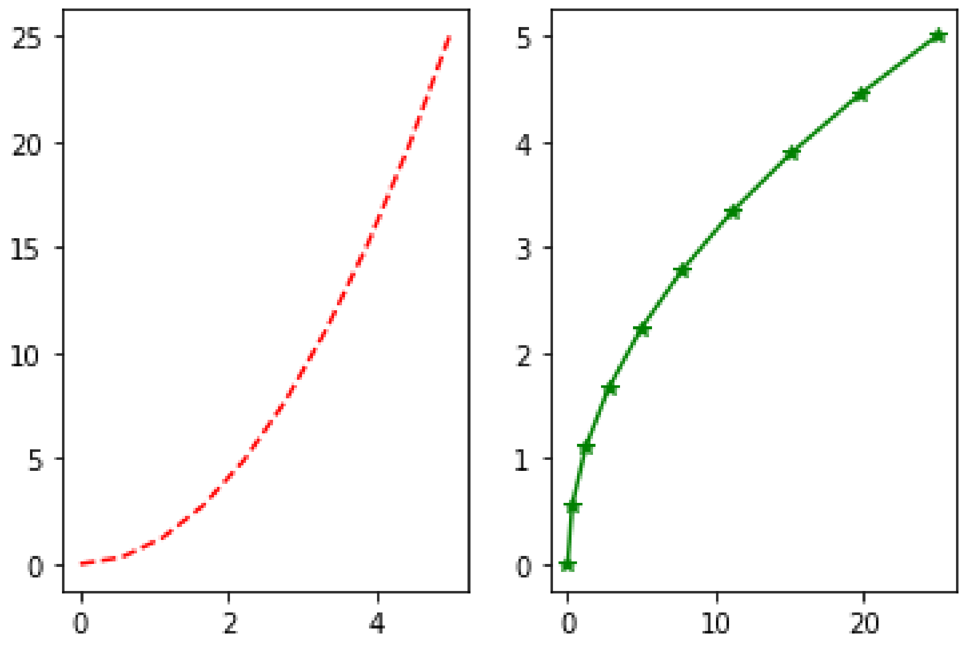
[498]:
```
x = np.linspace(0, 5, 10)
y = x ** 2
```

[499]:
```
figure()
plot(x, y, 'r')
xlabel('x')
ylabel('y')
title('title')
show()
```



Most of the plotting related functions in MATLAB are covered by the `pylab` module. For example, subplot and color/symbol selection:

[500]:
```
subplot(1,2,1)
plot(x, y, 'r--')
subplot(1,2,2)
plot(y, x, 'g*-');
```



The good thing about the pylab MATLAB-style API is that it is easy to get started with if you are familiar with MATLAB, and it has a minumum of coding overhead for simple plots.

However, I'd encourrage not using the MATLAB compatible API for anything but the simplest figures.

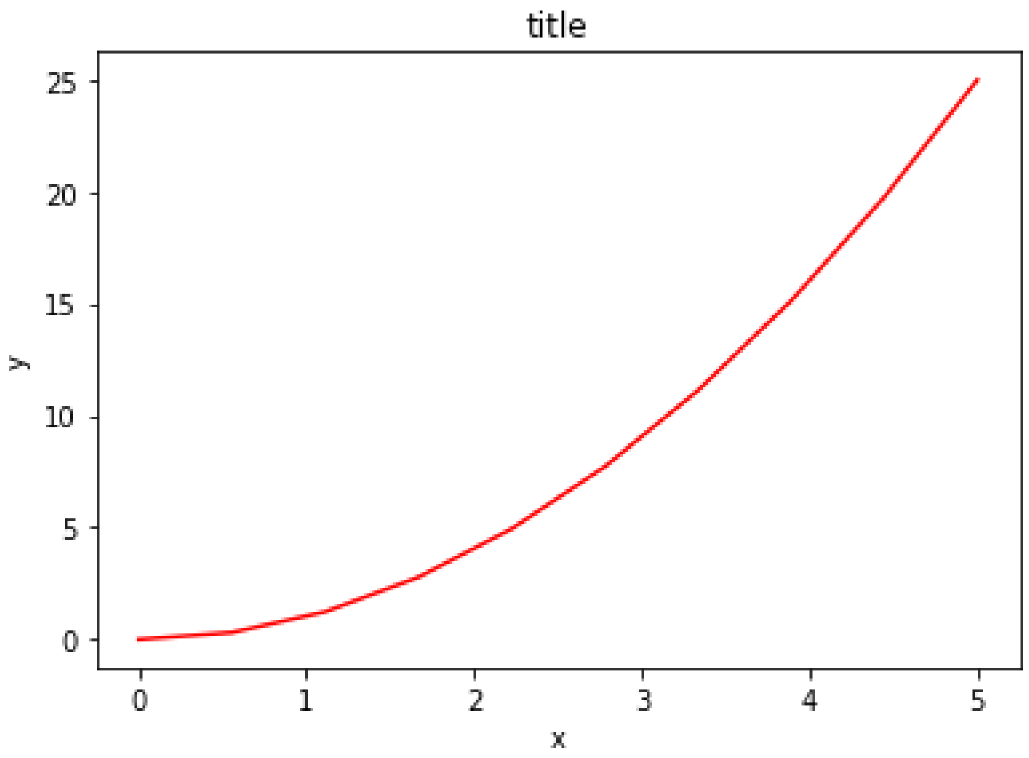
Instead, I recommend learning and using matplotlib's object-oriented plotting API. It is remarkably powerful. For advanced figures with subplots, insets and other components it is very nice to work with.

7.3 The matplotlib object-oriented API

The main idea with object-oriented programming is to have objects that one can apply functions and actions on, and no object or program states should be global (such as the MATLAB-like API). The real advantage of this approach becomes apparent when more than one figure is created, or when a figure contains more than one subplot.

To use the object-oriented API we start out very much like in the previous example, but instead of creating a new global figure instance we store a reference to the newly created figure instance in the `fig` variable, and from it we create a new axis instance `axes` using the `add_axes` method in the `Figure` class instance `fig`:

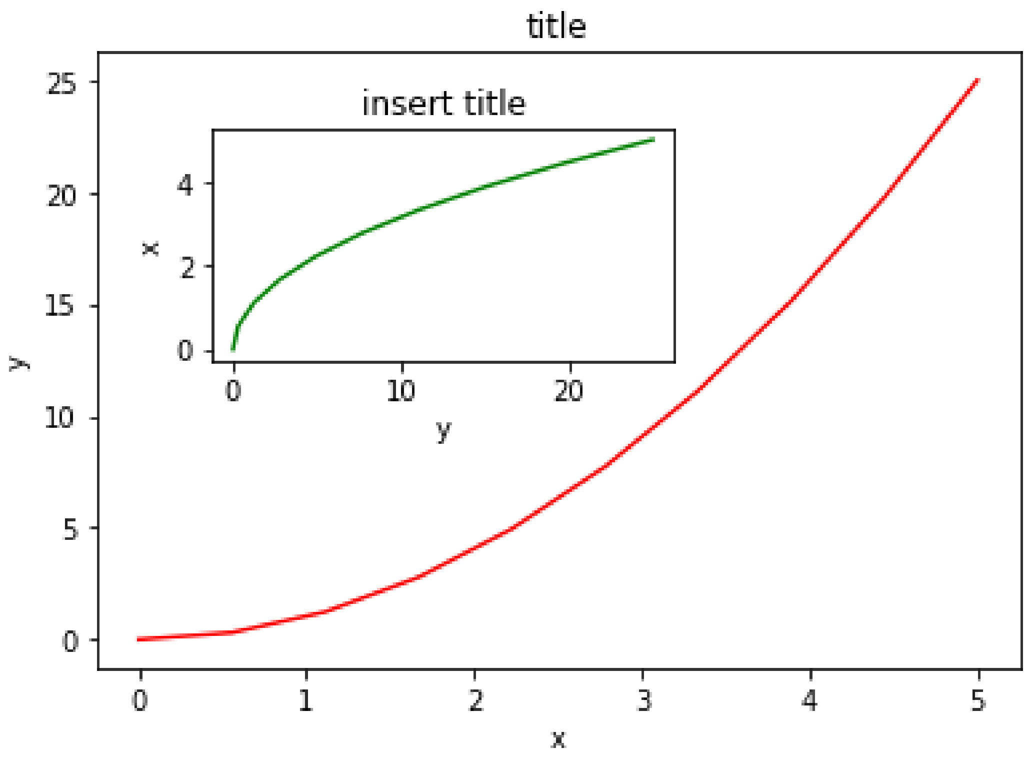
[501]:
```
fig = plt.figure()

axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range␣
 ↪0 to 1)

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```



Although a little bit more code is involved, the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

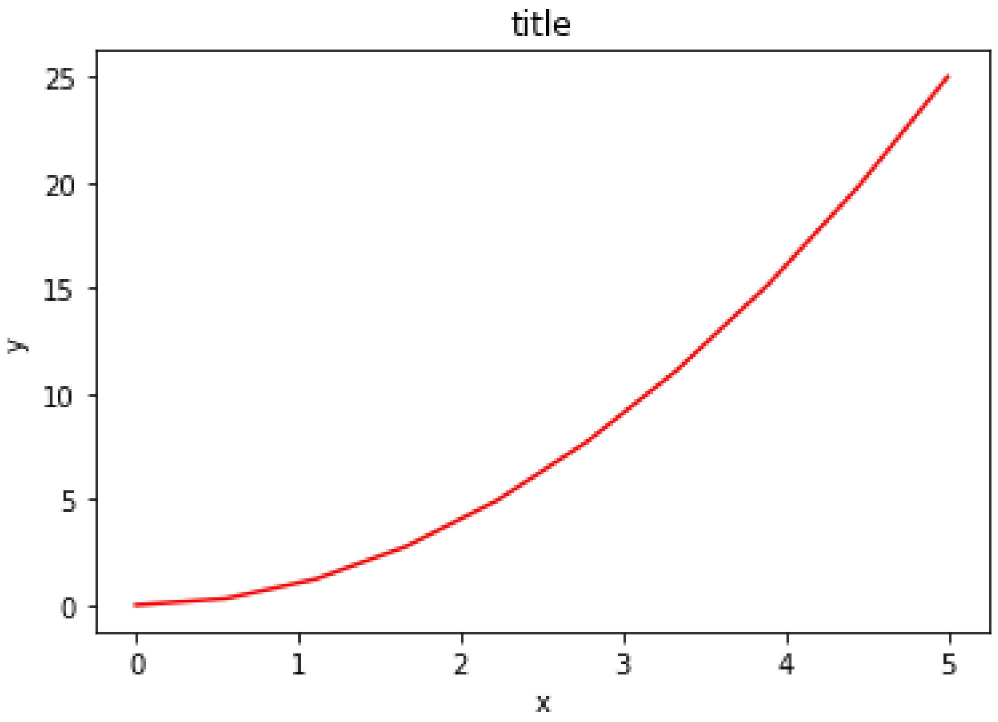
[502]:
```
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# main figure
axes1.plot(x, y, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('title')

# insert
axes2.plot(y, x, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
```

```
axes2.set_title('insert title');
```

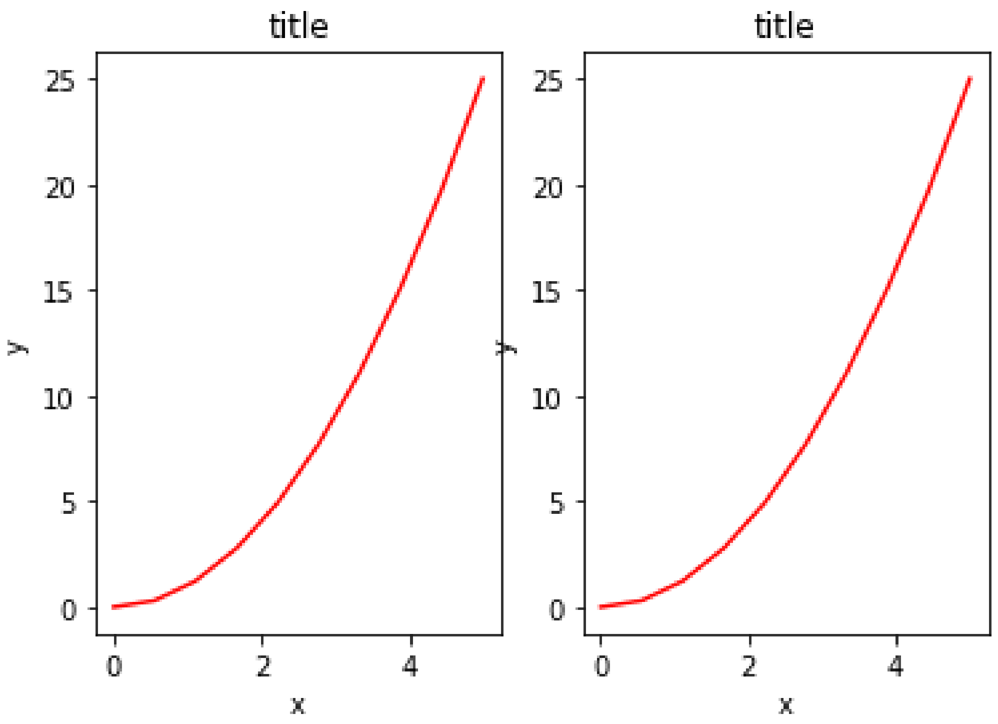


If we don't care about being explicit about where our plot axes are placed in the figure canvas, then we can use one of the many axis layout managers in matplotlib. My favorite is `subplots`, which can be used like this:

[503]:
```
fig, axes = plt.subplots()

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```



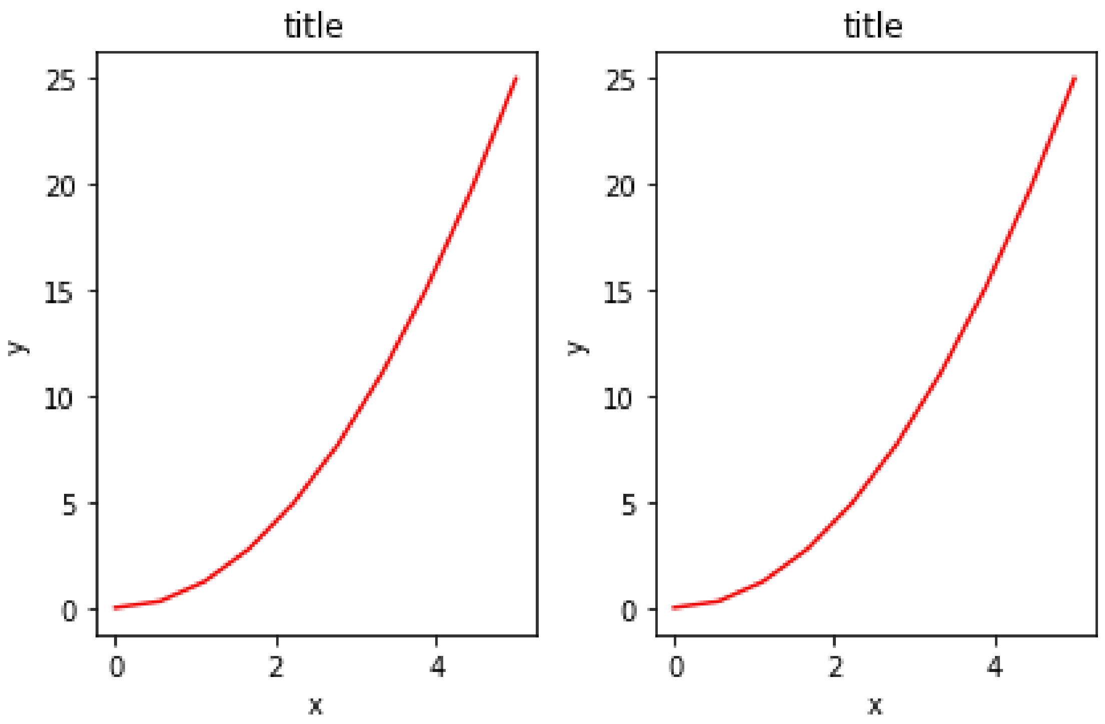
```
[504]: fig, axes = plt.subplots(nrows=1, ncols=2)

       for ax in axes:
           ax.plot(x, y, 'r')
           ax.set_xlabel('x')
           ax.set_ylabel('y')
           ax.set_title('title')
```



That was easy, but it isn't so pretty with overlapping figure axes and labels, right?

We can deal with that by using the `fig.tight_layout` method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

[505]:
```
fig, axes = plt.subplots(nrows=1, ncols=2)

for ax in axes:
    ax.plot(x, y, 'r')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

fig.tight_layout()
```



7.3.1 Figure size, aspect ratio and DPI

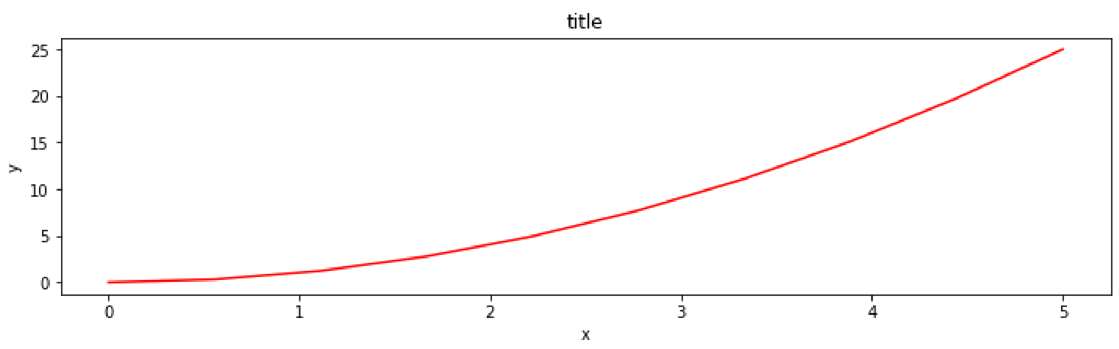
Matplotlib allows the aspect ratio, DPI and figure size to be specified when the `Figure` object is created, using the `figsize` and `dpi` keyword arguments. `figsize` is a tuple of the width and height of the figure in inches, and `dpi` is the dots-per-inch (pixel per inch). To create an 800x400 pixel, 100 dots-per-inch figure, we can do:

[506]:
```
fig = plt.figure(figsize=(8,4), dpi=100)
```

```
<Figure size 800x400 with 0 Axes>
```

The same arguments can also be passed to layout managers, such as the `subplots` function:

[507]:
```
fig, axes = plt.subplots(figsize=(12,3))

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```



7.3.2 Saving figures

To save a figure to a file we can use the `savefig` method in the `Figure` class:

[508]:
```
fig.savefig("filename.png")
```

Here we can also optionally specify the DPI and choose between different output formats:

[509]:
```
fig.savefig("filename.png", dpi=200)
```

What formats are available and which ones should be used for best quality? Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF. For scientific papers, I recommend using PDF whenever possible. (LaTeX documents compiled with `pdflatex` can include PDFs using the `includegraphics` command). In some cases, PGF can also be good alternative.

7.3.3 Legends, labels and titles

Now that we have covered the basics of how to create a figure canvas and add axes instances to the canvas, let's look at how decorate a figure with titles, axis labels, and legends.

Figure titles

A title can be added to each axis instance in a figure. To set the title, use the `set_title` method in the axes instance:

[510]:
```
ax.set_title("title");
```

Axis labels

Similarly, with the methods `set_xlabel` and `set_ylabel`, we can set the labels of the X and Y axes:

[511]:
```
ax.set_xlabel("x")
ax.set_ylabel("y");
```

Legends

Legends for curves in a figure can be added in two ways. One method is to use the `legend` method of the axis object and pass a list/tuple of legend texts for the previously defined curves:

[512]:
```
ax.legend(["curve1", "curve2", "curve3"]);
```

The method described above follows the MATLAB API. It is somewhat prone to errors and unflexible if curves are added to or removed from the figure (resulting in a wrongly labelled curve).

A better method is to use the `label="label text"` keyword argument when plots or other objects are added to the figure, and then using the `legend` method without arguments to add the legend to the figure:

[513]:
```
ax.plot(x, x**2, label="curve1")
ax.plot(x, x**3, label="curve2")
ax.legend();
```

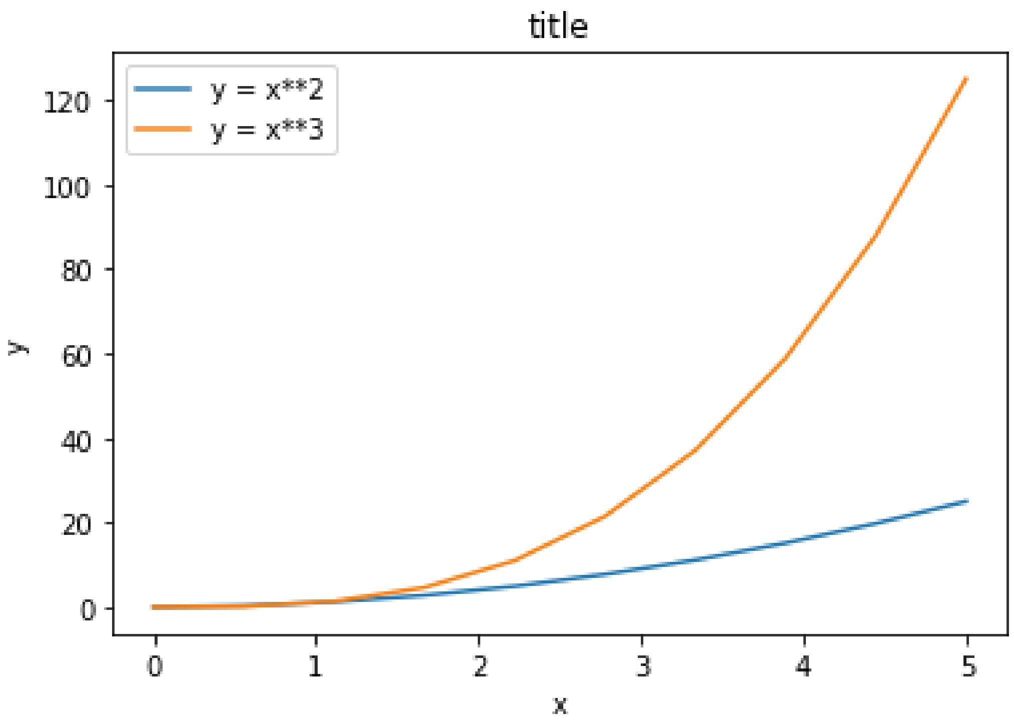
The advantage with this method is that if curves are added or removed from the figure, the legend is automatically updated accordingly.

The `legend` function takes an optional keyword argument `loc` that can be used to specify where in the figure the legend is to be drawn. The allowed values of `loc` are numerical codes for the various places the legend can be drawn. See http://matplotlib.org/users/legend_guide.html#legend-location for details. Some of the most common `loc` values are:

[514]:
```
ax.legend(loc=0) # let matplotlib decide the optimal location
ax.legend(loc=1) # upper right corner
ax.legend(loc=2) # upper left corner
ax.legend(loc=3) # lower left corner
ax.legend(loc=4); # lower right corner
# .. many more options are available
```

The following figure shows how to use the figure title, axis labels and legends described above:

[515]:
```
fig, ax = plt.subplots()

ax.plot(x, x**2, label="y = x**2")
ax.plot(x, x**3, label="y = x**3")
ax.legend(loc=2); # upper left corner
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('title');
```



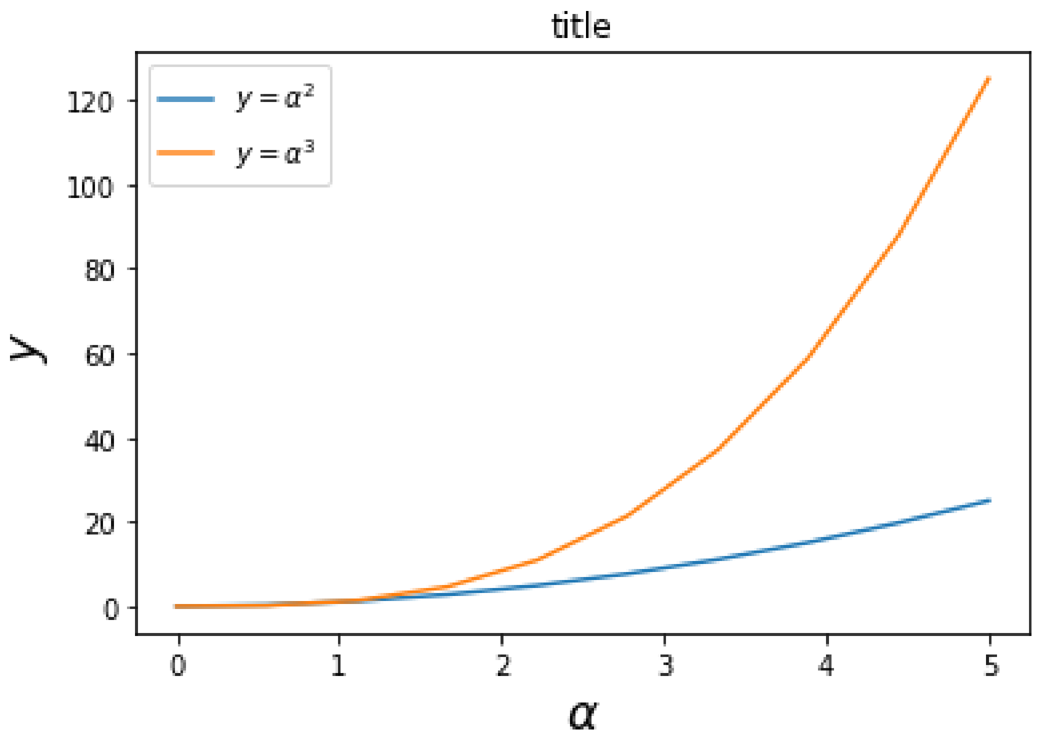
7.3.4 Formatting text: LaTeX, fontsize, font family

The figure above is functional, but it does not (yet) satisfy the criteria for a figure used in a publication. First and foremost, we need to have LaTeX formatted text, and second, we need to be able to adjust the font size to appear right in a publication.

Matplotlib has great support for LaTeX. All we need to do is to use dollar signs encapsulate LaTeX in any text (legend, title, label, etc.). For example, `"$y=x^3$"`.

But here we can run into a slightly subtle problem with LaTeX code and Python text strings. In LaTeX, we frequently use the backslash in commands, for example `\alpha` to produce the symbol α. But the backslash already has a meaning in Python strings (the escape code character). To avoid Python messing up our latex code, we need to use "raw" text strings. Raw text strings are prepended with an 'r', like `r"\alpha"` or `r'\alpha'` instead of `"\alpha"` or `'\alpha'`:

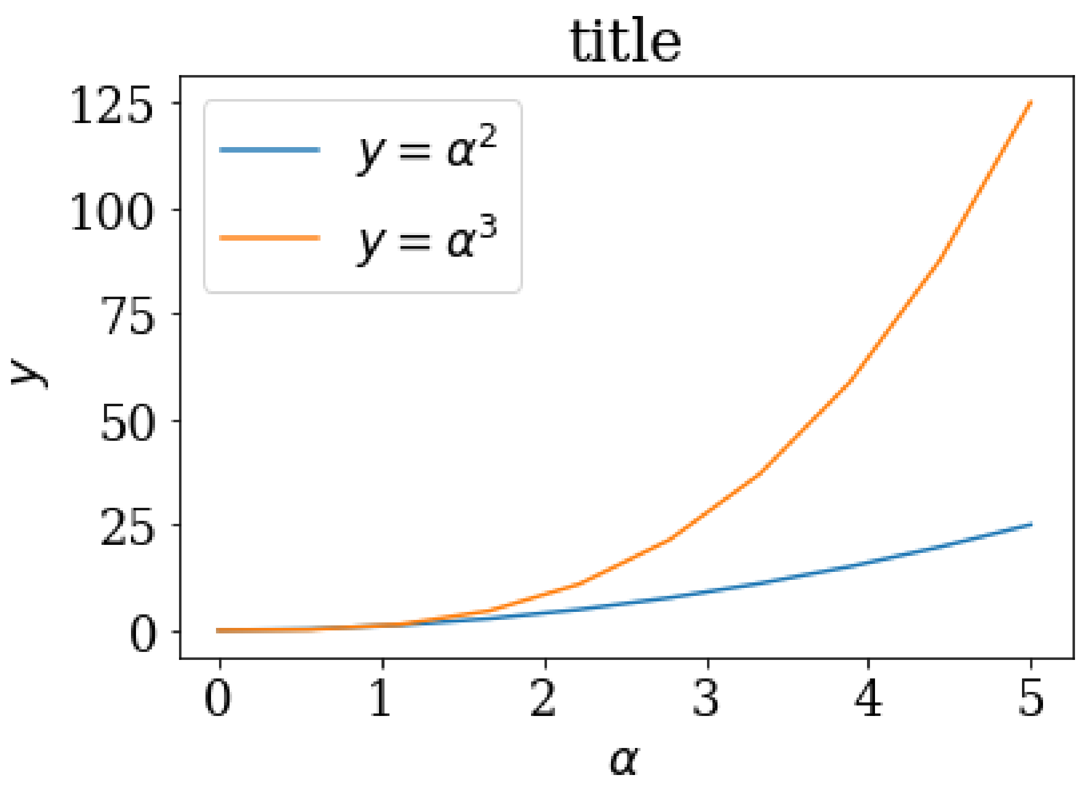
[516]:
```
fig, ax = plt.subplots()

ax.plot(x, x**2, label=r"$y = \alpha^2$")
ax.plot(x, x**3, label=r"$y = \alpha^3$")
ax.legend(loc=2) # upper left corner
ax.set_xlabel(r'$\alpha$', fontsize=18)
ax.set_ylabel(r'$y$', fontsize=18)
ax.set_title('title');
```



We can also change the global font size and font family, which applies to all text elements in a figure (tick labels, axis labels and titles, legends, etc.):

```
[517]: # Update the matplotlib configuration parameters:
       matplotlib.rcParams.update({'font.size': 18, 'font.family': 'serif'})
```

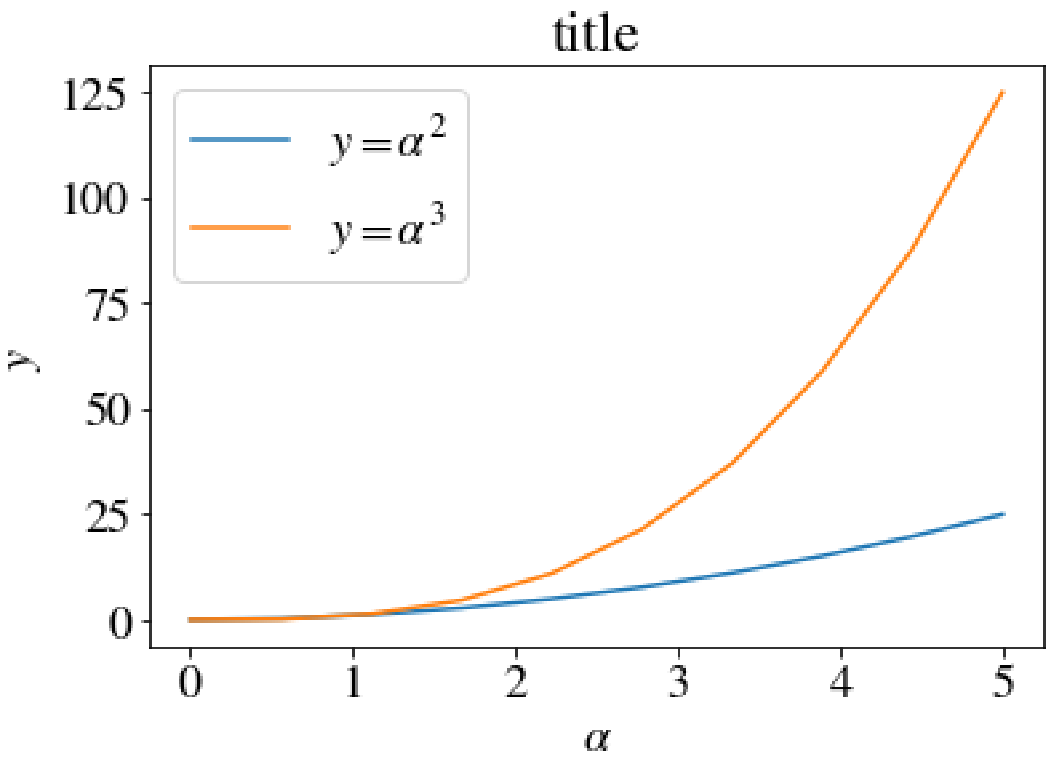
```
[518]: fig, ax = plt.subplots()

       ax.plot(x, x**2, label=r"$y = \alpha^2$")
       ax.plot(x, x**3, label=r"$y = \alpha^3$")
       ax.legend(loc=2) # upper left corner
       ax.set_xlabel(r'$\alpha$')
       ax.set_ylabel(r'$y$')
       ax.set_title('title');
```



A good choice of global fonts are the STIX fonts:

```
[519]: # Update the matplotlib configuration parameters:
matplotlib.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral',␣
 ↪'mathtext.fontset': 'stix'})
```

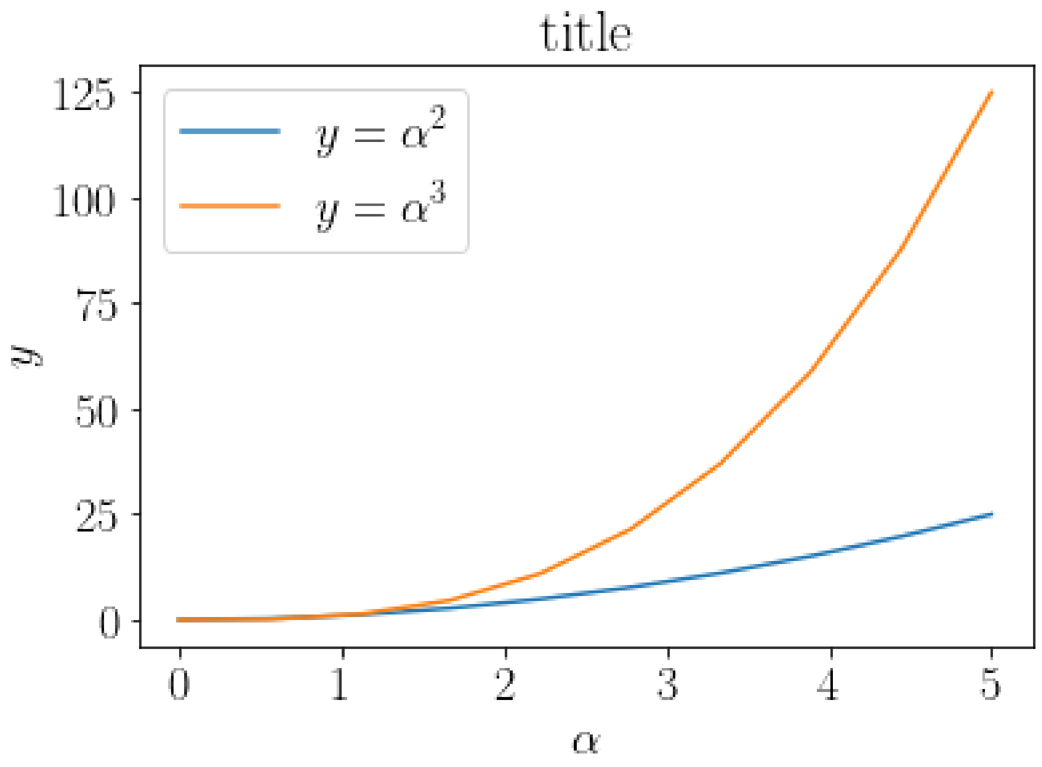
```
[520]: fig, ax = plt.subplots()

ax.plot(x, x**2, label=r"$y = \alpha^2$")
ax.plot(x, x**3, label=r"$y = \alpha^3$")
ax.legend(loc=2) # upper left corner
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel(r'$y$')
ax.set_title('title');
```



Or, alternatively, we can request that matplotlib uses LaTeX to render the text elements in the figure:

```
[521]: matplotlib.rcParams.update({'font.size': 18, 'text.usetex': True})
```

```
[522]: fig, ax = plt.subplots()

ax.plot(x, x**2, label=r"$y = \alpha^2$")
ax.plot(x, x**3, label=r"$y = \alpha^3$")
ax.legend(loc=2) # upper left corner
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel(r'$y$')
ax.set_title('title');
```



```
[523]: # restore
matplotlib.rcParams.update({'font.size': 12, 'font.family': 'sans', 'text.
 ↪usetex': False})
```

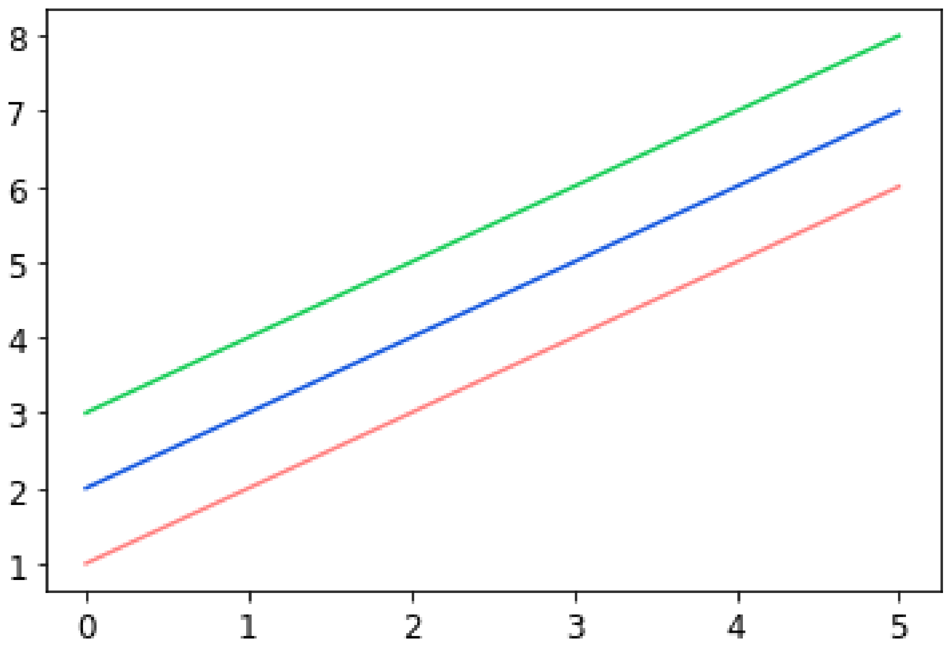
7.3.5 Setting colors, linewidths, linetypes

Colors With matplotlib, we can define the colors of lines and other graphical elements in a number of ways. First of all, we can use the MATLAB-like syntax where `'b'` means blue, `'g'` means green, etc. The MATLAB API for selecting line styles are also supported: where, for example, 'b.-' means a blue line with dots:

```
[524]: # MATLAB style line color and style
ax.plot(x, x**2, 'b.-') # blue line with dots
ax.plot(x, x**3, 'g--'); # green dashed line
```

We can also define colors by their names or RGB hex codes and optionally provide an alpha value using the `color` and `alpha` keyword arguments:

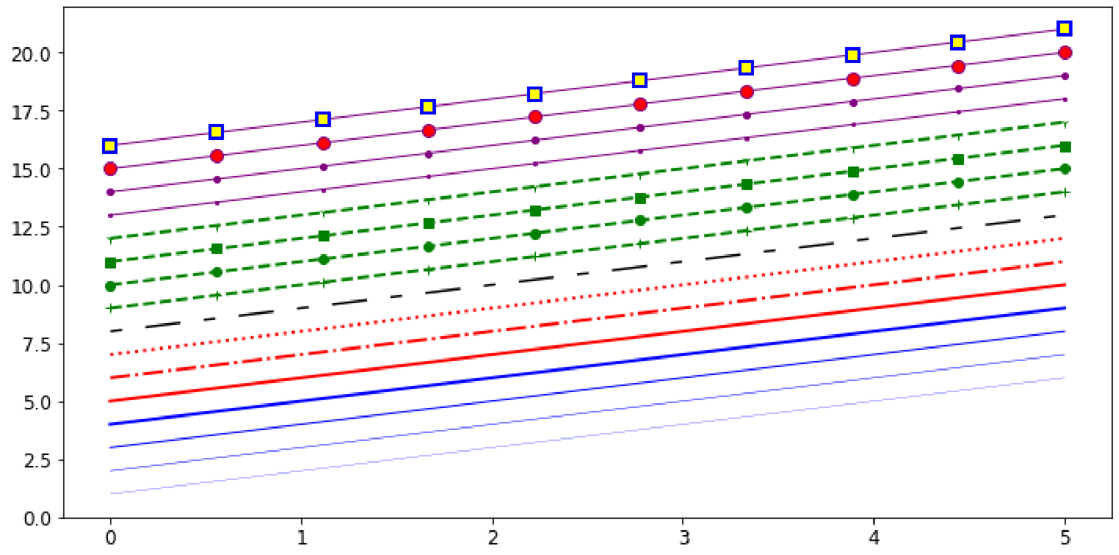
```
[525]: fig, ax = plt.subplots()

ax.plot(x, x+1, color="red", alpha=0.5) # half-transparant red
ax.plot(x, x+2, color="#1155dd")        # RGB hex code for a bluish color
ax.plot(x, x+3, color="#15cc55") ;       # RGB hex code for a greenish color
```



Line and marker styles To change the line width, we can use the `linewidth` or `lw` keyword argument. The line style can be selected using the `linestyle` or `ls` keyword arguments:

[526]:
```
fig, ax = plt.subplots(figsize=(12,6))

ax.plot(x, x+1, color="blue", linewidth=0.25)
ax.plot(x, x+2, color="blue", linewidth=0.50)
ax.plot(x, x+3, color="blue", linewidth=1.00)
ax.plot(x, x+4, color="blue", linewidth=2.00)

# possible linestype options '-', '--', '-.', ':', 'steps'
ax.plot(x, x+5, color="red", lw=2, linestyle='-')
ax.plot(x, x+6, color="red", lw=2, ls='-.')
ax.plot(x, x+7, color="red", lw=2, ls=':')

# custom dash
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2',␣
 ↪'3', '4', ...
ax.plot(x, x+ 9, color="green", lw=2, ls='--', marker='+')
ax.plot(x, x+10, color="green", lw=2, ls='--', marker='o')
ax.plot(x, x+11, color="green", lw=2, ls='--', marker='s')
ax.plot(x, x+12, color="green", lw=2, ls='--', marker='1')
```

```
# marker size and color
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8,␣
 ↪markerfacecolor="red")
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=2, markeredgecolor="blue");
```



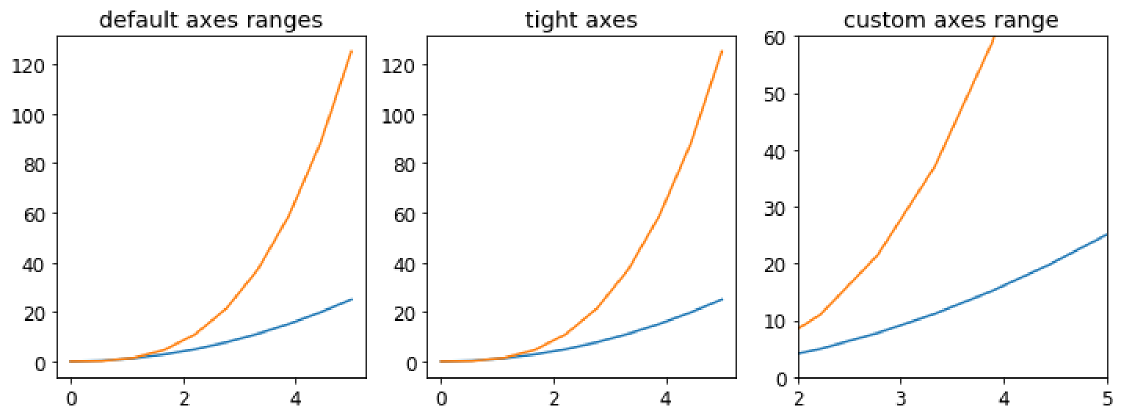
7.3.6 Control over axis appearance

The appearance of the axes is an important aspect of a figure that we often need to modify to make a publication quality graphics. We need to be able to control where the ticks and labels are placed, modify the font size and possibly the labels used on the axes. In this section we will look at controling those properties in a matplotlib figure.

Plot range The first thing we might want to configure is the ranges of the axes. We can do this using the `set_ylim` and `set_xlim` methods in the axis object, or `axis('tight')` for automatrically getting "tightly fitted" axes ranges:

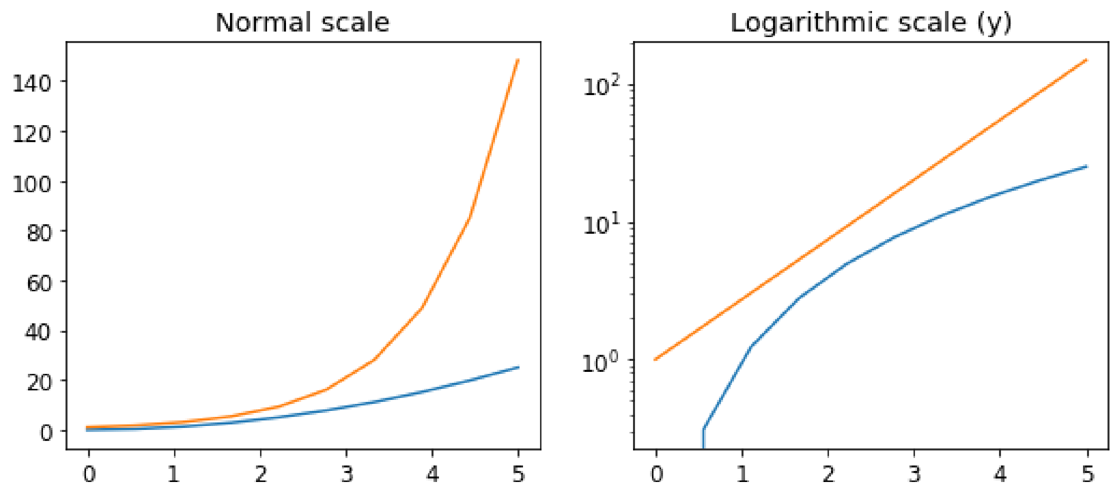
[527]:
```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].plot(x, x**2, x, x**3)
axes[0].set_title("default axes ranges")

axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")
```

```
axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range");
```



Logarithmic scale It is also possible to set a logarithmic scale for one or both axes. This functionality is in fact only one application of a more general transformation system in Matplotlib. Each of the axes' scales are set seperately using `set_xscale` and `set_yscale` methods which accept one parameter (with the value "log" in this case):

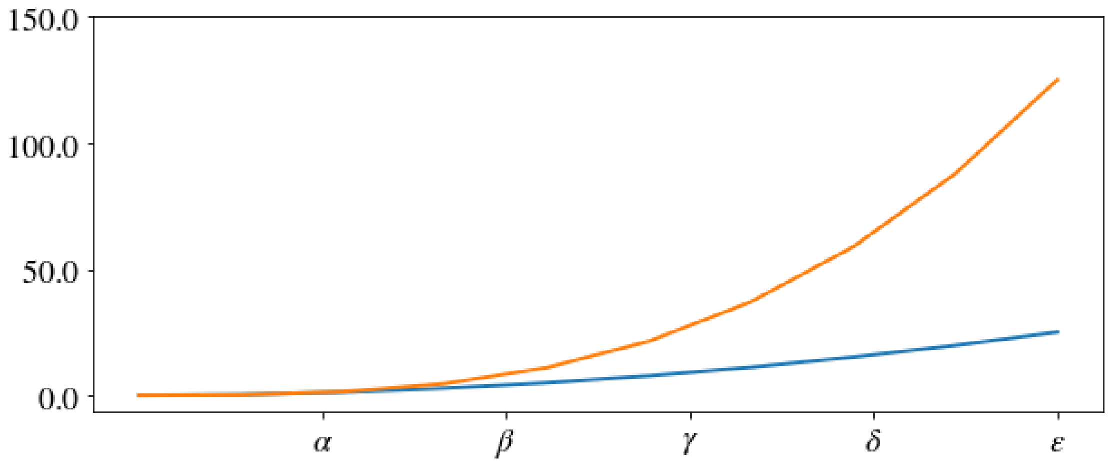
[528]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,4))

axes[0].plot(x, x**2, x, np.exp(x))
axes[0].set_title("Normal scale")

axes[1].plot(x, x**2, x, np.exp(x))
axes[1].set_yscale("log")
axes[1].set_title("Logarithmic scale (y)");
```



7.3.7 Placement of ticks and custom tick labels

We can explicitly determine where we want the axis ticks with `set_xticks` and `set_yticks`, which both take a list of values for where on the axis the ticks are to be placed. We can also use the `set_xticklabels` and `set_yticklabels` methods to provide a list of custom text labels for each tick location:

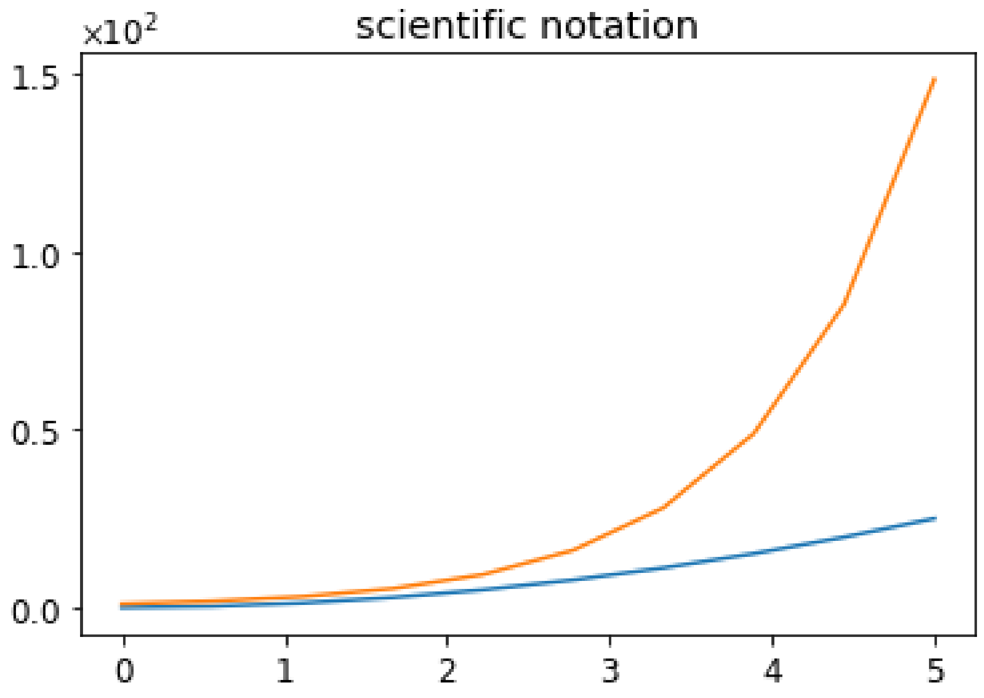
[529]:
```
fig, ax = plt.subplots(figsize=(10, 4))

ax.plot(x, x**2, x, x**3, lw=2)

ax.set_xticks([1, 2, 3, 4, 5])
ax.set_xticklabels([r'$\alpha$', r'$\beta$', r'$\gamma$', r'$\delta$',␣
 ↪r'$\epsilon$'], fontsize=18)

yticks = [0, 50, 100, 150]
ax.set_yticks(yticks)
ax.set_yticklabels(["$%.1f$" % y for y in yticks], fontsize=18); # use LaTeX␣
 ↪formatted labels
```



There are a number of more advanced methods for controlling major and minor tick placement in matplotlib figures, such as automatic placement according to different policies. See http://matplotlib.org/api/ticker_api.html for details.

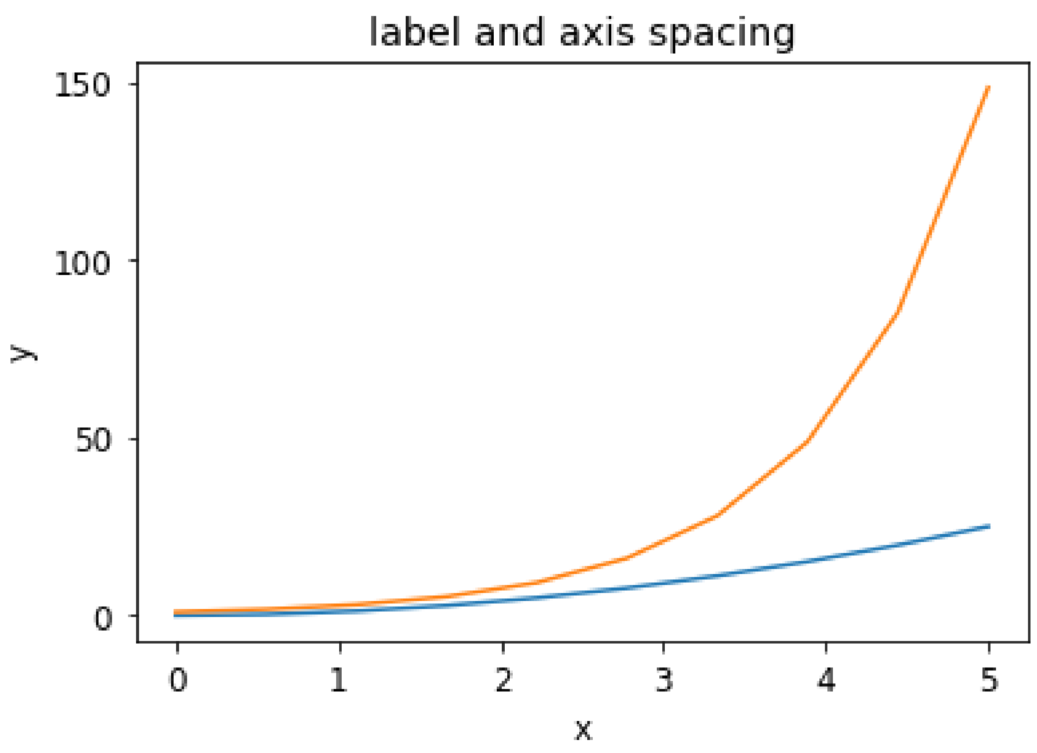
Scientific notation With large numbers on axes, it is often better use scientific notation:

[530]:
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_title("scientific notation")

ax.set_yticks([0, 50, 100, 150])

from matplotlib import ticker
formatter = ticker.ScalarFormatter(useMathText=True)
formatter.set_scientific(True)
formatter.set_powerlimits((-1,1))
ax.yaxis.set_major_formatter(formatter)
```



7.3.8 Axis number and axis label spacing

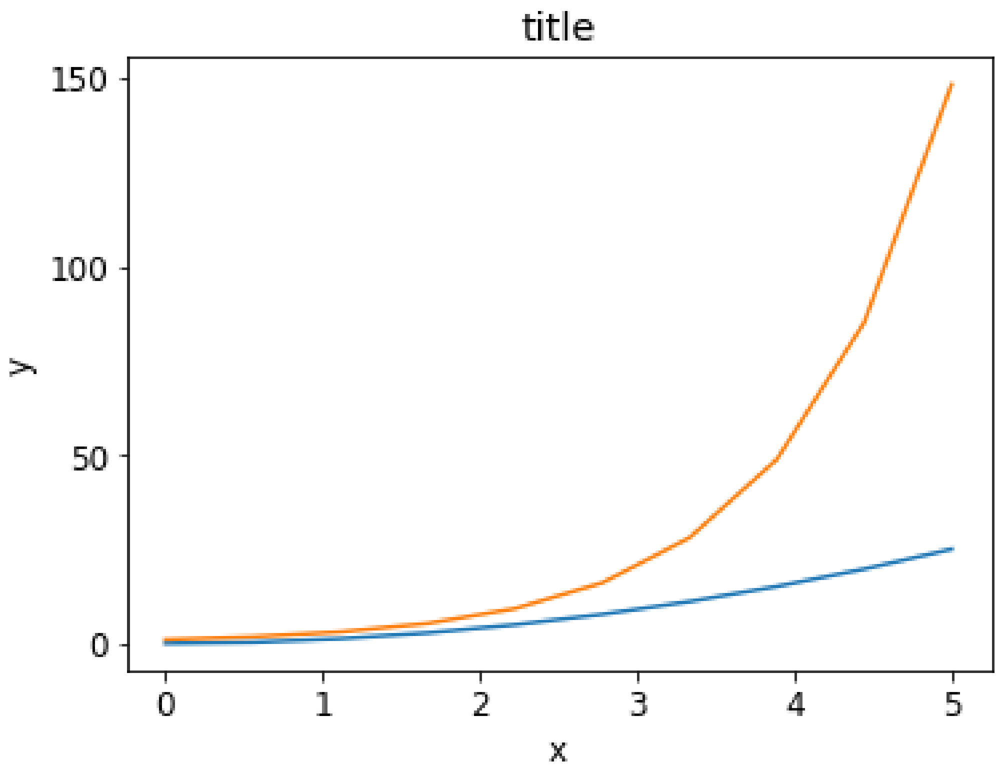
[531]:
```
# distance between x and y axis and the numbers on the axes
matplotlib.rcParams['xtick.major.pad'] = 5
matplotlib.rcParams['ytick.major.pad'] = 5

fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("label and axis spacing")

# padding between axis label and axis numbers
ax.xaxis.labelpad = 5
ax.yaxis.labelpad = 5

ax.set_xlabel("x")
ax.set_ylabel("y");
```



[532]:
```
# restore defaults
matplotlib.rcParams['xtick.major.pad'] = 3
matplotlib.rcParams['ytick.major.pad'] = 3
```

Axis position adjustments Unfortunately, when saving figures the labels are sometimes clipped, and it can be necessary to adjust the positions of axes a little bit. This can be done using `subplots_adjust`:

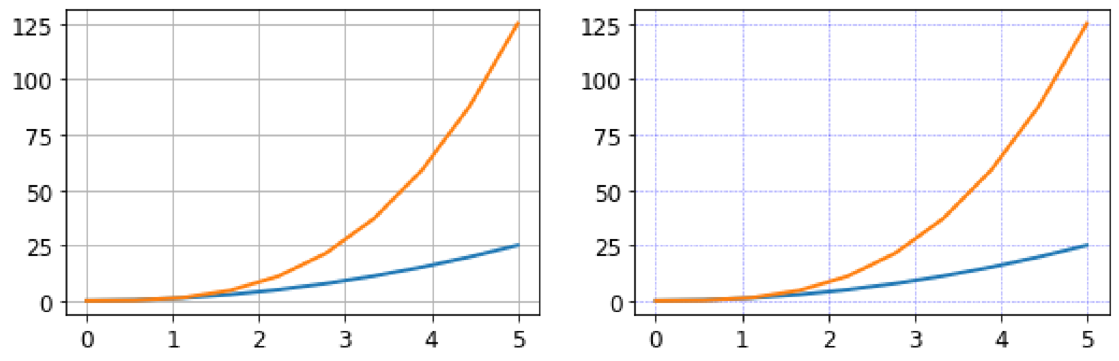
[533]:
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("title")
ax.set_xlabel("x")
ax.set_ylabel("y")

fig.subplots_adjust(left=0.15, right=.9, bottom=0.1, top=0.9);
```



7.3.9 Axis grid

With the `grid` method in the axis object, we can turn on and off grid lines. We can also customize the appearance of the grid lines using the same keyword arguments as the `plot` function:

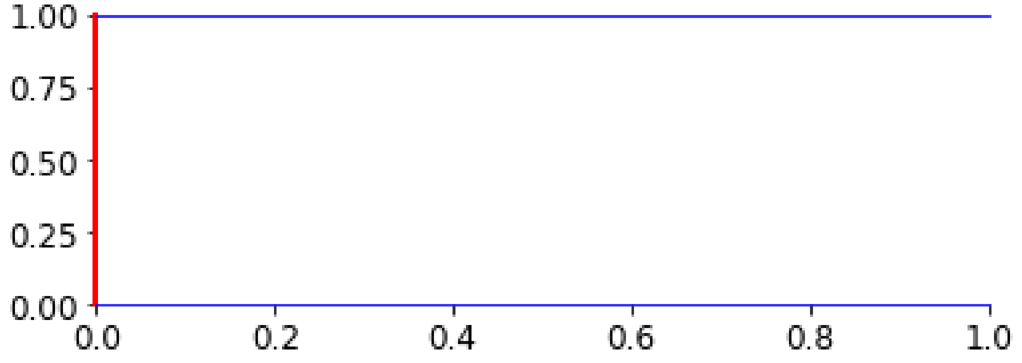
[534]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,3))

# default grid appearance
axes[0].plot(x, x**2, x, x**3, lw=2)
axes[0].grid(True)

# custom grid appearance
axes[1].plot(x, x**2, x, x**3, lw=2)
axes[1].grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
```



7.3.10 Axis spines

We can also change the properties of axis spines:

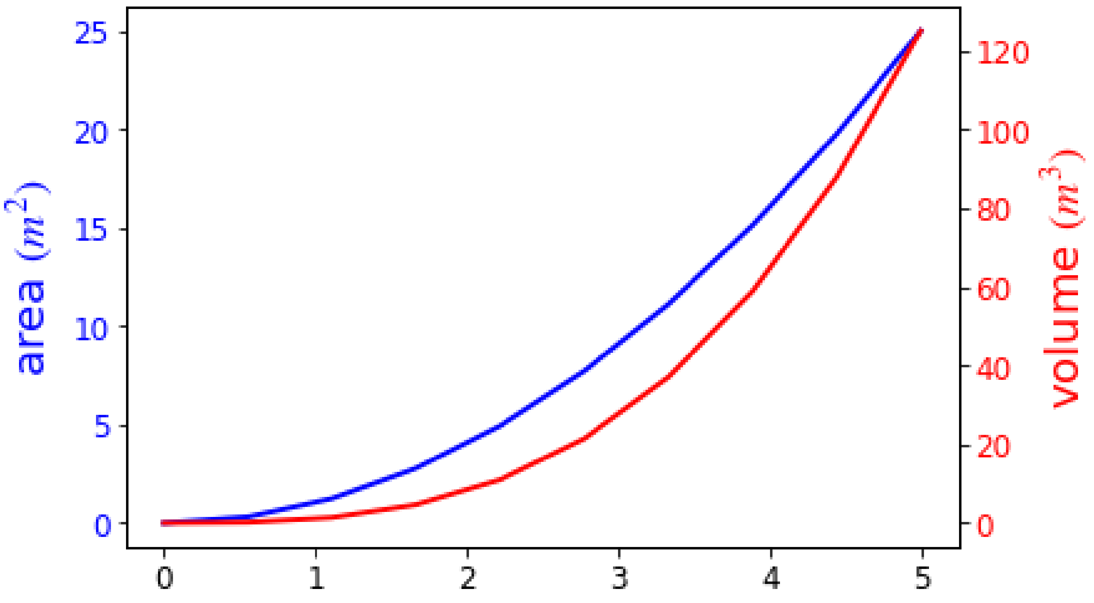
```
[535]: fig, ax = plt.subplots(figsize=(6,2))

ax.spines['bottom'].set_color('blue')
ax.spines['top'].set_color('blue')

ax.spines['left'].set_color('red')
ax.spines['left'].set_linewidth(2)

# turn off axis spine to the right
ax.spines['right'].set_color("none")
ax.yaxis.tick_left() # only ticks on the left side
```



7.3.11 Twin axes

Sometimes it is useful to have dual x or y axes in a figure; for example, when plotting curves with different units together. Matplotlib supports this with the `twinx` and `twiny` functions:

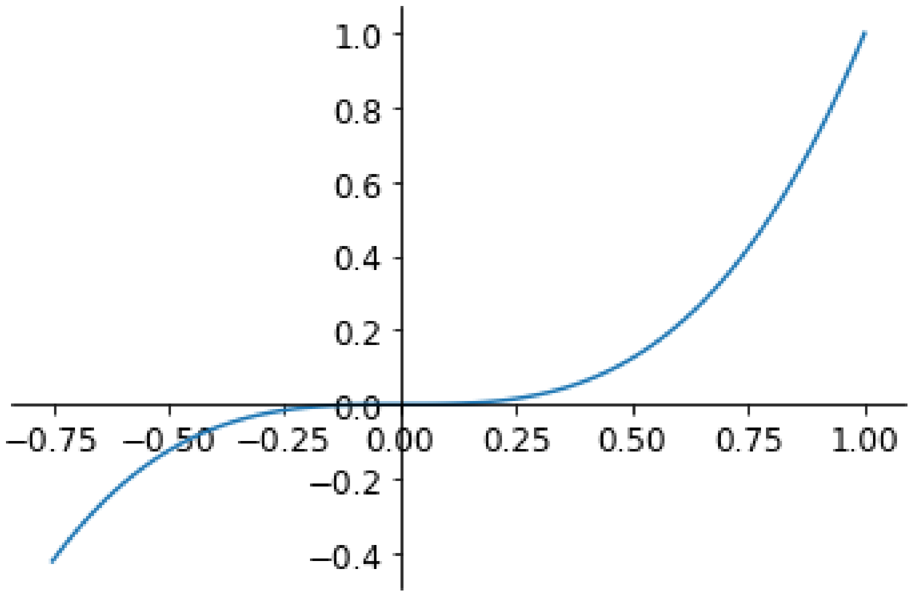
[536]:
```
fig, ax1 = plt.subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_ylabel(r"area $(m^2)$", fontsize=18, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, x**3, lw=2, color="red")
ax2.set_ylabel(r"volume $(m^3)$", fontsize=18, color="red")
for label in ax2.get_yticklabels():
    label.set_color("red")
```



7.3.12 Axes where x and y is zero

[537]:
```
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))   # set position of y spine to y=0
```

```
xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```



8 Problem (20 minutes)

Temperature data recorded at Heathrow Airprot is located in the file *heathrow.dat* from 1948 to 2018. The format of the data is give by

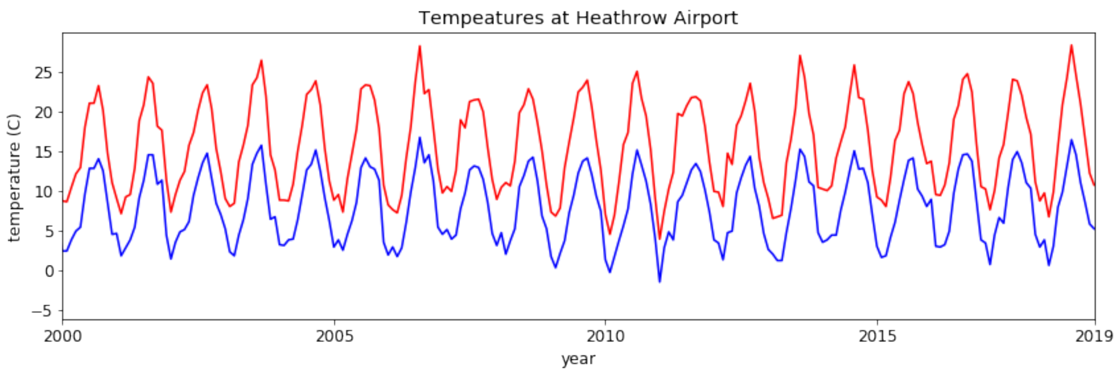
[538]:
```
!head ./data/heathrow_temp_data.dat
```

```
#  yyyy  mm   tmax    tmin     af    rain    sun
#             degC    degC    days     mm   hours
   1948   1    8.9     3.3    ---    85.0    ---
   1948   2    7.9     2.2    ---    26.0    ---
   1948   3   14.2     3.8    ---    14.0    ---
   1948   4   15.4     5.1    ---    35.0    ---
   1948   5   18.1     6.9    ---    57.0    ---
   1948   6   19.1    10.3    ---    67.0    ---
   1948   7   21.7    12.0    ---    21.0    ---
   1948   8   20.8    11.7    ---    67.0    ---
```

Instructions: Produce a plot of the min/max temperatures at Heathrow airport from the year 2000 to 2019. Plot the maximum temperature in red, and the minimum temperature in blue. Ensure to add axis labels and a title.

[539]:
```
# read in data
data = genfromtxt('./data/heathrow_temp_data.dat')
```

```
# create figure
fig, ax = plt.subplots(figsize=(14,4))
ax.plot(data[:,0]+data[:,1]/12.0, data[:,2], 'r-')
ax.plot(data[:,0]+data[:,1]/12.0, data[:,3], 'b-')
# set title and axes labels
ax.axis('tight')
ax.set_title('Tempeatures at Heathrow Airport')
ax.set_xlabel('year')
ax.set_ylabel('temperature (C)');
# adjust x-axis limits to be between 2000 and 2019
ax.set_xlim((2000,2019))
ax.set_xticks([2000, 2005, 2010, 2015,2019]);
```

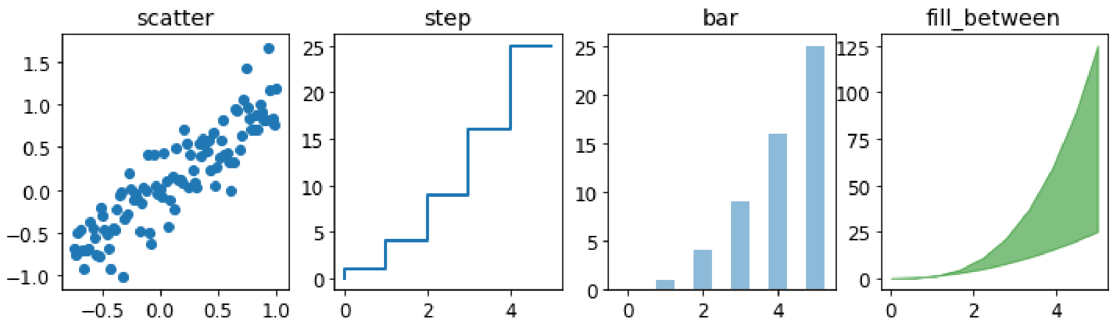


8.1 4.3 2D plotting styles

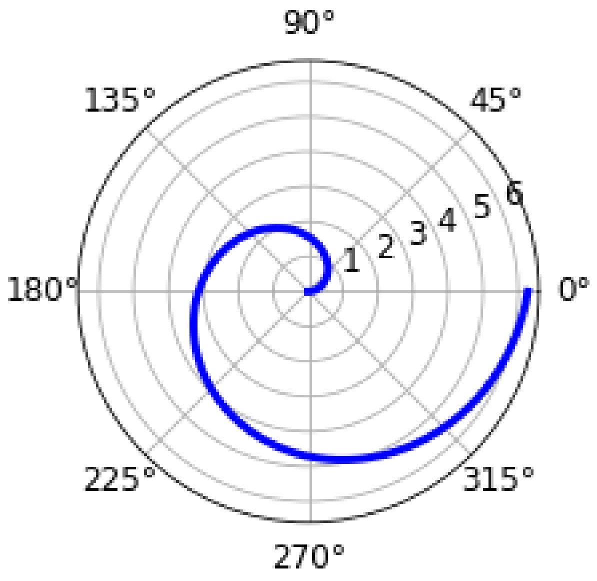
In addition to the regular `plot` method, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of available plot types: http://matplotlib.org/gallery.html. Some of the more useful ones are show below:

[540]:
```
n = np.array([0,1,2,3,4,5])
```

[541]:
```
fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(xx, xx + 0.25*np.random.randn(len(xx)))
axes[0].set_title("scatter")

axes[1].step(n, n**2, lw=2)
axes[1].set_title("step")

axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
axes[2].set_title("bar")

axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
axes[3].set_title("fill_between");
```

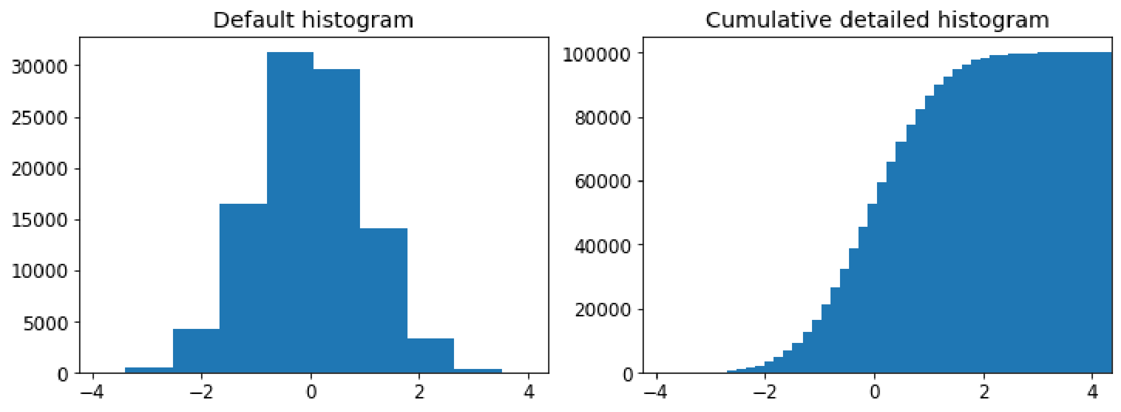


[542]:
```
# polar plot using add_axes and polar projection
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
t = np.linspace(0, 2 * np.pi, 100)
ax.plot(t, t, color='blue', lw=3);
```



[543]:
```
# A histogram
n = np.random.randn(100000)
fig, axes = plt.subplots(1, 2, figsize=(12,4))

axes[0].hist(n)
axes[0].set_title("Default histogram")
axes[0].set_xlim((min(n), max(n)))

axes[1].hist(n, cumulative=True, bins=50)
```

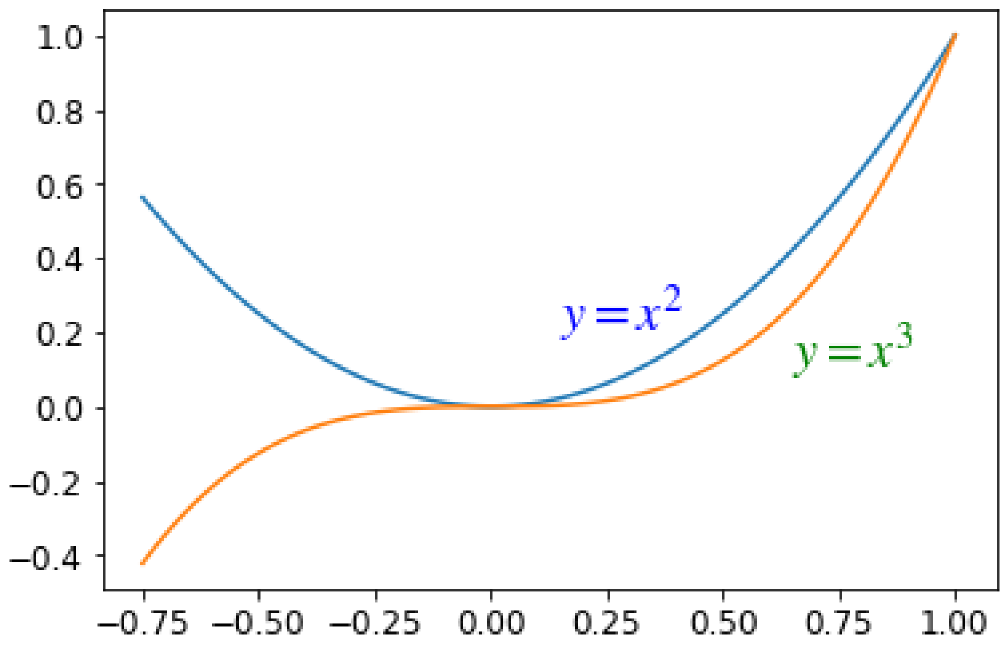
```
axes[1].set_title("Cumulative detailed histogram")
axes[1].set_xlim((min(n), max(n)));
```



8.1.1 Text annotation

Annotating text in matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles:

[544]:
```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")
ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green");
```

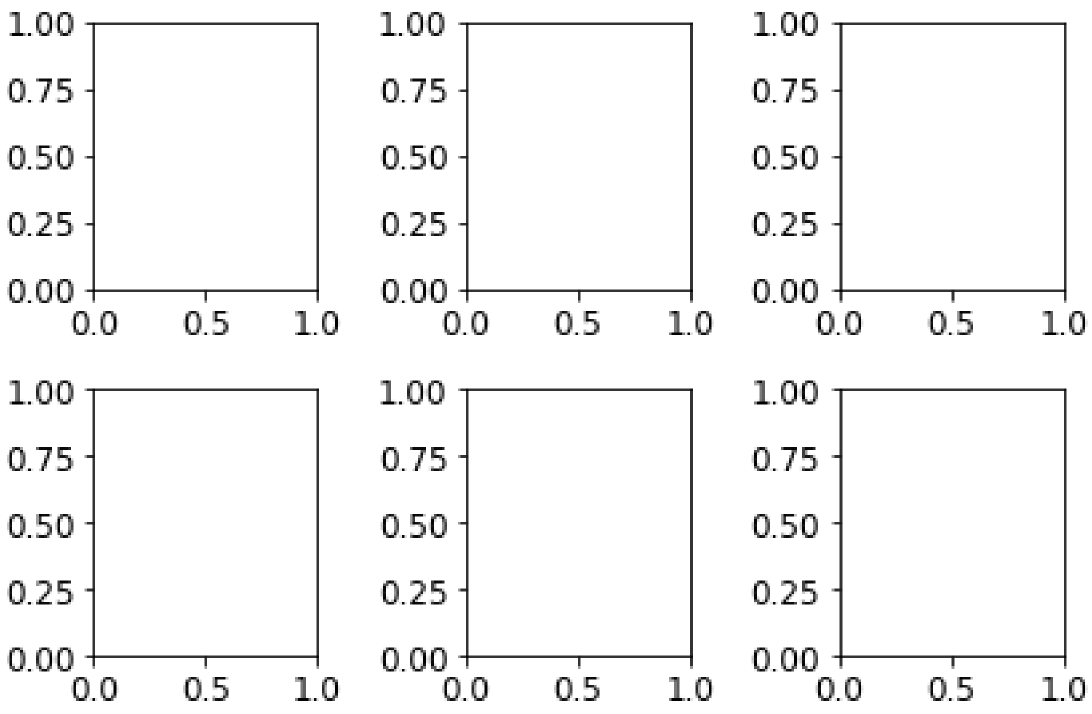


8.1.2 Figures with multiple subplots and insets

Axes can be added to a matplotlib Figure canvas manually using `fig.add_axes` or using a sub-figure layout manager such as `subplots`, `subplot2grid`, or `gridspec`:

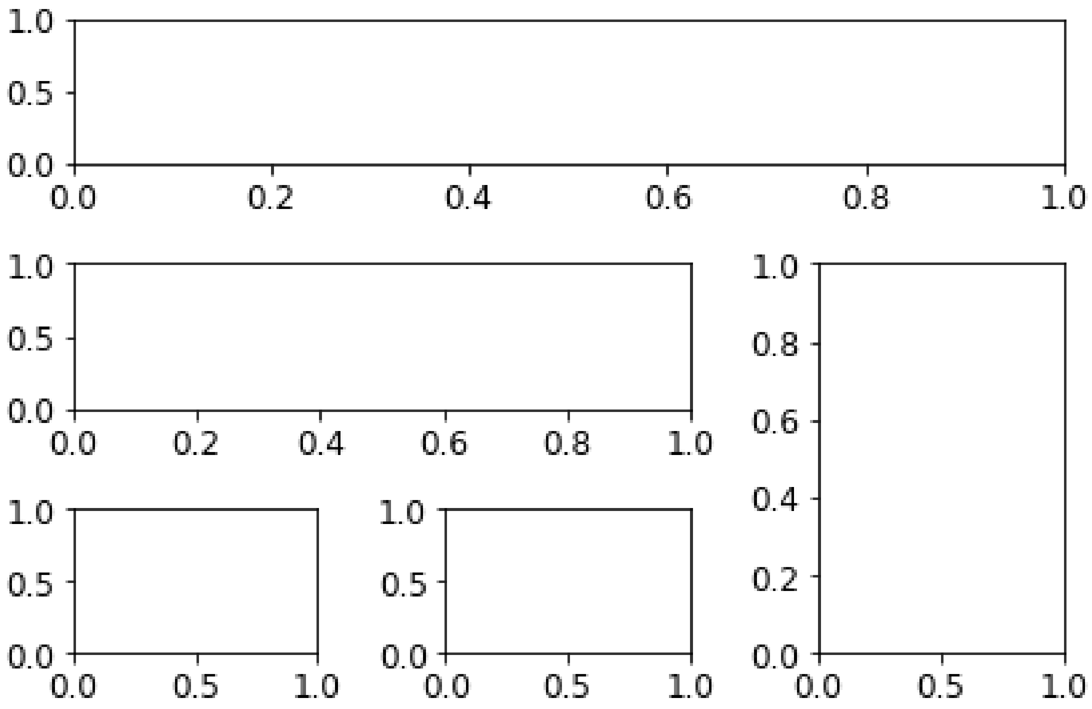
subplots

[545]:
```
fig, ax = plt.subplots(2, 3)
fig.tight_layout()
```



subplot2grid

```
[546]: fig = plt.figure()
       ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
       ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
       ax3 = plt.subplot2grid((3,3), (1,2), rowspan=2)
       ax4 = plt.subplot2grid((3,3), (2,0))
       ax5 = plt.subplot2grid((3,3), (2,1))
       fig.tight_layout()
```



8.1.3 Colormap and contour figures

Colormaps and contour figures are useful for plotting functions of two variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps. It is relatively straightforward to define custom colormaps. For a list of pre-defined colormaps, see: http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

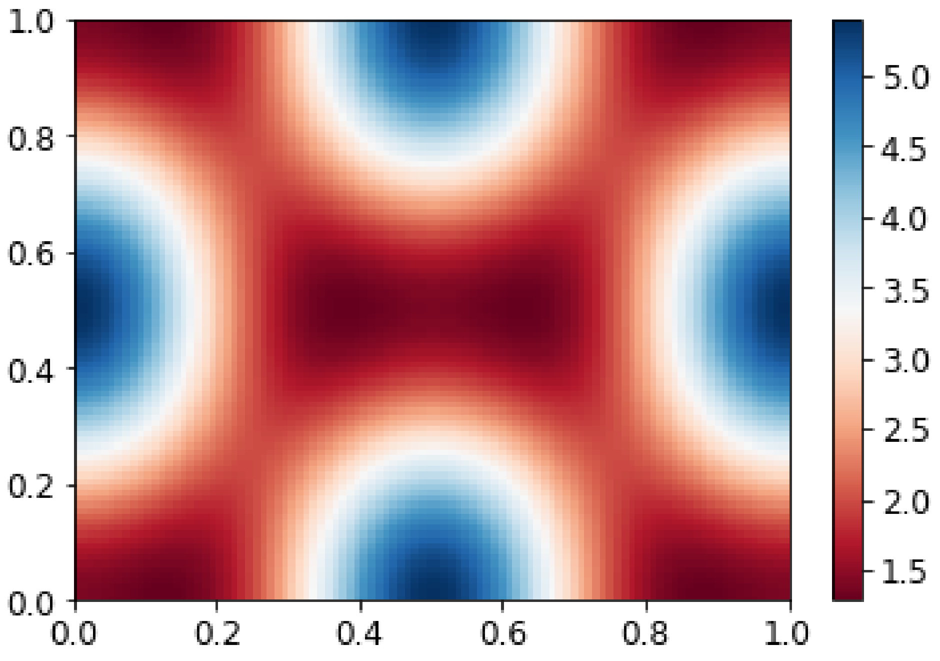
[547]:
```
alpha = 0.7
phi_ext = 2 * np.pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p) * np.cos(phi_m) - alpha * np.
 ↪cos(phi_ext - 2*phi_p)
```

[548]:
```
phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)
X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

pcolor

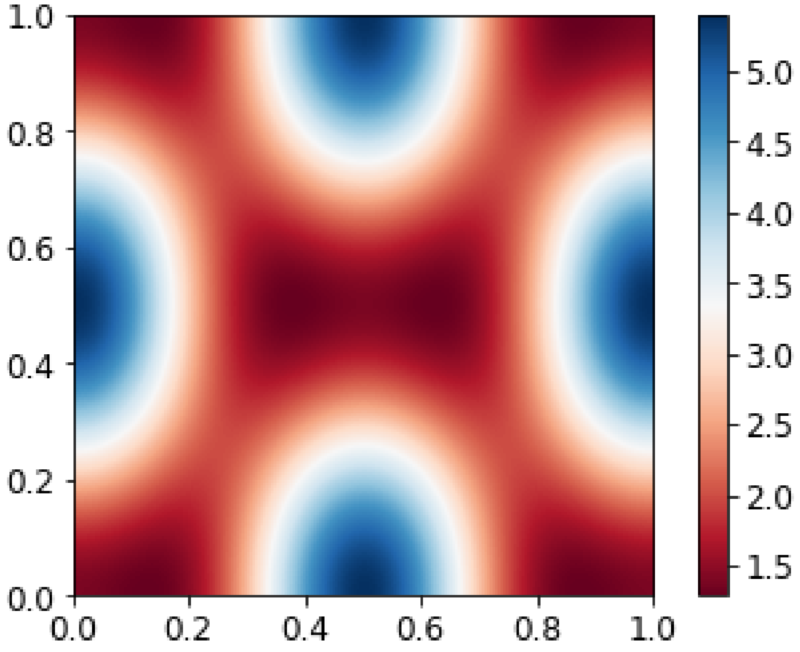
[549]:
```
fig, ax = plt.subplots()
```

```
p = ax.pcolor(X/(2*np.pi), Y/(2*np.pi), Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).
↪min(), vmax=abs(Z).max())
cb = fig.colorbar(p, ax=ax)
```



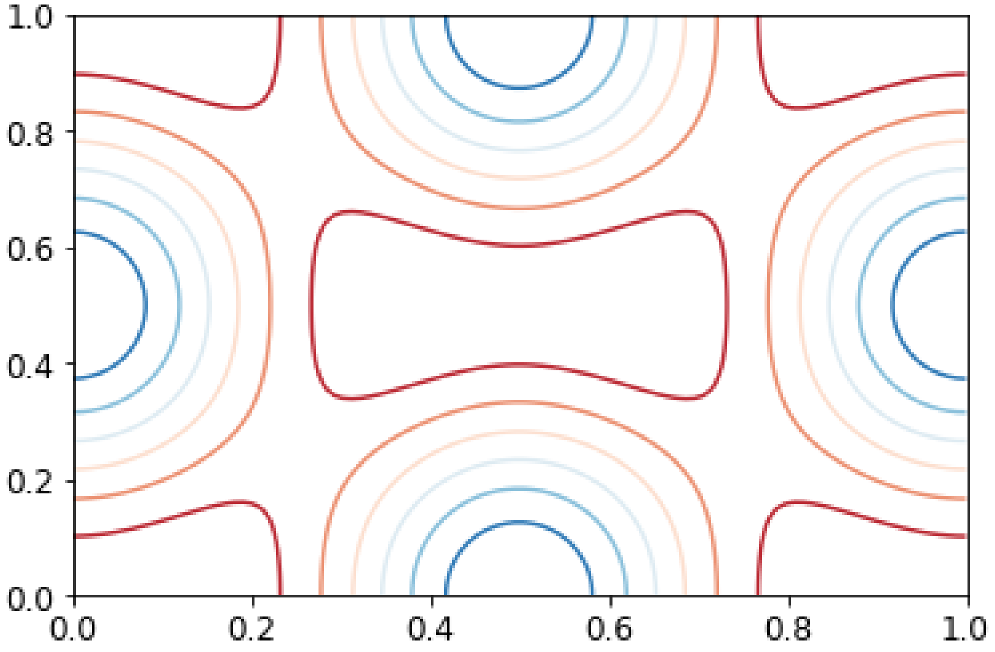
imshow

[550]:
```
fig, ax = plt.subplots()

im = ax.imshow(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).
↪max(), extent=[0, 1, 0, 1])
im.set_interpolation('bilinear')

cb = fig.colorbar(im, ax=ax)
```



contour

```
[551]: fig, ax = plt.subplots()

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).
 ↪max(), extent=[0, 1, 0, 1])
```



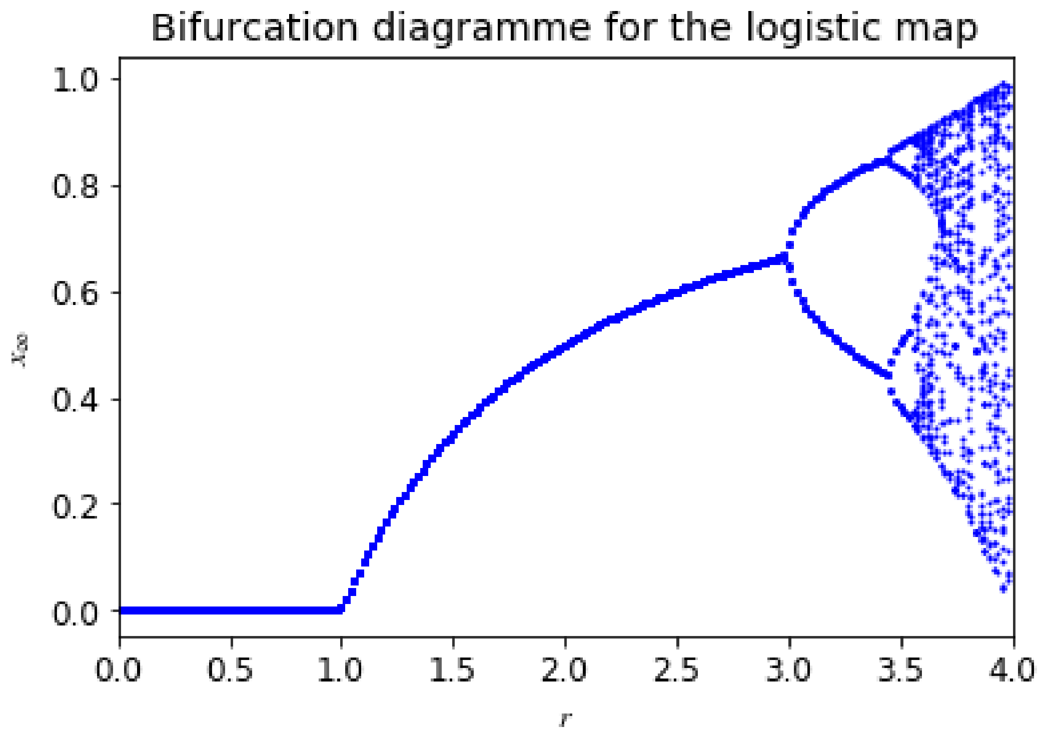
9 Problem (20 minutes)

The logistic map is a recursive relation for $x_n \in [0, 1]$ and provides an example for chaotic behaviour arising from a nonlinear dynamical system. The parameter $r \in [0, 4]$ is the bifurcation parameter and is responsbible for the long-time behaviour of x_n. The logistic map is given by

$$x_{n+1} = r x_n (1 - x_n) . \tag{7}$$

Instructions: Produce a bifrucation diagramme for the logistic map by plotting the birfurcation parameter r versus the values of $\lim_{n\to\infty} x_n$.

[552]:
```python
dr=0.02
N=200
fig, ax = plt.subplots()

for i in range(0,N):
    x0=0.5
    r = i*dr
    for j in range(1,100):
        x0 = r*x0*(1-x0)
    for j in range(1,40):
        x0 = r*x0*(1-x0)
        ax.plot(r,x0,'bo',markersize=1)

ax.set_xlim((0, 4))
ax.set_xlabel(r'$r$')
ax.set_ylabel(r'$x_\infty$')
ax.set_title("Bifurcation diagramme for the logistic map");
```


Bifurcation diagramme for the logistic map
1.0
0.8
0.6
0.4
0.2
0.0
x_∞
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
r

```
[ ]:
```