Introduction to Numerical Python

Volodymyr Byelobrov

python™

Contents

1 Preface

This short manual is a very brief introduction to the most basic and used features of Numerical Python. Python is currently one of few most used programming languages. There is motto "If you think about something, likely it is in Python". That is even more relevant to Numerical Python as it suddenly became a dominant numerical tool in science, big data, etc. Python packages cover almost all domains of science. Some people say that its three most known packages NumPy, SciPy, and MatPlotLib substitute MatLab. Of course it is not. They are different tools which both have their pros and cons. However, fortunately for Matlab admires, all these three packages have pretty similar syntax to MatLab.

It would be logical to start from basics of Python, then dive deeper in its numerous aspects and only then come to NumPy, which is actually a cherry on top. However we will start from that "cherry" and gradually learn more about Python.

One of the most attractive aspect of Python is its documentation and numerous specific books dedicated to it. As this manual is not anyhow exhaustive and comprehensive it is strictly advised to use it. For general Python information, please check the following resources:

- `https://docs.python.org/3/`
- `https://www.tutorialspoint.com/python/`
- `https://www.python-course.eu/`
- `https://www.programiz.com/python-programming/`
- `http://book.pythontips.com/en/latest/`

For numerical packages:

- `https://docs.scipy.org/doc/numpy/index.html`
- Python Data Science Handbook
- `https://scipython.com/`
- `https://scipy-lectures.org`

And, finally, MatPlotLib has a perfect documentation on its own:

- `https://matplotlib.org/contents.html`

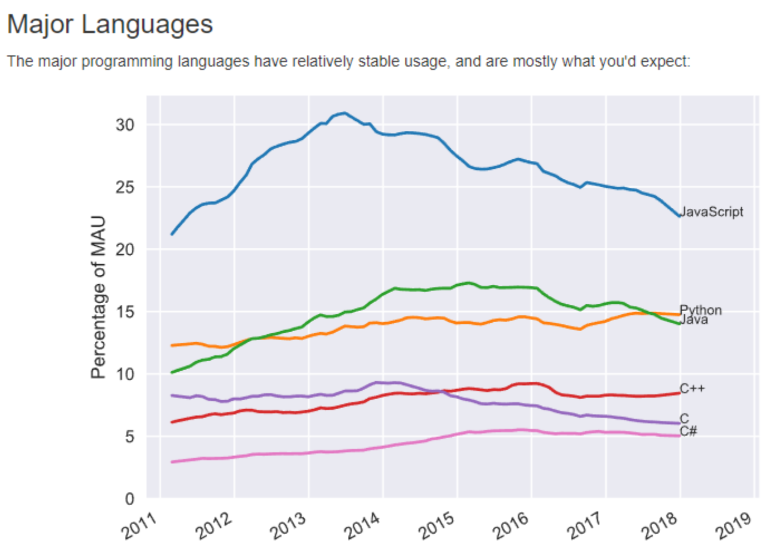
2 Introduction to Python

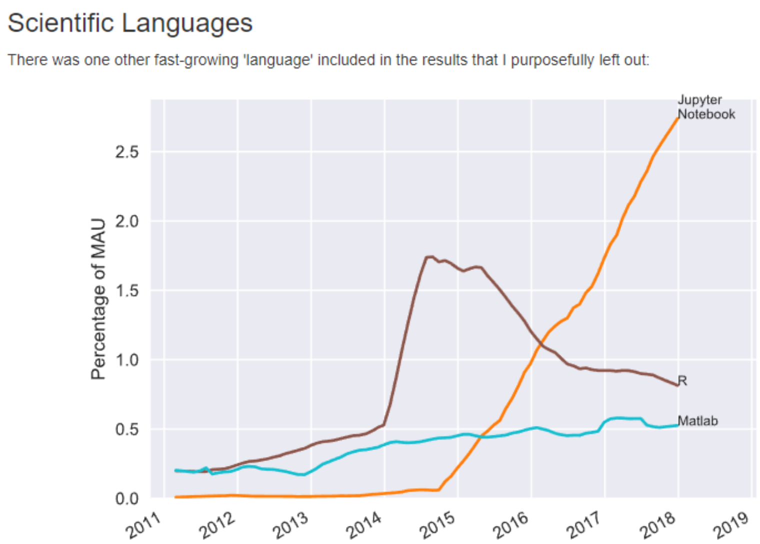
2.1 Basic information

Python is an interpreted high-level programming language for general-purpose programming. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales.

Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

Python is one of the most popular programming languages with over 1M projects on GitHub with dynamics of





Obviously Python became so popular due to numerous advantages, briefly summarized in the following very subjective list

- it's free and open source with a vibrant community
- extensive libraries if you think about something, like there is already implemented in Python
- combines object oriented programming (OOP) with features of functional languages
- supports vectorization
- highly readable syntax (indents)
- portable, the same code works (almost) always on different OS
- oriented for numerical calculations e.g. complex number is a base type
- containers and comprehensions are brilliantly implemented and their syntax became a template for other programming languages

However Python has also its disadvantages:

- it's free that means no official support what could be an issue for large projects, very large... that means it is not our problem
- some people aren't satisfied by its functional features
- slow speed comparing to compiling languages like C, C++, etc. As Python is **high-level**, **interpreted**, and **dynamically-typed** language it makes a small type-checking overhead of every operation. That may cause an increase significantly the time execution of small operations, like loops.
- problems with threading

Python Virtual Machine (PVM) The most famoust and studied here Python version is called CPython, It is a program written in C that translates source code into some efficient intermediate representation (code) and immediately executes this. Virtual Machine, explicitly executes stored pre-compiled code built by a compiler which is part of the interpreter system.

> Everything is an object...

Everything is an object is one of the most known Python motto. It means that every variable, function, or class is treated as objects. E.g. they may be past as a function parameter. Every object has its `id` and `type` that could be yielded by functions *id()* and *type()* .

```
>>> a = 2
>>> id(2)
140723092632432
>>> id(a)
140723092632432

>>> a = 3
>>> id(a)
140723092632464

>>> type(3)
<class 'int'>

>>> type(3j)
<class 'complex'>

>>> type('3j')
<class 'str'>
```

2.2 Comments

Comments are brilliant way to increase core readability. They are an integral part of any program and can come in the form of module-level docstrings, or even inline explanations that help shed light on a complex function.

Inline comments in Python are marked by # in the beginning. These comments may start from the line beginning with respect to indention, or be a continuation of a code line.

Block comments are surrounded by triple quotes """ or '''.

> "Code is more often read than written."
>
> — Guido Van Rossum

```
# inline comment may be from the start of line
# or comment the line they follow
a = 5                   # assign the variable a to 5
for i in range(10):
    print(i)            # or here
'''
block comments may contain
any number of lines
'''

"""
or like this
"""
```

Block comments after a function or class declaration are docstrings and saved in `__doc__`.

```
>>> def square(n):
        """
        the square function
        :param n: int, number
        :return:  n square
        """
        return n ** 2
>>> square(2)
4
>>> square.__doc__
the square function\n
:param n: int, number\n
:return:  n square\n
```

2.3 Arithmetic operators

operator	desciption	example	class method
+	addition	5 - 2 == 3	__add__(self, other)
*	multiplication	2 * 5 == 10	__mul__(self, other)
/	division	3 / 2 == 1.5	__truediv__(self, other)
%	modulus	23%5 == 3	__mod__(self, other)
**	exponent	2**10 == 1024 pow(2,10) == 1024	__pow__(self, other)
//	floor division	7//3==2, 7.//3.==2. -7//3==-3, -7.//3.==-3.	__floordiv__(self, other)

2.4 Comparison operators

operator	desciption	example	class method
==	equal	3 == 3	__eq__(self, other)
!=,<>	not equal	2 != 3	__ne__(self, other)
>	greater	5>3	__gt__(self, other)
>=	greater or equal	4 >= 3	__ge__(self, other)
<	less	1<3	__lt__(self, other)
<=	less or equal	1 <= 1	__le__(self, other)

2.5 Assignment operators

operator	desciption	example	class method
=	assignment	a=3 a,b=3,4	
+=	adds a value to the variable	a+=3 a==6	__iadd__(self, other)
-=	subtracts a value from the variable	a-=1 a==2	__isub__(self, other)
*=	multiplies the variable by a value	a*=2 a==6	__imul__(self, other)
/=	divides the variable by a value	a/=2 a==1.5	__idiv__(self, other)
//=	takes floor division of the variable by a value	a//=2 a==1	__ifloordiv__(self, other)
%=	takes modulus and assigns	a%=2 a==1	__imod__(self, other)
**=	takes exponent and assigns	a**=2 a==9	__ipow__(self, other)

2.6 Indention

Most of the programming languages like C, C++, Java use curly brackets to define a block of code. Python uses indentation. Generally four whitespaces are used for it and is preferred over tabs.

```
def square(x):
```

```python
    return x*x

def _max(a,b):
    if a>b:
        return a
    else:
        return b

for i in range(8):
    print(i)
```

> **The golden rule**
> Indent always follows a colon (:)

2.7 Basic types

type	description	value
bool	boolean value	True, False
int	integer value	a=2
float	number with a floating point	a=3.
complex	complex number	a=3.+4.j
str	string	a='3.+4.j'

3 IF statement

The in-line IF statement has the following form:

value1 **if** *statement* **else** *value2*

It returns *value1* if statements is **True** and *value2* otherwise, one should note that both keywords **if** and **else** are mandatory.

```
a, b = 3, 4
_max = a if a > b else b
```

The block IF statement has only one mandatory keyword **if** that followed by a statement and colon (:). It might accompanied by arbitrary number of blocks starting from **elif** statements, and closed by the final **else** statement.

```
if a > b and a > c:
    _max3 = a
elif b > c:
    _max3 = b
else:
    _max3 = c
```

Task
Write in-line statement for finding maximum value of three numbers

4 LOOPS

4.1 FOR statement

The for statement in Python has the ability to iterate over the items of any sequence, such as a list, a string, or specific object produced by the function range().

```
>>> for i in range(10):
>>>     print(i, end=' ')
0 1 2 3 4 5 6 7 8 9
```

The **else** clause of an FOR statement is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement.

4.2 WHILE statement

```
>>> counter = 0
>>> while counter < 10:
>>>     print(counter, end=' ')
>>>     counter += 1
0 1 2 3 4 5 6 7 8 9
```

If the **else** statement is used with a while loop, the else statement is executed when the condition becomes false.

```
>>> counter = 0
>>> while counter < 10:
>>>     print(counter, end=' ')
>>>     counter += 1
>>>     if counter == 7:
>>>         break
>>> else:
>>>     print('end')
0 1 2 3 4 5 6
```

4.3 BREAK and CONTINUE

In Python, break and continue statements can alter the flow of a normal loop.
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.
If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

```
>>> for i in range(1, 10):
>>>     if i % 7 == 0:
>>>         break
>>>     print(i, end=' ')
>>> print('the end')
1 2 3 4 5 6 the end
```

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

```
>>> for i in range(1, 10):
>>>     if i % 7 == 0:
>>>         continue
>>>     print(i, end=' ')
>>> print('the end')
1 2 3 4 5 6 8 9 the end
```

4.4 Function RANGE

The function range produces an iterable object optimized for the FOR statement.

range*(start, stop, step)*

start	(optional) starting point of the sequence. It defaults to 0.
stop	(required) endpoint of the sequence.
step	(optional) step size of the sequence. It defaults to 1.

example	output
range(10)	0 1 2 3 4 5 6 7 8 9
range(1,10)	1 2 3 4 5 6 7 8 9
range(1,10,2)	1 3 5 7 9
range(10,-1,-1)	10 9 8 7 6 5 4 3 2 1 0
range(10,-1,-2)	10 8 6 4 2 0

prime number is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers.

Task 1

find all prime numbers less than 10,000 and measure execution time

Task 2

make it quicker...

Task 3

and even quicker..?

5 Python: Strings

Python strings are containers of characters. Their behaviour is more similar to `list`. Therefore the rule of slicing or application of the function ***len***(*iterable*) is the same as for `list`.

```
>>> s = 'Hello World!'
>>> s
'Hello World!'
>>> len(s)
12
>>> s[0]
'H'
>>> s[-1]
'!'
>>> s[::-1]
'!dlroW olleH'
>>> s[::2]
'HloWrd'
```

5.1 Formatting

```
>>> a, b, c = 2, 3, 4
>>> '{}x^2 + {}x + {} = 0'.format(a,b,c)
'2x^2 + 3x + 4 = 0'
>>> f'{a}x^2 + {b}x + {c} = 0'
'2x^2 + 3x + 4 = 0'
```

5.2 Operations

+	'abc'+'def' == 'abcdef'
*	'ab ' * 5 == 'ab ab ab ab ab '
in	'bcd' in 'abcde' == True

5.3 Methods

***zfill**(n)*	fills with zeros the string beginning to create a string of length n
***lower**()* , ***upper**()*	returns the lowercase or uppercase version of the string
***strip**()*	returns a string with whitespace removed from the start and end
***isdigit**()*	tests if all the string chars are in the various character classes
***find**('other')*	searches for the given other string within s, and returns the first index where it begins or -1 if not found
***replace**('old', 'new')*	returns a string where all occurrences of 'old' have been replaced by 'new'
***split**('delim')*	returns a list of substrings separated by the given delimiter.
***join**(list)*	opposite of split(), joins the elements in the given list together using the string as the delimiter

5.4 Special characters

\b	ASCII Backspace (BS) character
\n	ASCII Linefeed (LF) character
\t	ASCII Horizontal Tab (TAB) character

Task 1

- How many digits are in 2^{123}?
- How many digits are in 2^{1234}?
- Can you do it without taking power of 2?

Task 2

- Does 2^{1234} contain the sequence 777?
- Does 2^{12345} contain the sequence 777?
- Make the expression that will imitate work of str.zfill()

Task 3

- Find the smallest x that 2^x contains the sequence 777.
- Find the smallest x that 2^x contains the sequence 7d7d7, where d is any digit.

5.5 Regular Expression

Regular Expression (RegEx) is a sequence of characters that define a search pattern. Usually this pattern is used by string searching algorithms for "find" or "find and replace" operations on strings, or for input validation. It is a technique developed in theoretical computer science and formal language theory. In **Python** standard string methods are capable to do the same work, however in some searches **RegEx** are proved to be an eloquent tool for working with string. In **Python** has a module named **re** to work with regular expressions.

```python
import re
```

Besides Python you may verify your patters on following sites:

- pyregex.com
- regexr.com
- regex101.com

The most important in **RegEx** is to construct correctly the pattern that will fetch your search.

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

[ ]	square brackets	specifies a set of characters you wish to match
.	period or wildcard	matches any single character (except newline \n)
^	caret	is used to check if a string starts with a certain character
$	dollar	is used to check if a string ends with a certain character
*	star	matches zero or more occurrences of the pattern left to it
+	plus	matches one or more occurrences of the pattern left to it
?	question mark	matches zero or one occurrence of the pattern left to it
		{m} — the preceding item is matched exactly m times
{}	braces	{m,} — the preceding item is matched m or more times
		{m,n} — matched at least m times, but not more than n times
|	alternation	is used for alternation (logical or operator)
()	group	is used to group sub-patterns
\	backslash	is used to escape various characters including all metacharacters

By the way, underscore _ is also considered an alphanumeric character
Special sequences make commonly used patterns easier to write.

\A	matches if the specified characters are at the start of a string
\b	matches if the specified characters are at the beginning or end of a word
\B	opposite of \b. Matches if the specified characters are not at the beginning or end of a word
\d	Matches any decimal digit. Equivalent to [0-9]
\D	Matches any non-decimal digit. Equivalent to [^0-9]
\s	Matches where a string contains any whitespace character. Equivalent to [\t\n\r\f\v]
\S	matches where a string contains any non-whitespace character. Equivalent to [^\t\n\r\f\v]
\w	matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]
\W	matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_]
\Z	Matches if the specified characters are at the end of a string

findall*(pattern, string)*	returns a list of strings containing all matches
split*(pattern, string)*	splits the string where there is a match and returns a list of strings where the splits have occurred
sub*(pattern,replace, string)*	returns a string where matched occurrences are replaced with the content of replace variable
search*(pattern, string)*	looks for the first location where the RegEx pattern produces a match with the string. If the search is successful, re.search() returns a **match object**; if not, it returns `None`

Match object

match.group()	returns the part of the string where there is a match
match.start()	returns the index of the start of the matched substring
match.end()	returns the end index of the matched substring
match.span()	returns a tuple containing start and end index of the matched part

6 Python: Containers

6.1 Lists

Python list is the most versatile datatype available in Python, which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that the items in a list need not be of the same type.

```
a = [1,2,3,4,5]                              # assign
len(a) == 5                                  # get the length
a[0] == a[-5] == 1                           # get an item
a[-1] == a[4] == 5
del a[2]; a == [1,2,4,5]                             # remove an item
a[2] = 'abc'; a == [1,2,'abc',5]                     # assign an item
a[1] = ['a','b','c']                                 # assign an item
a == [1,['a','b','c'],12,13,'abc',5]
a[1:2] = [11,12,13]; a == [1,11,12,13,'abc', 5] # weird
a[1:4] = [-1,-2]; a == [1,-1,-2,'abc',5] # very weird
```

In Python **list** slicing is no different from **string** one.

List operations Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

```
a =[0,1,2,3,4,5]
b = ['a', 'b','c','d','e']
```

a*2	[0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5]
a+b	[0, 1, 2, 3, 4, 5, 'a', 'b', 'c', 'd', 'e']
len()	len(a+b)=10
	min(a),max(a) == 0 , 4
min(),max()	min(b),max(b) == a , e
zip()	list(zip(a,b)) == [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
enumerate()	list(enumerate(b))==[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string. a =[0,1,2,3,4]

List methods

append()	add single element to the list	a.append(5) a==[0, 1, 2, 3, 4, 5]
extend()	add elements of a list to the list	a.extend([5,6]) a==[0, 1, 2, 3, 4, 5,6]
insert()	inserts element to the list	a.insert(3,5) [0, 1, 2, 5, 3, 4, 5, 6, 7]
remove()	removes element from the list	a.append(5) a==[0, 1, 2, 3, 4, 5]
index()	returns smallest index of element in list	a.index(5)==3
count()	returns occurrences of element in a list	a.count(5) == 2
pop()	removes element at given index	a.pop() == 7
sort(n)	sorts elements of a list (in place)	a == [0, 1, 2, 4, 5, 5, 6]

According to the Binomial Theorem

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

where the Binomial coefficients $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Unexpanded Form	Expanded Form
$(a+b)^0 =$	1
$(a+b)^1 =$	$a + b$
$(a+b)^2 =$	$a^2 + 2ab + b^2$
$(a+b)^3 =$	$a^3 + 3a^2b + 3ab^2 + b^3$
$(a+b)^4 =$	$a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$
$(a+b)^5 =$	$a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5$

```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
  1   5   10  10  5   1
1   6   15  20  15  6   1
```

6.2 Sets

A **set** is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). However, the set itself is mutable. We can add or remove items from it. Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

Sets are **mutable** collections of **unique** items of **immutable** types. Sets do not support **indexing**.

Assigning empty set

```python
set1 = set()
```

Assigning set

```python
set1 = {1,2,3,4}
set2 = set('Hello World!')
set2 == {'o', 'e', '!', ' ', 'r', 'H', 'W', 'd', 'l'}
```

Comprehension

```python
set1 = {i for i in range(4)}
```

Equivalent block statement for set creation

```
set1 = set()
for i in range(4):
    set1.add(i)
set1 == {0, 1, 2, 3}
```

Set operations fully implement logic operations therefore they have identical names and properties.

$A \mid B$	A.union(B)	Returns a set which is the union of sets A and B.
$A \,\&\, B$	A.intersection(B)	Returns a set which is the intersection of sets A and B.
$A - B$	A.difference(B)	Returns the elements included in A, but not included in B.
$A \wedge B$	A.symmetric_difference(B)	Returns the elements belonging to either A or B, but not to both sets simultaneousl).
$A <= B$	A.issubset(B)	Returns true if A is a subset of B.
$A >= B$	A.issuperset(B)	Returns true if B is a subset of A.
$A < B$		Equivalent to $A <= B$ and $A! = B$
$A > B$		Equivalent to $A >= B$ and $A! = B$

```
A = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18}
B = {0, 3, 6, 9, 12, 15, 18}
A | B == {0, 2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18}
A & B == {0, 18, 12, 6}
A - B == {2, 4, 8, 10, 14, 16}
A ^ B == {2, 3, 4, 8, 9, 10, 14, 15, 16}
```

add(x)	add element x to the set
remove(x)	remove x from the set; raises KeyError if not present
discard(x)	removes x from the set if present
pop$()$	remove and return an arbitrary element from the set; raises KeyError if empty
clear$()$	remove all elements from the set

Task
Sets are collection of **immutable** objects. However could you find and example when you can't pass it an **immutable** object?

Use the package NetworkX for constructing and displaying graphs. Assembly a random graph. You may use the script graph.py

Task 1

Write your function to colour nodes according to the number of neighbours the node has

Task 2

Write your own function to calculate number of connected components and set the same colour to all nodes of one connected component.

6.3 Tuples

Tuples are **immutable** collections of **not-unique** items of arbitrary types.

Assigning empty tuple

```
tuple1 = ()
tuple2 = tuple()
```

Assigning tuple

```
tuple1 = (1,2,3,4)
tuple2 = tuple('python')
```

Equivalent block statement for tuple creation Comprehension

```
tuple1 = tuple(i for i in range(4))
```

Block statement

```
tuple1 = ()
for i in range(4):
    tuple1 += i,
tuple1 == (1, 2, 3, 4)
```

A **tuple** is a sequence of immutable Python objects. Tuples are sequences, just like lists. The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists.

Tuples possess all list operations, functions, and methods like ***remove()*** and ***index()*** those don't require their mutation, i.e. min(), max(), any(), *, + are all presented, but sort(), insert(), extend() are absent.

Task

Tuples are immutable i.d. you can't change neither its length or items, however does it really mean you can't change them at all? Could you find an example of creating **tuple** and then altering it, its **id** should be the same.

6.4 Dictionaries

Dictionaries are **mutable** collections of pairs key:value, where keys are **unique** with the dictionary and of **immutable** types, while values don't have any restrictions. Dictionaries support indexing by key value.

Assigning empty list

```python
dict1 = dict()
dict2 = {}
```

Assigning dictionary

```python
dict1 = {1:1, 2:4, 3:9, 4:16}
```

Comprehension

```python
dict1 = {i: i**2 for i in range(4)}
```

Python **dictionary** is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair. Dictionaries are optimized to retrieve values when the key is known.

***get**(key)*	For key key, returns value or default if key not in dictionary
***items**()*	Returns a list of dict's (key, value) tuple pairs
***keys**()*	Returns list of dictionary dict's keys
***values**()*	Returns list of dictionary dict's values

```python
A = {i: i ** 2 for i in range(10) if i % 2 == 0}
A.get(2) == A[2] == 4
A.items() == dict_items([(0, 0), (2, 4), (4, 16), (6, 36), (8, 64)])
A.keys() == dict_keys([0, 2, 4, 6, 8])
A.values() == dict_values([0, 4, 16, 36, 64])
```

Task
Print out the Bessel function roots alongside with their order (two integers) in ascending order
Hint
Use the function $jn\_zeros(n, nt)$ from the package scipy.special

Task
Type `import this` and you will see some text. Extract this text.
Hint
You might need to use the function `dir()` or find the source file.

6.5 Containers overview

	list	**tuple**	**set**	**dict**	**str**
initialization	[ ] list()	() tuple()	set()	{} dict()	” str()
mutable	✓		✓	✓	✓
immutable		✓			
index	✓	✓			✓
key				✓	
len	✓	✓	✓	✓	✓
max	✓	✓	✓	✓	✓
min	✓	✓	✓	✓	✓
any	✓	✓	✓	✓	?
all	✓	✓	✓	✓	?

	comprehension expression	result
list	[i for i in range(5)]	[0, 1, 2, 3, 4]
tuple	tuple(i for i in range(5))	(0, 1, 2, 3, 4)
set	{i for i in range(5)}	{0, 1, 2, 3, 4}
dict	{i:i**2 for i in range(5)}	{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
str	''.join([str(i) for i in range(5)])	'01234'

7 Python: Functions

7.1 Function Declaration

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your program and a high degree of code reusing.

In Python, functions are the first class objects, which means that –

- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.

```
def function_name(a, b=1, c=None, *args): #parameters
    '''
    optional description of the function
    :param a: description of parameters
    :return : return description
    '''
    result = a+b if c is None else a+b+c
    return result
```

Accessing to the function documentation, similarly setting it, one could do through the field function_name.__doc__.
After declaring a function you may call it in any part of your script, additionally you may rename any function in any part of your code.

```
>>> new_print = print
>>> new_print('Hello World!')
Hello World!
```

```
def new_sum(a, b=1, c=None):  # parameters
    return a + b if c is None else a + b + c

val = new_sum(1)
print('val:\t', val)
val2 = new_sum(a=2)
print('val2:\t', val2)
print('val3:\t', new_sum(3, b=2))
```

There is not function overloading in Python. However if it is needed to use one function for different sets Python allows manipulation with parameters inside the function body using **default** value, tuple of variables ***args**, dictionary of keyworded variables ****kwargs**

```
>>> def new_sum(a, b=1, c=None):  # parameters
>>>     result = a + b if c is None else a + b + c
>>>     return result
```

```
>>> new_sum(1)
2
>>> new_sum(a=2)
3
>>> new_sum(3, b=2)
5
```

Similarly to copying of two objects, **mutable** and **immutable** objects behave on passing as function parameter. **Mutable** objects are passed by **reference**. **Immutable** objects are passed by **value**.

```
>>> def increase(n):
>>>     n += 3
>>>     return n
>>> var = 5
>>> print('var before:', var)
var before: 5
>>> increase(var)
>>> print('var after:', var)
var after: 5
```

```
>>> def append(lst):
>>>     lst.append(3)
>>>     return lst
>>> l = [1, 2]
>>> print('before:', l)
before: [1, 2]
>>> append(_list)
>>> print('after:', l)
after: [1, 2, 3]
```

7.2 Asterisk Operator *

The prefix operators `*` is used for unpacking data from collections.

```
>>> lst = [1, 2, 3]
>>> print('calling without asteriks', lst)
calling without asteriks [1, 2, 3]
>>> print('calling with asteriks', *lst)
calling with asteriks 1 2 3
```

The prefix operators `*` and `**` are used for passing data to a function that has ***args** or ****kwargs** parameters.

```
def my_sum(a, b, c):
    return a + b + c
lst = [1, 2, 3]
print(my_sum(*lst))
dic = {'a': 1, 'b': 2, 'c': 3}
print(my_sum(**dic))
```

```
lst = [1, 2, 3, 4]
print(my_sum(*lst)) # error
dic = {'a': 10, 'b': 20, 'c': 30, 'd': 4}
print(my_sum(**dic))  # error
```

7.3 Anonymous function

The **lambda** keyword in Python provides a shortcut for declaring small anonymous functions. Lambda functions behave just like regular functions declared with the *def* keyword. They can be used whenever function objects are required.

```
>>> f = lambda x: x**3 if x%2 == 1 else x**2
>>> [f(i) for i in range(2,7)]
[4, 27, 16, 125, 36]
```

The **lambda** functions are used for short statements or as a callable parameter

```
>>> def increase_by(n):
>>>     return lambda x: x + n
>>> func_list = [increase_by(n) for n in range(10)]
>>> [f(1) for f in func_list]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

One of the most frequent lambda use is as parameter of **sort** function.

```
a = [3, 6, 2, 8, 3]
sorted(a) == [2, 3, 3, 6, 8]

b = [4, -2, 1, -12, -4, -3]
sorted(b, key=lambda x: abs(x))==[1,-2,-3,4,-4,-12]

c=('Mrs Dalloway said she would buy'\
+ 'the flowers herself').split(' ')
sorted(c, key=lambda x: x[1])==['Dalloway','said',\
'herself','she','the','flowers','would','Mrs','buy']
```

Explain the following behaviour of lambdas in list comprehensions. Will it be the same in tuple, set comprehensions, dictionaries, generators? How prevent of copying the last item to all one?

```
>>> def increase_by(n):
>>>     return lambda x: x + n

>>> func_list = [increase_by(n) for n in range(10)]
>>> [f(1) for f in func_list]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> func_list1 = [lambda x: x + n for n in range(10)]
>>> [f(1) for f in func_list1]
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

7.4 Recursion

A recursive function is a function defined in terms of itself via self-referential expressions.

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Task
Make a function that will flatten nested lists.

```
        lst = [1, [2, [3, [4, 5, [6, 7, 8], 9], 10]], 12]
        flatten(lst) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12]
```

Hint
Use the function Iterable from the package collections that returns True if the given object is iterable.

7.5 Function callable

```
import collections
isinstance([], collections.Iterable) == True
```

type	Iterable	type	Iterable
int	False	**bool**	False
float	False	**complex**	False
list	True	**tuple**	True
str	True	**dict**	True
set	True	**frozenset**	True

Built-in function **callable** returns True if the object argument appears callable, False if not. If this returns true, it is still possible that a call fails, but if it is false, calling object will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a __call__() method.

```
callable(print) == True
callable(print()) == False
callable(int) == True
callable(4) == False
callable(list) ==True
callable([]) == False
```

7.6 Function isinstance

Returns a Boolean stating whether the object is an instance or subclass of another object.

```
import numbers

```

```
isinstance(2, int) == True
isinstance(2., int) == False
isinstance(2., float) == True
isinstance(2, float)) == False
isinstance(2j, complex) == True
isinstance(2j, int)) == False
isinstance(3+2j, numbers.Number) == True
isinstance(3+2j, (int, float, complex)) == True
isinstance('2', str) == True
```

Syntax: `assert statement[, 'message']`

The **assert** keyword is used when debugging code.

The **assert** keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError.

```
>>> var = '4'

>>> assert isinstance(var, int), f'object var has\
                    type {type(var)} and value {var}'
AssertionError: object var has type <class 'str'> and value 4
```

Task
Write in-line statement for finding maximum value of three numbers

7.7 Decorators, Enclosures, Nested functions

A function defined inside another function is called a **nested function**. Nested functions can access variables of the enclosing scope.

In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using `nonlocal` keyword) in order to modify them.

Following is an example of a nested function accessing a non-local variable.

```
def list_power(lst):
    # enclosing function
    power = 2
    print('enclosing function scope', dir())

    def lst_iteration():
        # nested function
        print('nested function scope', dir())
        nonlocal power
        power += 1
        print([item ** power for item in lst])

    lst_iteration()
```

Here one can see that the nested function lst_iteration() was able to access the non-local variable power and lst of the enclosing function, however to alter them it ought to declare it through keyword `nonlocal`.

Nevertheless the enclosing function may return the nested one instead of explicitly calling it as in the example above.

```python
def list_power(lst):
    # enclosing function
    power = 2

    def lst_iteration():
        # nested function
        return [item ** power for item in lst]

    return lst_iteration

if __name__ == '__main__':
    print(list_power([1, 2, 3])())
```

Please note, that the returned value is a function so to get similar outcome as before one need to call it.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

In decorators, functions are taken as the argument into another function and then called inside the wrapper function.

```python
import datetime

def time_it(func):
    t1 = datetime.datetime.now()

    def func_wrapper(*args, **kwargs):
        return func(*args, **kwargs)

    t2 = datetime.datetime.now()
    print((t2 - t1).microseconds, 'ms')
    return func_wrapper


if __name__ == '__main__':
    import numpy as np
    import random
    N = 10 ** 7
    array = [random.random() for i in range(N)]
    np_array = np.array(array)

    @time_it
    def python_max(ar):
        return max(ar)


    @time_it
```

```
    def numpy_max(ar):
        return np.max(ar)

    print(python_max(array))
    print(numpy_max(np_array))
```

One can pass arguments to decorators, however in this case the function should be once more wrapped.

```
import datetime

def time_it(name):
    def named_function(func):
        t1 = datetime.datetime.now()

        def func_wrapper(*args, **kwargs):
            return func(*args, **kwargs)

        t2 = datetime.datetime.now()
        print(name, (t2 - t1).microseconds, 'ms')
        return func_wrapper
    return named_function


if __name__ == '__main__':
    import numpy as np
    import random

    N = 10 ** 5

    array = [random.random() for i in range(N)]
    np_array = np.array(array)

    @time_it('Python execution time')
    def python_max(ar):
        return max(ar)

    @time_it('Numpy execution time')
    def numpy_max(ar):
        return np.max(ar)

    print(python_max(array))
    print(numpy_max(np_array))
```

8 Python: OOP

Everything is an Object in Python... So what is an object? Objects are instances of classes... up to now all our codes were *object-based* - we've passed objects around our scripts, used them in expressions, called their methods, and so on. However we couldn't create **new types**. For this we will learn *object-oriented programming*.

def declares a *function* or a block of code.
class define a *class* or simply a logical grouping of data and functions (methods).

NB: OOP is absolutely optional, nevertheless it shortens the code and provides a lot of other benefits...

```python
class Fruit:
    edible = True  # static field

    def __init__(self, name, calories):
        self.name = name
        self.calories = calories

    def __str__(self):
        return f'{self.name} has {self.calories} cal per 100 g.'

    def get_name(self):  # attribute
        return self.name

    @staticmethod
    def is_edible():
        return Fruit.edible
```

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list self , but you do not give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object itself, and by convention, it is given the name self.

This also means that if you have a method which takes no arguments, then you still have to have one argument - the self.

There are four fundamental OOP concepts, which makes OOP approach as powerful as it is.

- **Inheritance.** This allows classes to be arranged in a hierarchy that represents "is-a-type-of" relationships. Child class derives functionality of its parent. This is powerful tool of code reuse.
- **Polymorphism.** This might be complicated to understand at first, but basically polymorphism is ability to provide access to entities of different types through single interface. The basic goal of polymorphism is to make applications more modular and extensible.
- **Abstraction.** Abstraction is simplifying complex reality by modelling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem.

- **Encapsulation.** Encapsulation is a strategy used as part of abstraction. Encapsulation refers to the state of objects - objects encapsulate their state and hide it from the outside; outside users of the class interact with it through its methods, but cannot access the classes state directly.

8.1 Abstraction

Abstraction means we focus on the essential qualities of something to create a class.

```
class Fruit:
    def __init__(self, name, calories):
        self.name = name
        self.calories = calories

    def __str__(self):
        return f'{self.name} has {self.calories} cal per 100 g.'

if __name__ == '__main__':
    apple = Fruit('Apple', 52)
    apricot = Fruit('Apricot', 48)
    avocado = Fruit('Avocado', 160)

    fruit_list = [apple, apricot, avocado]
    for fruit in fruit_list:
        print(fruit)
```

Output

```
Apple has 52 cal per 100 g.
Apricot has 48 cal per 100 g.
Avocado has 160 cal per 100 g.
```

8.2 Encapsulation

To encapsulate is to keep the contents of an object together. And also to separate the object from other objects, and to protect those contents. The principle is that an object should not reveal anything about itself except what is necessary for other part of application to work. Encapsulation helps to reduce dependencies between different parts of the application. So that a change in one place won't cascade down and need many changes elsewhere.

```
class Fruit:
    def __init__(self, name, calories):
        self.__name = name
        self.calories = calories
    def __str__(self):
        return f'{self.__name} has {self.calories} cal per 100 g.'

if __name__ == '__main__':
    apple = Fruit('Apple', 52)
    print(apple)
    apple.calories = 56
    print(apple)
    apple.__name = 'Pear'  # you can't access 'private' methods or field directly
    print(apple)
```

Other programming languages have protected class methods too, but Python does not.

Encapsulation gives you more control over the degree of coupling in your code, it allows a class to change its implementation without affecting other parts of the code.

Output

```
Apple has 52 cal per 100 g.
Apple has 56 cal per 100 g.
Apple has 56 cal per 100 g.
```

8.3 Inheritance

We could create a class, but instead of writing it from scratch, we could use another class as a base class.

```
class Fruit:
    def __init__(self, name, calories, type='Berry'):
        self.__name = name
        self.calories = calories
        self.type = type
    def __str__(self):
        return f'{self.__name} has {self.calories} cal per 100 g.'

class Berry(Fruit):
    def __init__(self, name, calories):
        Fruit.__init__(self, name, calories, type='Berry')

class Pomes(Fruit):
    def __init__(self, name, calories):
        Fruit.__init__(self, name, calories, type='Pomes')

if __name__ == '__main__':
    apple = Pomes('Apple', 52)
    print(apple)
    raspberry = Berry('Raspberry', 64)
    print(raspberry)
```

Output

```
Apple has 52 cal per 100 g.
Raspberry has 64 cal per 100 g.
```

8.4 Polymorphism

Polymorphism means that a class could have many forms. So, its behavior will depend on as what class we treat it.

```
class Fruit:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return '{} is a fruit'.format(self.name)

class Berry(Fruit):
    def __init__(self, name):
        Fruit.__init__(self, name)
    def __str__(self):
        return '{} has many seed'.format(self.name)

class Drupe(Fruit):
    def __init__(self, name):
        Fruit.__init__(self, name)
    def __str__(self):
        return '{} has one seed'.format(self.name)

if __name__ == '__main__':
    raspberry = Berry('Raspberry')
    print(raspberry)
    peach = Drupe('Peach')
    print(peach)
    peach2 = Fruit('Peach')
    print(peach2)
```

Output

```
Raspberry has many seed
Peach has one seed
Peach is fruit
```

9 Package NumPy

9.1 Introduction

NumPy is a Python extension module that provides efficient operation on arrays of homogeneous data. It allows Python to serve as a high-level language for manipulating numerical data, much like for example IDL or MATLAB.

NumPy is designed to decrease execution time to Fortran/C level keeping easiness and readability of Python code by pushing repeated operation in a statically-typed compiled layer. **NumPy** is widely used as a Data Science tool as the latest global Python developers survey shows:


```
>>> import numpy as np
>>> np.__version__
'1.15.4'
```

You also may import all NumPy methods and fields in the global namespace using **from** statement. However this way is highly not recommended.

```
>>> from numpy import *
>>> pi
3.141592653589793
```

NumPy deals with all **Python** numerical types, like int, float, complex, and bool. However in **Numpy** all these types could be defined with different precision e.g. *np.float16*, *np.float32*, *np.complex128*, *np.int8*, *np.int32* etc. One should note that unlike Python where int values are not limited, **Numpy** int range is limited. Functions ***np.iinfo***() and ***np.finfo***() provide information about **Numpy** integer and floating point types, respectively.

```
>>> np.iinfo(np.int16)
iinfo(min=-32768, max=32767, dtype=int16)

>>> np.iinfo(np.int32)
iinfo(min=-2147483648, max=2147483647, dtype=int32)

>>> np.finfo(np.float16)
```

```
finfo(resolution=0.001, min=-6.55040e+04,\
                        max=6.55040e+04, dtype=float16)
```

Numpy has a floating point representation of (positive) infinity *np.inf*.

```
>>> 1/np.float32(0)
inf

>>> np.log(0)
-inf

>>> 1/np.inf
0

>>> np.inf + 3
inf

>>> np.inf * 3
inf

>>> np.inf + np.inf
inf

>>> np.log(np.inf)
inf
```

There is also a floating point representation of Not a Number *np.nan*.

```
>>> np.sqrt(-1)
nan

>>> np.sin(np.inf)
nan

>>> np.nan + 2
nan

>>> np.inf - np.inf
nan
```

In the first example above, it might seem that Numpy can't cope with taking the square root of -1. In fact, this behaviour is only caused by type-defined objects in Numpy. And it is easily resolved if we pass a complex-type object:

```
>>> np.sqrt(np.complex(-1))
1j
>>> np.sqrt(-1 + 0j)
1j
```

9.2 ndarrays basic

NumPy **ndarray**s are one of the most important and widely-used features of Numerical Python. They are a core of many other packages like Pandas.

An **ndarray** is a multidimensional container of items of the **same type** and **size**. The number of dimensions and items in an array is defined by its shape, which is a tuple of N positive integers that specify the sizes of each dimension. The type of items in the array is specified by a separate data-type object (dtype), one of which is associated with each ndarray.

Ndarray initialization NumPy offers several way to initialize the ndarray. The most general way is to call the function np.ndarrat(), frankly, it is rarely used, still we start from it.

numpy.ndarray(*shape, dtype=float, buffer=None, offset=0, strides=None, order=None*)

Parameters:

shape : ***tuple of ints***
Shape of created array.

dtype : ***data-type, optional***
Any object that can be interpreted as a numpy data type.

buffer : ***object exposing buffer interface, optional***
Used to fill the array with data.

offset : ***int, optional***
Offset of array data in buffer.
interpreted only if **buffer** is an object exposing the buffer interface

strides : ***tuple of ints, optional***
Strides of data in memory.
interpreted only if **buffer** is an object exposing the buffer interface

order : ***{'C', 'F'}, optional***
Row-major (C-style) or column-major (Fortran-style) order.
interpreted only if **buffer** is an object exposing the buffer interface

Listing 1: Illustration of the low-level ndarray constructor

```
>>> np.ndarray(shape=(3,), dtype=np.float64)
array([0.00000000e+000, 0.00000000e+000, 2.47823328e-320])

>>> np.ndarray(shape=(4, 3), dtype=np.bool)
array([[ True,  True,  True],
       [ True,  True,  True],
       [False, False,  True],
       [ True,  True,  True]])

>>> np.ndarray(shape=(1, 4), dtype=np.complex128)
array([[0.00000000e+000+0.00000000e+000j,
        0.00000000e+000+0.00000000e+000j,
        0.00000000e+000+2.47823328e-320j,
        3.57360117e-306+3.57499184e-306j]])

>>> array = np.ndarray(shape=(2, 3), dtype=np.int)
>>> array
```

```
array([[          0,            0, -1253916128],
       [        618,            0, -2147483648]])
```

One can note, as you see it above, the **ndarray** constructor creates an array with arbitrary data. Also, ndarray may have any **Python** type like lists, sets etc, hence there are not originally designed for it. Nevertheless to do it you should pass **dtype=np.object**.

Now, consider some important attributes

Attributes: **dtype :** ***dtype object***
Data-type of the array's elements.
flags : ***dict***
Information about the memory layout of the array.
size : ***int***
Number of elements in the array.
ndim : ***int***
Number of array dimensions.
shape : ***tuple of ints***
Tuple of array dimensions.

Listing 2: ndarray attributes

```
>>> array.dtype
dtype('float64')
>>> array.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
>>> array.size
6              # == array.shape[0] * array.shape[1]
>>> array.shape
(2, 3)
>>> array.ndim
2              # == len(array.shape[0])
```

As you see, the array after using teh constructor **np.ndarray()** is not very neat as it is filled with random data. If we would like to have, for example, an array which element are equal to a given scalar value, we have two ways. Firstly, the method **fill()** assigns all array elements to the given scalar value. Secondly, using broadcasting (which will be discussed further) one may assign all elements to a particular scalar value, or using the Python **Ellipsis** literal "..." which is used for advanced slicing notation. Certainly, the most obvious way would be iterating over the array, whence hereof not counted above.

Listing 3: ndarray attributes

```
>>> array.fill(1)
>>> array
array([[1, 1, 1],
```

```
       [1, 1, 1]])

>>> array[:, :] = 2
>>> array
array([[2, 2, 2],
       [2, 2, 2]])

>>> array[...] = 3
>>> array
array([[3, 3, 3],
       [3, 3, 3]])
```

One should cautious of applying the last two approaches, as because of the PVM (Python Virtual Machine) design this operation may affect all future instances of the same type and size.

```
>>> ar1 = np.ndarray(shape=(2,3), dtype=np.int)
>>> ar1
array([[    3604533,     3407920,  1870623728],
       [        462,           0, -2147483648]])
>>> ar1[...] = 1
>>> ar1
array([[1, 1, 1],
       [1, 1, 1]])
>>> ar2 = np.ndarray(shape=(2,3), dtype=np.int)
>>> ar2
array([[1, 1, 1],
       [1, 1, 1]])
```

Additionally, the above ways are not efficient. Fortunately, NumPy has several cozy constructors those allow to create an array with desired data.

numpy.array(*object, dtype=None, copy=True, order='K', subok=False, ndmin=0)*

numpy.zeros(*shape, dtype=float, order='C')*

numpy.zeros_like(*a, dtype=None, order='K', subok=True)*

numpy.ones(*shape, dtype=float, order='C')*

numpy.ones_like(*a, dtype=None, order='K', subok=True)*

numpy.full(*shape, fill_value, dtype=None, order='C')*

numpy.full_like(*a, fill_value, dtype=None, order='K', subok=True)*

numpy.eye(*N, M=None, k=0, dtype=<class 'float'>, order='C')*

numpy.identity*(n, dtype=None)*

numpy.diag*(v, k=0)*

```
>>> np.array([[1,2,3], [2,3,4], [3,4,5]])
array([[1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])

>>> np.zeros(shape=(2, 3))
array([[0., 0., 0.],
       [0., 0., 0.]])

>>> np.ones(shape=(3, 2), dtype=np.complex128)
array([[1.+0.j, 1.+0.j],
       [1.+0.j, 1.+0.j],
       [1.+0.j, 1.+0.j]])

>>> np.full((2,2), True, dtype=np.bool)
array([[ True,  True],
       [ True,  True]])

>>> np.eye(N=3, M=4, dtype=np.complex128)
array([[1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j]])
```

One may create a ndarray by calling the function np.arange(n) which generates one-dimensional array witha values from 0 till n-1, and then manipulating the array shape by the function reshape().

```
>>> np.arange(12).reshape(3,4)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Also you can generate random **float** or **int** ndarrays using **NumPy** package **np.random**.

```
import numpy as np
>>> np.random.randn(3,4)
array([[ 0.96055287,  1.05930852, -1.02454838, -0.27037388],
       [ 0.17874319, -0.77100283,  0.85711132, -0.46897319],
       [-1.45428268,  0.19297873, -0.15411337, -0.54117629]])

>>> np.random.randint(10, size=(3,4))
array([[7, 2, 7, 3],
       [4, 8, 6, 8],
       [7, 6, 0, 6]])
```

9.3 ufunc

ufunc is a short form of **Universal Functions**, which are a special type of functions defined within the NumPy library those operate element-wise on arrays.
What are the **ufuncs**?

- Arithmetic Operations: $+ \quad - \quad * \quad @ \quad / \quad // \quad \% \quad **$
- Bitwise Operations: $\& \quad | \quad \backsim \quad \wedge \quad >> \quad <<$
- Comparison Operations: $< \quad > \quad <= \quad => \quad == \quad !=$
- Mathematical Functions: np.sin, np.cos np.sqrt np.exp, np.log, np.log10, etc.
- and many, many more

Arithmetic Operations Most arithmetic operation are valid for a **ndarray** and scalar.

```
>>> a = np.array([2, 4, 6, 8]).reshape(2, 2)
>>> a
array([[2, 4],
       [6, 8]])
>>> a + 1
array([[3, 5],
       [7, 9]])
>>> a - 5
array([[-3, -1],
       [ 1,  3]])
>>> a * 2
array([[ 4,  8],
       [12, 16]])
>>> a / 2
array([[1., 2.],
       [3., 4.]])
>>> a % 3
array([[2, 1],
       [0, 2]], dtype=int32)
>>> a // 3
array([[0, 1],
       [2, 2]], dtype=int32)
>>> a / 3
array([[0.66666667, 1.33333333],
       [2.        , 2.66666667]])
>>> a ** 2
array([[ 4, 16],
       [36, 64]], dtype=int32)
```

Generally, all arithmetic operations are implemented for two **ndarray**s, however one should also take into account arrays' dimension. The very basic rule says if the shape coincide then operation may be executed. Further we will expand this rule.

numpy.arange([start,]stop, [step,] dtype=None)

```
>>> a = np.array([10,20,30,40]).reshape(2,2)
>>> a
array([[10, 20],
       [30, 40]])
>>> b = np.array([2,4,6,8]).reshape(2,2)
>>> b
array([[2, 4],
       [6, 8]])
>>> a + b
array([[12, 24],
       [36, 48]])
>>> a - b
array([[ 8, 16],
       [24, 32]])
>>> a * b   # elementwise multiplication
array([[ 20,  80],
       [180, 320]])
>>> a @ b   # matrix product
array([[140, 200],
       [300, 440]])
>>> a / b
array([[5., 5.],
       [5., 5.]])
>>> b % a
array([[2, 4],
       [6, 8]], dtype=int32)
```

```
>>> a = np.arange(10,100,10).reshape(3,3)
>>> a
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
>>> a > 40
array([[False, False, False],
       [False,  True,  True],
       [ True,  True,  True]])
>>> a % 20 == 0          # here two operation performed
array([[False,  True, False],
       [ True, False,  True],
       [False,  True, False]])
>>> a == np.array([10, 50, 90])          # dimensions don't
array([[ True, False, False],             # coincide
       [False,  True, False],             # broadcasting
       [False, False,  True]])            # is used
```

```
>>> a = np.array([0, 30, 45, 60])
>>> np.sin(a * np.pi / 180)
array([0.        , 0.5       , 0.70710678, 0.8660254 ])

>>> a = np.arange(0, 90, 10).reshape(3, 3)
>>> np.abs(np.exp(1j * a * np.pi / 180))
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

9.4 Broadcasting

Broadcasting is a set of rule by which **ufuncs** operate on arrays of different size and/or dimensions.

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.

Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

```
>>> a = np.arange(12).reshape(3,4)
>>> b = np.arange(4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b
array([0, 1, 2, 3])
>>> a + b
array([[ 0,  2,  4,  6],
       [ 4,  6,  8, 10],
       [ 8, 10, 12, 14]])
```

Here before addition operation is executed the array b suffers a transformation from having

```
>>> b.shape
(4,)
```

```
>>> _b.shape
(3, 4)
```

```
>>> a = np.arange(3).reshape((3, 1))
>>> b = np.arange(3)
>>> a+b
```

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

9.5 Indexing of 1D ndarrays

Numpy indexing is similar to **Python** indexing of the sequences. Here indexing is shown on example of an 1D ndarray still the same rules are appliable to other sequences like lists (**list**), tuples (**tuple**), and strings (**str**).

Consider a one-dimensional array a of length N. Every its element may be accessed by passing into brackets the integer value from 0 till $N-1$ for *indexing from front*, or by passing the integer from -1 till $-N$ for *indexing from rear*.

value	0	1	2	3	4	5	6	7	8	9	10	11
index from front	0	1	2	3	4	5	6	7	8	9	10	11
index from rear	-12	-11	-10	9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> a = np.arange(12)
>>> a[0]
0
>>> a[3]
3
>>> a[11]
11
>>> a[-1]      # the last element
11
>>> a[-3]
9
>>> a[-12]
0
```

9.6 Slicing of 1D ndarrays

Comprehensive, eloquent, and sophisticated **Python** slicing for sequences is fully implemented in **Numpy**. Hence the last is enriched by numerous amount of additional tools for ndarrays. Still here we start from very basic Numpy slicing similar to one used for other sequences like lists (**list**), tuples (**tuple**), and strings (**str**).

The operator [n:m] returns the part of the sequence from the n'th element to the m'th element, including the first but excluding the last. This behaviour is counter-intuitive; it makes more sense if you imagine the indices pointing between the characters.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string.

```
>>> a = np.arange(12)
>>> a
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

```
>>> a[3:]
[ 3  4  5  6  7  8  9 10 11]
>>> a[-9:]
[ 3  4  5  6  7  8  9 10 11]
>>> a[:9]
[0 1 2 3 4 5 6 7 8]
>>> a[:-3]
[0 1 2 3 4 5 6 7 8]
>>> a[3:9]
[3 4 5 6 7 8]
```

Please, note that `a[:m]` and `a[m:]` split the array into two pieces.

```
>>> a = np.arange(12)
>>> np.all(np.concatenate((a[:5], a[5:])) == a)
True
```

The general form of the slicing contains three parameters:

array*[start : stop : step]*

Here the parameters *start*, *stop*, *step* could be negative if bigger than `-len(array)`. The second column is used when the *step* parameter is past. In the later case *start*, *stop* might be omitted. If *step* is negative the array is return in reverse order.

```
>>> a = np.arange(12)
>>> a[::1]
[ 0  1  2  3  4  5  6  7  8  9 10 11]
>>> a[::2]
[ 0  2  4  6  8 10]
>>> a[::3]
[0 3 6 9]
>>> a[::4]
[0 4 8]
>>> a[::-1]
[11 10  9  8  7  6  5  4  3  2  1  0]
>>> a[::-2]
[11  9  7  5  3  1]
>>> a[::-3]
[11  8  5  2]
>>> a[::-4]
[11  7  3]
```

Slicing by ndarray It is possible to slice a ndarray by another one. There is only one limitation for the later it values should be in the section `[-len(array), len(array)-1]`. In other words, the past array length may be arbitrary as well as its elements may be in any order (ascending, descending, or repeating).
The resulting array will contain as many elements as the one that was used for indexing.

```
>>> a = np.arange(12)
>>> a
[ 0  1  2  3  4  5  6  7  8  9 10 11]
>>> a[np.array([3,-9, 3,-9, 3,-9, 3,-9, 3,-9, 3,-9, 3,-9])]
array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3])

>>> a[a[::3]]
array([0, 3, 6, 9])

>>> a[a[::3]+1]
array([ 1,  4,  7, 10])
```

Slicing by Boolean ndarray The second approach is to pass the ndarray of N `bool`, where N is is the given array length `len(array)`.
The resulting array length will be equal to the number of `True` elements in the array used for slicing.

```
>>> a = np.arange(12)

>>> a[np.array([1, 0] * 6, np.bool)]
array([ 0,  2,  4,  6,  8, 10])
>>> a[np.array([True, False,True, False,True, False,
                True, False,True, False,True, False])]
array([ 0,  2,  4,  6,  8, 10])
```

9.7 Ndarray Methods

An ndarray object has many methods which operate on or with the array in some fashion, typically returning an array result. All methods are briefly explained in official documentation.

abs(*x*)	Calculate the absolute value element-wise.
sum(*a, axis=None*)	Sum of array elements over a given axis.
count_nonzero(*a, axis=None*)	Counts the number of non-zero values in the array a.
sort(*a, axis=-1*)	Return a sorted copy of an array.
argsort(*a, axis=-1*)	Returns the indices that would sort an array. Perform an indirect sort along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as a that index data along the given axis in sorted order.
diff(*a, n=1, axis=-1*)	Calculate the n-th discrete difference along the given axis.
all(*a, axis=None*)	Test whether all array elements along a given axis evaluate to True.
any(*a, axis=None*)	Test whether any array element along a given axis evaluates to True. Returns single boolean unless axis is not None

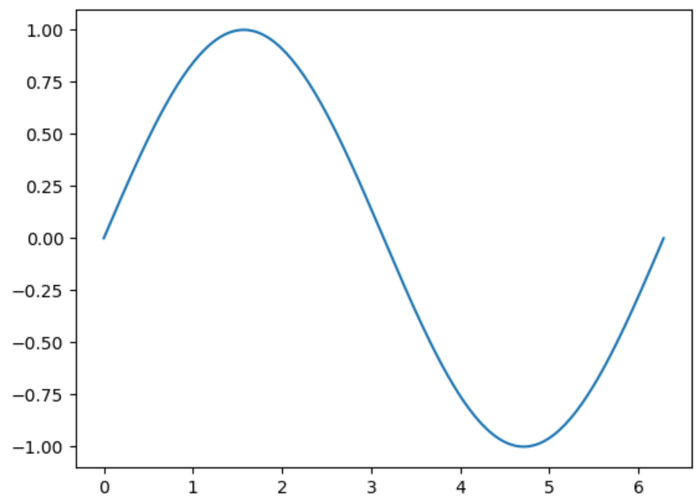
10 Package MatPlotLib

Matplotlib is a plotting library for the **Python** programming language and its numerical mathematics extension **NumPy**. It provides a large number of function for output of two and three dimensional data.

10.1 One-dimensional plots

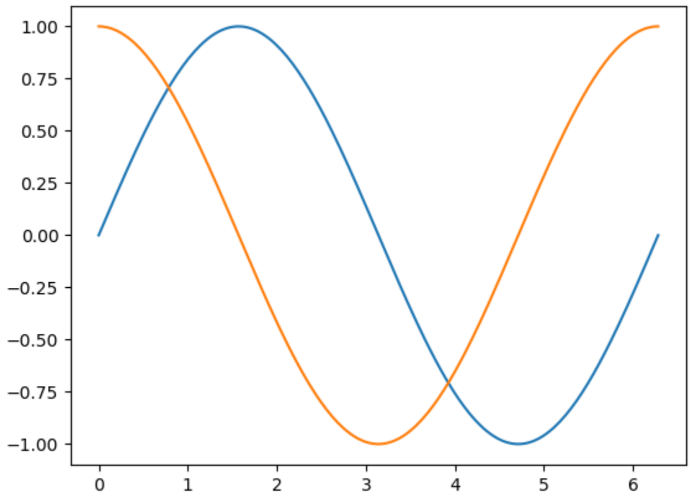
Creating one-dimensional plots are really easy! It requires only few basic step.

```python
import numpy as np                  # numpy import
import matplotlib.pylab as plt      # matplotlib import

arg = np.linspace(0, 2 * np.pi, 200) # prepairing data

plt.plot(arg, np.sin(arg))          # making a plot
plt.show()                          # and showing it
```



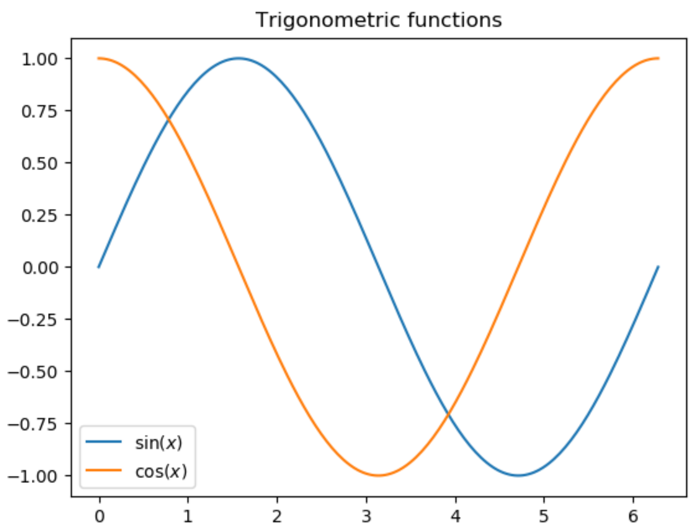
If we would like to have several plots on one figure we just call once more the function **plt.plot()**

```python
import numpy as np
import matplotlib.pylab as plt

arg = np.linspace(0, 2 * np.pi, 200)

plt.plot(arg, np.sin(arg))
plt.plot(arg, np.cos(arg))   # new line
plt.show()
```



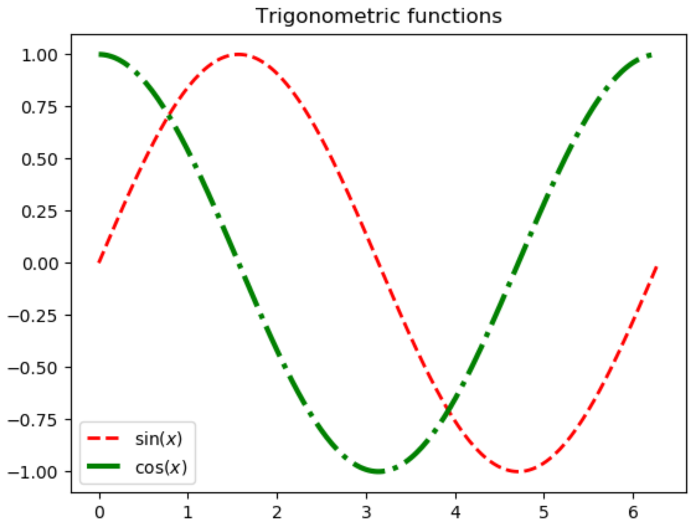
For adding title we just call the function **plt.title()**, for adding labels we modify parameters of the function **plt.plot()** and then call the function **plt.legend()** what is mandatory, otherwise the labels won't be shown.

```
import numpy as np
import matplotlib.pylab as plt

arg = np.linspace(0, 2 * np.pi, 200)

plt.title('Trigonometric functions')
plt.plot(arg, np.sin(arg), label=r'$\sin(x)$')   # modified line
plt.plot(arg, np.cos(arg), label=r'$\cos(x)$')   # modified line
plt.legend()                                     # new line
plt.show()
```



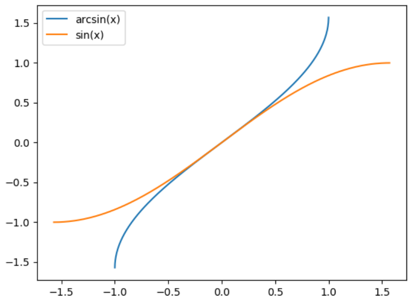
There are also a vast number of operations to polish that plot, nevertheless the last is how to maintain a line style, for it you should just pass appropriate values to the function **plt.plot()** parameters of **c** (color), **lw** (line width),**ls** (line style).

```
import numpy as np
import matplotlib.pylab as plt

arg = np.linspace(0, 2 * np.pi, 200)

plt.title('Trigonometric functions')
plt.plot(arg, np.sin(arg), c='r', lw=2, ls='--',
          label=r'$\sin(x)$') # modified line
plt.plot(arg, np.cos(arg), c='g', lw=3, ls='-.',
         label=r'$\cos(x)$') # modified line
plt.legend()
plt.show()
```



Task:
On one plot make to graphs of a function and its inverse one without calling explicitly the last one i.e.

```
arg = np.linspace(-0.5*np.pi, 0.5*np.pi, 200)
val = np.sin(arg)

plt.plot(arg, val,      label=r'$\sin(x)$')
# add here a line to add arcsin function without calling it
plt.legend()
plt.show()
```



10.2 Image proceeding

MatPlotLib can also proceed image files with extension '.png'. In fact after loading them, MatPlotLib treat them as three-dimensional ndarrays (:,:,3) where the first dimension describes a Y coordinate of the point, the second one does a X coordinate of it. The point with coordinates $(0,0)$ corresponds to the left-top corner. And the third one is array of three float numbers in the segment $[0,1]$. They are components of three colours RED, GREEN, BLUE.

```
import matplotlib.image
import matplotlib.pylab as plt
biedronka = matplotlib.image.imread('biedronka.png')
plt.imshow(biedronka)
plt.show()
```

After creating the ndarray *biedronka* it's easy to get its properties

```
>>> type(biedronka)
<class 'numpy.ndarray'>
>>> biedronka.shape
(450, 600, 3)
```

```
>>> biedronka.dtype
float32
```

It is very easy to construct colour histograms in MatPlotLib, for that we need only, literally, two functions, the first is the NumPy function **np.raven()** that flattens ndarrays and the second one is MatPlotLib **plt.hist()** that builds histograms

```
plt.figure()
plt.hist(biedronka[:, :, 0].ravel(), 256, [0, 1], color='r')
plt.xlim(0,1)
plt.draw()
plt.figure()
plt.hist(biedronka[:, :, 1].ravel(), 256, [0, 1], color='g')
plt.xlim(0,1)
plt.draw()
plt.figure()
plt.hist(biedronka[:, :, 2].ravel(), 256, [0, 1], color='b')
plt.xlim(0,1)
plt.draw()
```

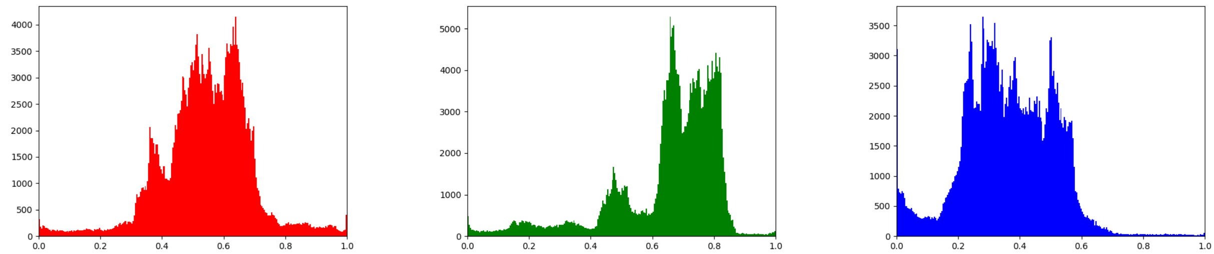


Figure 1: Colour distribution histograms

```
plt.figure()
plt.hist(biedronka[:, :, 0].ravel(), 256, [0, 1],
         alpha=0.5, color='r')
plt.hist(biedronka[:, :, 1].ravel(), 256, [0, 1],
         alpha=0.5, color='g')
plt.hist(biedronka[:, :, 2].ravel(), 256, [0, 1],
         alpha=0.5, color='b')
plt.xlim(0, 1)
plt.show()
```

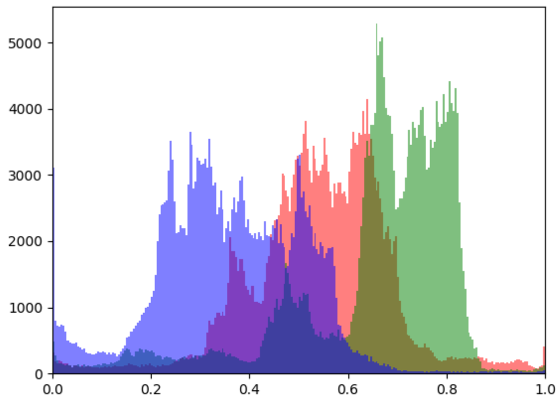


Figure 2: Colour distribution histograms

```
plt.figure()
biedronka[:biedronka.shape[0] // 2,
          :biedronka.shape[1] // 2, :] = 0
biedronka[biedronka.shape[0] // 2:,
          biedronka.shape[1] // 2:, :] = 1
plt.imshow(biedronka)

plt.show()
```

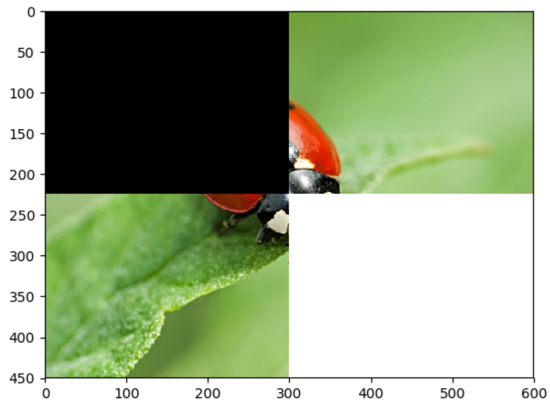
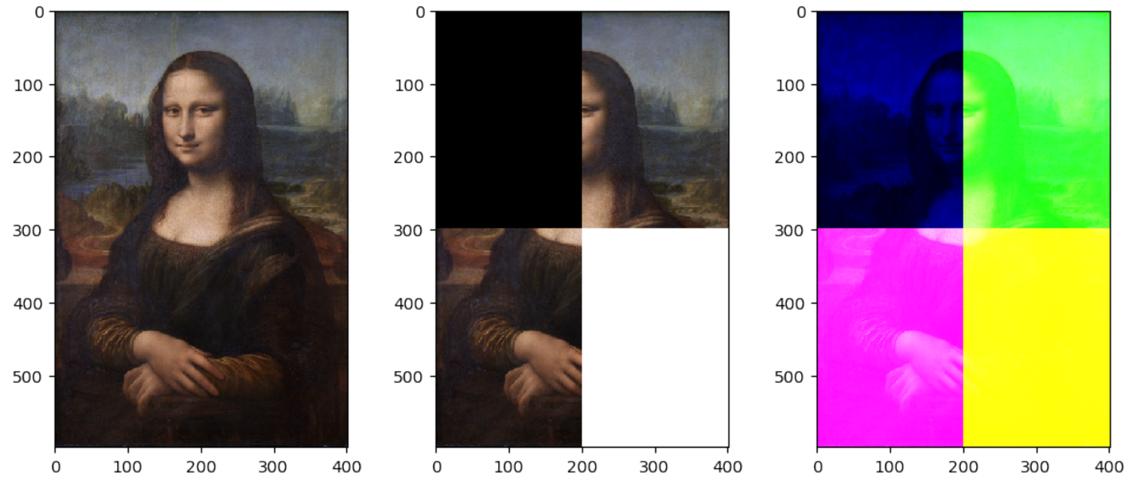


Figure 3: Colour distribution histograms

If the imported '.png' file has the RGBA format then the resulting array will have dimension of the shape (:,:,4) where besides of three colour components RED, GREEN, BLUE there is the alpha channel that indicates how opaque each pixel is.

```
monalisa = matplotlib.image.imread('monalisa.png')

plt.figure()
plt.imshow(monalisa)
plt.draw()

ml = monalisa.copy()  # first way of copying
plt.figure()
ml[:ml.shape[0] // 2, :ml.shape[1] // 2, :-1] = 0
ml[ml.shape[0] // 2:, ml.shape[1] // 2:, :-1] = 1
plt.imshow(ml)
plt.draw()

ml = monalisa[...]  # another way of array copying
                    # actually the same as ml = monalisa[:,:,:]
plt.figure()
ml[:ml.shape[0] // 2, :ml.shape[1] // 2, [0, 1]] = 0
ml[ml.shape[0] // 2:, :ml.shape[1] // 2, [0, 2]] = 1
ml[:ml.shape[0] // 2, ml.shape[1] // 2:, [1]] = 1
ml[ml.shape[0] // 2:, ml.shape[1] // 2:, [0, 1]] = 1
plt.imshow(ml)
plt.draw()

plt.show()
```



Task:
Make and draw a grayscale image from given coloured one.
Hint:
In a grayscale image all three colour components (red, green, blue) are equal.

Task:
Using only ndarray indexing transform the original picture to the following

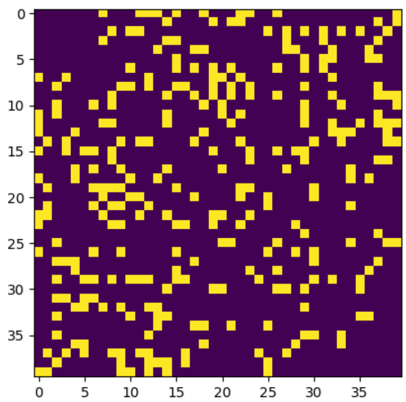
10.3 Two-dimensional plots

MatPlotLib has many approaches for constructing two-dimensional plots. The most basic is the function

plt.imshow*(X)*

that displays values ndarray X. It was used previously for displaying PNG-files. However it also suitable for plotting any other array.

```
import matplotlib.pylab as plt
import numpy as np
import networkx as nx
import matplotlib.cm

G = nx.binomial_graph(n=40, p=0.2)
plt.figure('Graph')
nx.draw_circular(G, node_color=range(len(G)),
                 cmap=matplotlib.cm.jet, with_labels=True)
plt.figure('Graph Matrix')
plt.imshow(nx.to_numpy_matrix(G))

plt.show()
```



Task:
Build and display a dissimilarity matrix of a random graph using the **NetworkX** function shortest_path.
Please note, the last function returns error if two nodes are not connected, i.e. lie in different connected components.