

---

# Introduction to Numerical Python

---

Volodymyr Byelobrov



# Contents

1	Preface . . . . .	3
2	Introduction to Python . . . . .	4
2.1	Basic information . . . . .	4
2.2	Comments . . . . .	6
2.3	Arithmetic operators . . . . .	7
2.4	Comparison operators . . . . .	8
2.5	Assignment operators . . . . .	8
2.6	Indention . . . . .	8
2.7	Basic types . . . . .	9
3	IF statement . . . . .	10
4	LOOPS . . . . .	11
4.1	FOR statement . . . . .	11
4.2	WHILE statement . . . . .	11
4.3	BREAK and CONTINUE . . . . .	11
4.4	Function RANGE . . . . .	12
5	Python: Strings . . . . .	13
5.1	Formatting . . . . .	13
5.2	Operations . . . . .	13
5.3	Methods . . . . .	14
5.4	Special characters . . . . .	14
5.5	Regular Expression . . . . .	15
6	Python: Containers . . . . .	17
6.1	Lists . . . . .	17
6.2	Sets . . . . .	18
6.3	Tuples . . . . .	20
6.4	Dictionaries . . . . .	20
6.5	Containers overview . . . . .	22
7	Python: Functions . . . . .	24
7.1	Function Declaration . . . . .	24
7.2	Asterisk Operator * . . . . .	25
7.3	Anonymous function . . . . .	26
7.4	Recursion . . . . .	27
7.5	Function callable . . . . .	27
7.6	Function isinstance . . . . .	28
7.7	Decorators, Enclosures, Nested functions . . . . .	28
8	Package NumPy . . . . .	32
8.1	Introduction . . . . .	32
8.2	ndarrays basic . . . . .	34
8.3	ufunc . . . . .	38

	8.4	Broadcasting . . . . .	40
	8.5	Indexing of 1D ndarrays . . . . .	41
	8.6	Slicing of 1D ndarrays . . . . .	42
9		Package Matplotlib . . . . .	44
	9.1	One-dimensional plots . . . . .	44
	9.2	Image proceeding . . . . .	47
	9.3	Two-dimensional plots . . . . .	52

# 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 admirers, 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

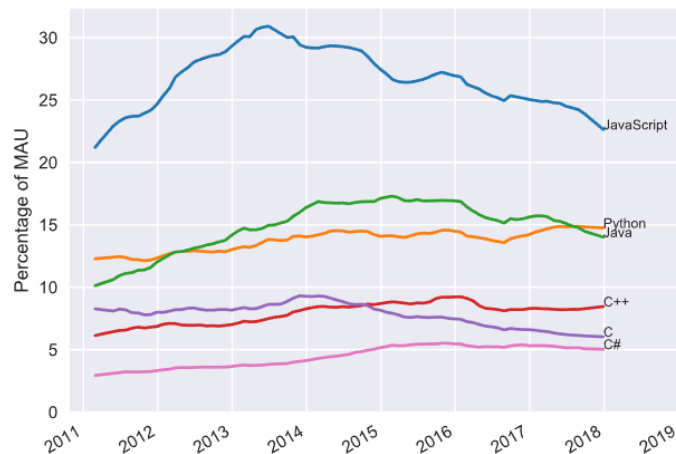
**Python** is an interpreted high-level programming language for general-purpose programming. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming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

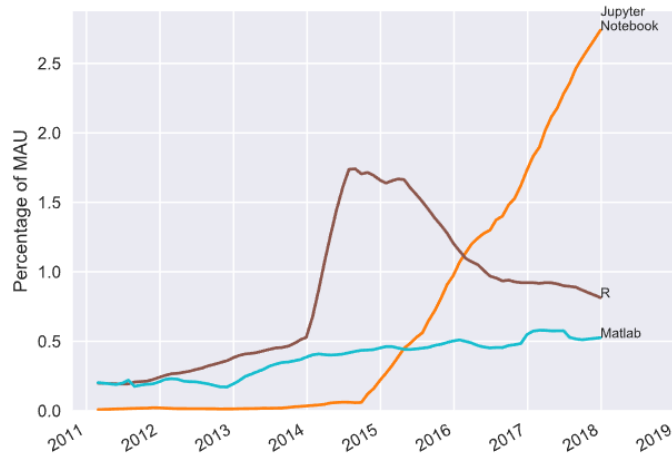
#### Major Languages

The major programming languages have relatively stable usage, and are mostly what you'd expect:



#### Scientific Languages

There was one other fast-growing 'language' included in the results that I purposefully left out:



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 famous 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 passed as a function parameter. Every object has its **id** and **type** that could be yielded by functions `id()` and `type()`.

```

1 >>> a = 2
2 >>> id(2)
3 140723092632432
4 >>> id(a)
5 140723092632432
6
7 >>> a = 3
8 >>> id(a)
9 140723092632464
10
11 >>> type(3)
12 <class 'int'>
13
14 >>> type(3j)
15 <class 'complex'>
16
17 >>> type('3j')
18 <class 'str'>

```

## 2.2 Comments

Comments are brilliant way to increase code 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 indentation, 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

```

1 # inline comment may be from the start of line
2 # or comment the line they follow
3 a = 5 # assign the variable a to 5
4 for i in range(10):
5     print(i) # or here
6 '''
7 block comments may contain
8 any number of lines
9 '''
10
11 """
12 or like this
13 """

```

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

```
1 >>> def square(n):
2     """
3     the square function
4     :param n: int, number
5     :return: n square
6     """
7     return n ** 2
8 >>> square(2)
9 4
10 >>> square.__doc__
11 the square function\n
12 :param n: int, number\n
13 :return: n square\n
```

## 2.3 Arithmetic operators

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



## 2.4 Comparison operators

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

## 2.5 Assignment operators

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

## 2.6 Indention

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

```

1 def square(x):
2     return x*x
3
4 def _max(a,b):
5     if a>b:
6         return a
7     else:
8         return b
9
10 for i in range(8):
11     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.

```
1 a, b = 3, 4
2 _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.

```
1 if a > b and a > c:
2     _max3 = a
3 elif b > c:
4     _max3 = b
5 else:
6     _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().

```
1 >>> for i in range(10):
2 >>>     print(i, end=' ')
3 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

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

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

```

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

```

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

```
1 >>> s = 'Hello World!'
2 >>> s
3 'Hello World!'
4 >>> len(s)
5 12
6 >>> s[0]
7 'H'
8 >>> s[-1]
9 '!'
10 >>> s[::-1]
11 '!dlroW olleH'
12 >>> s[::2]
13 'HloWrld'
```

### 5.1 Formatting

```
1 >>> a, b, c = 2, 3, 4
2 >>> '{}x^2 + {}x + {} = 0'.format(a,b,c)
3 '2x^2 + 3x + 4 = 0'
4 >>> f'{a}x^2 + {b}x + {c} = 0'
5 '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

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

## 5.4 Special characters

<code>\b</code>	ASCII Backspace (BS) character
<code>\n</code>	ASCII Linefeed (LF) character
<code>\t</code>	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 an expression that will perform the same work as `str.zfill()`

### Task 3

Suppose you have a string that represents a mathematical statement with two kinds of parentheses `()` and `{}`. Write a function that will verify mathematical correctness of this statement, e.g. `'3 * (y / {(x - 1) * (6 + z)})'` is correct and `'3 * (x - 1 + z)'` is not.

## 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.

```
import re
```

Besides Python you may verify your patterns on following sites:

- [pyregex.com](http://pyregex.com)
- [regexr.com](http://regexr.com)
- [regex101.com](http://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
{ }	braces	{m} the preceding item is matched exactly <i>m</i> times
		{m,} the preceding item is matched <i>m</i> or more times
		{m,n} matched at least <i>m</i> times, but not more than <i>n</i> 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.



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

***findall(pattern, string)***

***split(pattern, string)***

***sub(pattern,replace, string)***

***search(pattern, string)***

***match(pattern, string)***

returns a list of strings containing all matches

splits the string where there is a match and returns a list of strings where the splits have occurred

returns a string where matched occurrences are replaced with the content of replace variable

looks for the first location where RegEx pattern produces a match with the string. If the search is successful, `re.search()` returns a **match object**; if not, it returns **None**

only checks if the RegEx matches at the beginning of the string, as before `re.match()` returns a **match object**; if not, it returns **None**

```
1 >>> import re
2 >>> re.findall('\d+', 'DF 168 BG')
3 ['168']
4 >>> re.findall('[A-Z]{2}', 'DF 168 BG')
5 ['DF', 'BG']
```

```
1 >>> import re
2 >>> re.search(r'7{3}', str(2 ** 24))
3 <re.Match object; span=(2, 5), match='777'>
```

### Match object

<code>match.group()</code>	returns the part of the string where there is a match
<code>match.start()</code>	returns the index of the start of the matched substring
<code>match.end()</code>	returns the end index of the matched substring
<code>match.span()</code>	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.

```
1 a = [1,2,3,4,5]           # assign
2 len(a) == 5               # get the length
3 a[0] == a[-5] == 1       # get an item
4 a[-1] == a[4] == 5
5 del a[2]; a == [1,2,4,5]  # remove an item
6 a[2] = 'abc'; a == [1,2,'abc',5] # assign an item
7 a[1] = ['a','b','c']     # assign an item
8 a == [1,['a','b','c'],12,13,'abc',5]
9 a[1:2] = [11,12,13]; a == [1,11,12,13,'abc',5] # weird
10 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.

```
1 a =[0,1,2,3,4,5]
2 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

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

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


## 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

```
1 set1 = set()
```

Assigning set

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

Comprehension

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

Equivalent block statement for set creation

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

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

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

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

<b><i>add(x)</i></b>	add element x to the set
<b><i>remove(x)</i></b>	remove x from the set; raises <code>KeyError</code> if not present
<b><i>discard(x)</i></b>	removes x from the set if present
<b><i>pop()</i></b>	remove and return an arbitrary element from the set; raises <code>KeyError</code> if empty
<b><i>clear()</i></b>	remove all elements from the set

### Task

Sets are collection of **immutable** objects. However could you find an 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

```
1 tuple1 = ()
2 tuple2 = tuple()
```

Assigning tuple

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

Equivalent block statement for tuple creation Comprehension

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

Block statement

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

```
1 dict1 = dict()  
2 dict2 = {}
```

Assigning dictionary

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

Comprehension

```
1 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.

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

```
1 A = {i: i ** 2 for i in range(10) if i % 2 == 0}  
2 A.get(2) == A[2] == 4  
3 A.items() == dict_items([(0, 0), (2, 4), (4, 16), (6, 36), (8, 64)])  
4 A.keys() == dict_keys([0, 2, 4, 6, 8])  
5 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
<b>list</b>	[i for i in range(5)]	[0, 1, 2, 3, 4]
<b>tuple</b>	tuple(i for i in range(5))	(0, 1, 2, 3, 4)
<b>set</b>	{i for i in range(5)}	{0, 1, 2, 3, 4}
<b>dict</b>	{i:i**2 for i in range(5)}	{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
<b>str</b>	".join([str(i) for i in range(5)])	'01234'

An iterator performs traversal and gives access to data elements in a container, but does not perform iteration. You might be confused so let's take it a bit slow. There are three parts namely:

- An **iterable** is any object in Python which has an `__iter__` or a `__getitem__` method defined which returns an iterator or can take indexes. In short an iterable is any object which can provide us with an iterator.
- An **iterator** is any object in Python which has a `__next__` method defined. That's an iterator.
- **Iteration** is the process of taking an item from something e.g a list. When we use a loop to loop over something it is called iteration. It is the name given to the process itself.

By **index** (lists and tuples)

```
1 lst = [1, 2, 3, 4]
2 for index in range(len(lst)):
3     print(lst[index], end=' ')
```

By **element** (lists, tuples, and sets)

```
1 set1 = {1, 2, 3, 4}
2 for s in set1:
3     print(s, end=' ')
```

By **index** and **element** (lists, tuples, and sets)

```
1 tuple1 = tuple([1, 2, 3, 4])
2 for i, s in enumerate(tuple1):
3     print((i, s), end=' ')
```

### Task

- What is the difference between applying enumerate to **list**, **tuple**, and **set**?
- Why you can't iterate sets by indices but can call the **enumerate()** function?

The **zip()** function take iterables (can be zero or more), makes iterator that aggregates elements based on the iterables passed, and returns an iterator of tuples. Using the zip function it is easy to merge several iterable objects.

```
1 vals = [1, 2, 3, 4]
2 keys = ['a', 'b', 'c', 'd']
3 dic1 = {k: v for k, v in zip(keys, vals)}
4 dic1 == {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

### Task

Write your own function using **zip()** which will work as **enumerate()**.



## 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.

```
1 def function_name(a, b=1, c=None, *args): #parameters
2     '''
3     optional description of the function
4     :param a: description of parameters
5     :return : return description
6     '''
7     result = a+b if c is None else a+b+c
8     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.

```
1 >>> new_print = print
2 >>> new_print('Hello World!')
3 Hello World!
```

```
1 def new_sum(a, b=1, c=None): # parameters
2     return a + b if c is None else a + b + c
3
4 val = new_sum(1)
5 print('val:\t', val)
6 val2 = new_sum(a=2)
7 print('val2:\t', val2)
8 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**

```

1 >>> def new_sum(a, b=1, c=None): # parameters
2 >>>     result = a + b if c is None else a + b + c
3 >>>     return result
4
5 >>> new_sum(1)
6 2
7 >>> new_sum(a=2)
8 3
9 >>> new_sum(3, b=2)
10 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**.

```

1 >>> def increase(n):
2 >>>     n += 3
3 >>>     return n
4 >>> var = 5
5 >>> print('var before:', var)
6 var before: 5
7 >>> increase(var)
8 >>> print('var after:', var)
9 var after: 5

```

```

1 >>> def append(lst):
2 >>>     lst.append(3)
3 >>>     return lst
4 >>> l = [1, 2]
5 >>> print('before:', l)
6 before: [1, 2]
7 >>> append(_list)
8 >>> print('after:', l)
9 after: [1, 2, 3]

```

## 7.2 Asterisk Operator \*

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

```

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

```

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

```

1 def my_sum(a, b, c):
2     return a + b + c
3 lst = [1, 2, 3]
4 print(my_sum(*lst))
5 dic = {'a': 1, 'b': 2, 'c': 3}
6 print(my_sum(**dic))
7 lst = [1, 2, 3, 4]
8 print(my_sum(*lst)) # error
9 dic = {'a': 10, 'b': 20, 'c': 30, 'd': 4}
10 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.

```

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

```

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

```

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

```

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

```

1 a = [3, 6, 2, 8, 3]
2 sorted(a) == [2, 3, 3, 6, 8]
3
4 b = [4, -2, 1, -12, -4, -3]
5 sorted(b, key=lambda x: abs(x)) == [1, -2, -3, 4, -4, -12]
6
7 c=('Mrs Dalloway said she would buy'\
8 + 'the flowers herself').split(' ')
9 sorted(c, key=lambda x: x[1]) == ['Dalloway', 'said', \
10 '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?

```

1 >>> def increase_by(n):
2 >>>     return lambda x: x + n

```

```

3
4 >>> func_list = [increase_by(n) for n in range(10)]
5 >>> [f(1) for f in func_list]
6 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
7
8 >>> func_list1 = [lambda x: x + n for n in range(10)]
9 >>> [f(1) for f in func_list1]
10 [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.

```

1 def fibonacci(n):
2     if n == 0:
3         return 0
4     elif n == 1:
5         return 1
6     else:
7         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

```

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

```

type	Iterable	type	Iterable
<b>int</b>	False	<b>bool</b>	False
<b>float</b>	False	<b>complex</b>	False
<b>list</b>	True	<b>tuple</b>	True
<b>str</b>	True	<b>dict</b>	True
<b>set</b>	True	<b>frozenset</b>	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.

```

1 callable(print) == True
2 callable(print()) == False
3 callable(int) == True
4 callable(4) == False
5 callable(list) == True
6 callable([]) == False

```

## 7.6 Function isinstance

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

```

1 import numbers
2
3 isinstance(2, int) == True
4 isinstance(2., int) == False
5 isinstance(2., float) == True
6 isinstance(2, float) == False
7 isinstance(2j, complex) == True
8 isinstance(2j, int) == False
9 isinstance(3+2j, numbers.Number) == True
10 isinstance(3+2j, (int, float, complex)) == True
11 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`.

```

1 >>> var = '4'
2
3 >>> assert isinstance(var, int), f'object var has\
4                               type {type(var)} and value {var}'
5 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.

```

1 def list_power(lst):
2     # enclosing function
3     power = 2
4     print('enclosing function scope', dir())
5
6     def lst_iteration():
7         # nested function
8         print('nested function scope', dir())
9         nonlocal power
10        power += 1
11        print([item ** power for item in lst])
12
13    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.

```

1 def list_power(lst):
2     # enclosing function
3     power = 2
4
5     def lst_iteration():
6         # nested function
7         return [item ** power for item in lst]
8
9     return lst_iteration
10
11 if __name__ == '__main__':
12     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.

```

1 import datetime
2
3 def time_it(func):
4     t1 = datetime.datetime.now()
5
6     def func_wrapper(*args, **kwargs):
7         return func(*args, **kwargs)
8
9     t2 = datetime.datetime.now()
10    print((t2 - t1).microseconds, 'ms')
11    return func_wrapper
12
13
14 if __name__ == '__main__':
15     import numpy as np
16     import random
17     N = 10 ** 7
18     array = [random.random() for i in range(N)]
19     np_array = np.array(array)
20
21     @time_it
22     def python_max(ar):
23         return max(ar)

```

```

24
25
26 @time_it
27 def numpy_max(ar):
28     return np.max(ar)
29
30 print(python_max(array))
31 print(numpy_max(np_array))

```

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

```

1 import datetime
2
3 def time_it(name):
4     def named_function(func):
5         t1 = datetime.datetime.now()
6
7         def func_wrapper(*args, **kwargs):
8             return func(*args, **kwargs)
9
10        t2 = datetime.datetime.now()
11        print(name, (t2 - t1).microseconds, 'ms')
12        return func_wrapper
13    return named_function
14
15
16 if __name__ == '__main__':
17     import numpy as np
18     import random
19
20     N = 10 ** 5
21
22     array = [random.random() for i in range(N)]
23     np_array = np.array(array)
24
25     @time_it('Python execution time')
26     def python_max(ar):
27         return max(ar)
28
29     @time_it('Numpy execution time')
30     def numpy_max(ar):
31         return np.max(ar)
32
33     print(python_max(array))
34     print(numpy_max(np_array))

```

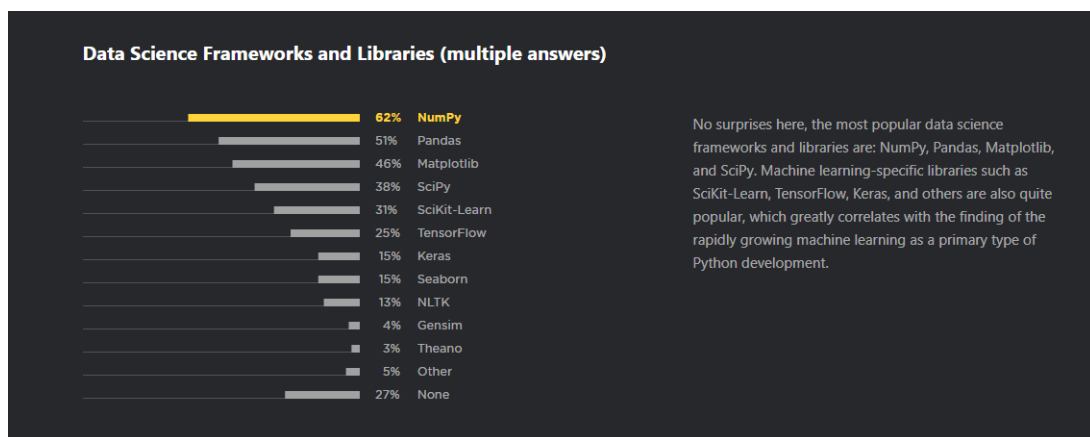


## 8 Package NumPy

### 8.1 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:



```
1 >>> import numpy as np
2 >>> np.__version__
3 '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.

```
1 >>> from numpy import *
2 >>> pi
3 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.

```
1 >>> np.iinfo(np.int16)
2 iinfo(min=-32768, max=32767, dtype=int16)
3
4 >>> np.iinfo(np.int32)
```

```

5 iinfo(min=-2147483648, max=2147483647, dtype=int32)
6
7 >>> np.finfo(np.float16)
8 finfo(resolution=0.001, min=-6.55040e+04, \
9         max=6.55040e+04, dtype=float16)

```

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

```

1 >>> 1/np.float32(0)
2 inf
3
4 >>> np.log(0)
5 -inf
6
7 >>> 1/np.inf
8 0
9
10 >>> np.inf + 3
11 inf
12
13 >>> np.inf * 3
14 inf
15
16 >>> np.inf + np.inf
17 inf
18
19 >>> np.log(np.inf)
20 inf

```

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

```

1 >>> np.sqrt(-1)
2 nan
3
4 >>> np.sin(np.inf)
5 nan
6
7 >>> np.nan + 2
8 nan
9
10 >>> np.inf - np.inf
11 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:

```

1 >>> np.sqrt(np.complex(-1))
2 1j
3 >>> np.sqrt(-1 + 0j)
4 1j

```

## 8.2 ndarrays basic

NumPy **ndarrays** 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

```

1 >>> np.ndarray(shape=(3,), dtype=np.float64)
2 array([0.00000000e+000, 0.00000000e+000, 2.47823328e-320])
3
4 >>> np.ndarray(shape=(4, 3), dtype=np.bool)
5 array([[ True,  True,  True],
6        [ True,  True,  True],
7        [False, False,  True],
8        [ True,  True,  True]])
9
10 >>> np.ndarray(shape=(1, 4), dtype=np.complex128)
11 array([[0.00000000e+000+0.00000000e+000j,

```

```

12         0.00000000e+000+0.00000000e+000j,
13         0.00000000e+000+2.47823328e-320j,
14         3.57360117e-306+3.57499184e-306j]])
15
16 >>> array = np.ndarray(shape=(2, 3), dtype=np.int)
17 >>> array
18 array([[          0,           0, -1253916128],
19        [         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

```

1 >>> array.dtype
2 dtype('float64')
3 >>> array.flags
4 C_CONTIGUOUS : True
5 F_CONTIGUOUS : False
6 OWNDATA : True
7 WRITEABLE : True
8 ALIGNED : True
9 WRITEBACKIFCOPY : False
10 UPDATEIFCOPY : False
11 >>> array.size
12 6 # == array.shape[0] * array.shape[1]
13 >>> array.shape
14 (2, 3)
15 >>> array.ndim
16 2 # == len(array.shape[0])

```

As you see, the array after using the 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 elements 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

```
1 >>> array.fill(1)
2 >>> array
3 array([[1, 1, 1],
4        [1, 1, 1]])
5
6 >>> array[:, :] = 2
7 >>> array
8 array([[2, 2, 2],
9        [2, 2, 2]])
10
11 >>> array[...] = 3
12 >>> array
13 array([[3, 3, 3],
14        [3, 3, 3]])
```

One should be 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.

```
1 >>> ar1 = np.ndarray(shape=(2,3), dtype=np.int)
2 >>> ar1
3 array([[ 3604533,  3407920, 1870623728],
4        [    462,         0, -2147483648]])
5 >>> ar1[...] = 1
6 >>> ar1
7 array([[1, 1, 1],
8        [1, 1, 1]])
9 >>> ar2 = np.ndarray(shape=(2,3), dtype=np.int)
10 >>> ar2
11 array([[1, 1, 1],
12        [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)
```

```
1 >>> np.array([[1,2,3], [2,3,4], [3,4,5]])
2 array([[1, 2, 3],
3        [2, 3, 4],
4        [3, 4, 5]])
5
6 >>> np.zeros(shape=(2, 3))
7 array([[0., 0., 0.],
8        [0., 0., 0.]])
9
10 >>> np.ones(shape=(3, 2), dtype=np.complex128)
11 array([[1.+0.j, 1.+0.j],
12        [1.+0.j, 1.+0.j],
13        [1.+0.j, 1.+0.j]])
14
15 >>> np.full((2,2), True, dtype=np.bool)
16 array([[ True,  True],
17        [ True,  True]])
18
19 >>> np.eye(N=3, M=4, dtype=np.complex128)
20 array([[1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
21        [0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
22        [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 with values from 0 till `n-1`, and then manipulating the array shape by the function `reshape()`.

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

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

```

1 import numpy as np
2 >>> np.random.randn(3,4)
3 array([[ 0.96055287,  1.05930852, -1.02454838, -0.27037388],
4        [ 0.17874319, -0.77100283,  0.85711132, -0.46897319],
5        [-1.45428268,  0.19297873, -0.15411337, -0.54117629]])
6
7 >>> np.random.randint(10, size=(3,4))
8 array([[7, 2, 7, 3],
9        [4, 8, 6, 8],
10       [7, 6, 0, 6]])

```

### 8.3 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: + − ∗ @ / // % ∗∗
- Bitwise Operations: & | ~ ^ >> <<
- Comparison Operations: < > <= => == !=
- 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.

```

1 >>> a = np.array([2, 4, 6, 8]).reshape(2, 2)
2 >>> a
3 array([[2, 4],
4        [6, 8]])
5 >>> a + 1
6 array([[3, 5],
7        [7, 9]])
8 >>> a - 5
9 array([[ -3,  -1],
10       [ 1,   3]])
11 >>> a * 2
12 array([[ 4,  8],
13       [12, 16]])
14 >>> a / 2
15 array([[1., 2.],
16       [3., 4.]])
17 >>> a % 3
18 array([[2, 1],
19       [0, 2]], dtype=int32)
20 >>> a // 3

```

```

21 array([[0, 1],
22        [2, 2]], dtype=int32)
23 >>> a / 3
24 array([[0.66666667, 1.33333333],
25        [2.        , 2.66666667]])
26 >>> a ** 2
27 array([[ 4, 16],
28        [36, 64]], dtype=int32)

```

Generally, all arithmetic operations are implemented for two **ndarrays**, 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)***

```

1 >>> a = np.array([10,20,30,40]).reshape(2,2)
2 >>> a
3 array([[10, 20],
4        [30, 40]])
5 >>> b = np.array([2,4,6,8]).reshape(2,2)
6 >>> b
7 array([[2, 4],
8        [6, 8]])
9 >>> a + b
10 array([[12, 24],
11         [36, 48]])
12 >>> a - b
13 array([[ 8, 16],
14         [24, 32]])
15 >>> a * b    # elementwise multiplication
16 array([[ 20,  80],
17         [180, 320]])
18 >>> a @ b    # matrix product
19 array([[140, 200],
20         [300, 440]])
21 >>> a / b
22 array([[5., 5.],
23         [5., 5.]])
24 >>> b % a
25 array([[2, 4],
26         [6, 8]], dtype=int32)

```

## Comparison Operations

```

1 >>> a = np.arange(10,100,10).reshape(3,3)
2 >>> a
3 array([[10, 20, 30],
4        [40, 50, 60],
5        [70, 80, 90]])

```



```

6 >>> a > 40
7 array([[False, False, False],
8        [False,  True,  True],
9        [ True,  True,  True]])
10 >>> a % 20 == 0           # here two operation performed
11 array([[False,  True, False],
12        [ True, False,  True],
13        [False,  True, False]])
14 >>> a == np.array([10, 50, 90])    # dimensions don't
15 array([[ True, False, False],      # coincide
16        [False,  True, False],      # broadcasting
17        [False, False,  True]])     # is used

```

## Mathematical Functions

```

1 >>> a = np.array([0, 30, 45, 60])
2 >>> np.sin(a * np.pi / 180)
3 array([0.          , 0.5          , 0.70710678, 0.8660254 ])
4
5 >>> a = np.arange(0, 90, 10).reshape(3, 3)
6 >>> np.abs(np.exp(1j * a * np.pi / 180))
7 array([[1., 1., 1.],
8        [1., 1., 1.],
9        [1., 1., 1.]])

```

## 8.4 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.

```

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

```

```

10 array([[ 0,  2,  4,  6],
11        [ 4,  6,  8, 10],
12        [ 8, 10, 12, 14]])

```

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

```

1 >>> b.shape
2 (4,)

```

```

1 >>> _b.shape
2 (3, 4)

```

```

1 >>> a = np.arange(3).reshape((3, 1))
2 >>> b = np.arange(3)
3 >>> a+b
4 array([[0, 1, 2],
5        [1, 2, 3],
6        [2, 3, 4]])

```

## 8.5 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 applicable 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*.

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

```

1 >>> a = np.arange(12)
2 >>> a[0]
3 0
4 >>> a[3]
5 3
6 >>> a[11]
7 11
8 >>> a[-1]      # the last element
9 11
10 >>> a[-3]
11 9
12 >>> a[-12]
13 0

```

## 8.6 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.

```
1 >>> a = np.arange(12)
2 >>> a
3 [ 0  1  2  3  4  5  6  7  8  9 10 11]
4 >>> a[3:]
5 [ 3  4  5  6  7  8  9 10 11]
6 >>> a[-9:]
7 [ 3  4  5  6  7  8  9 10 11]
8 >>> a[:9]
9 [0 1 2 3 4 5 6 7 8]
10 >>> a[:-3]
11 [0 1 2 3 4 5 6 7 8]
12 >>> a[3:9]
13 [3 4 5 6 7 8]
```

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

```
1 >>> a = np.arange(12)
2 >>> np.all(np.concatenate((a[:5], a[5:])) == a)
3 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.

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

```

8 >>> a[::4]
9 [0 4 8]
10 >>> a[::-1]
11 [11 10 9 8 7 6 5 4 3 2 1 0]
12 >>> a[::-2]
13 [11 9 7 5 3 1]
14 >>> a[::-3]
15 [11 8 5 2]
16 >>> a[::-4]
17 [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.

```

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

```

**Slicing by Boolean ndarray** The second approach is to pass the ndarray of  $N$  **bool**, where  $N$  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.

```

1 >>> a = np.arange(12)
2
3 >>> a[np.array([1, 0] * 6, np.bool)]
4 array([ 0,  2,  4,  6,  8, 10])
5 >>> a[np.array([True, False, True, False, True, False,
6                 True, False, True, False, True, False])]
7 array([ 0,  2,  4,  6,  8, 10])

```

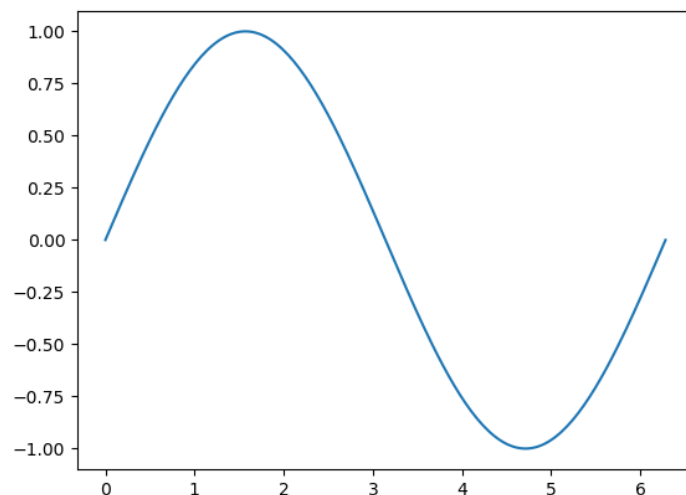
## 9 Package Matplotlib

**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.

### 9.1 One-dimensional plots

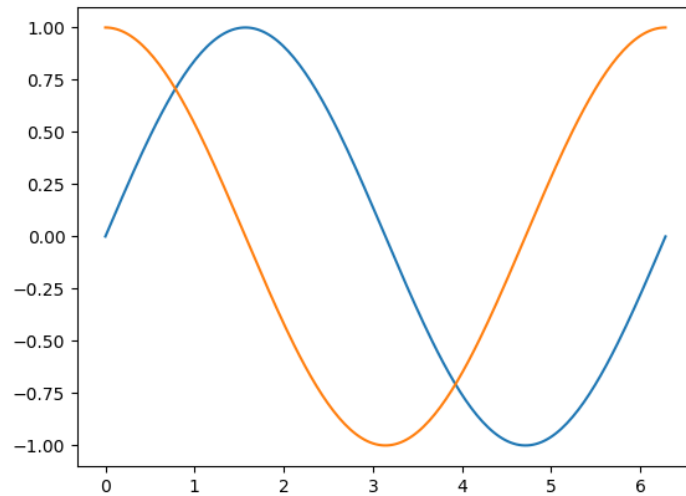
Creating one-dimensional plots are really easy! It requires only few basic step.

```
1 import numpy as np          # numpy import
2 import matplotlib.pyplot as plt # matplotlib import
3
4 arg = np.linspace(0, 2 * np.pi, 200) # preparing data
5
6 plt.plot(arg, np.sin(arg))      # making a plot
7 plt.show()                     # and showing it
```



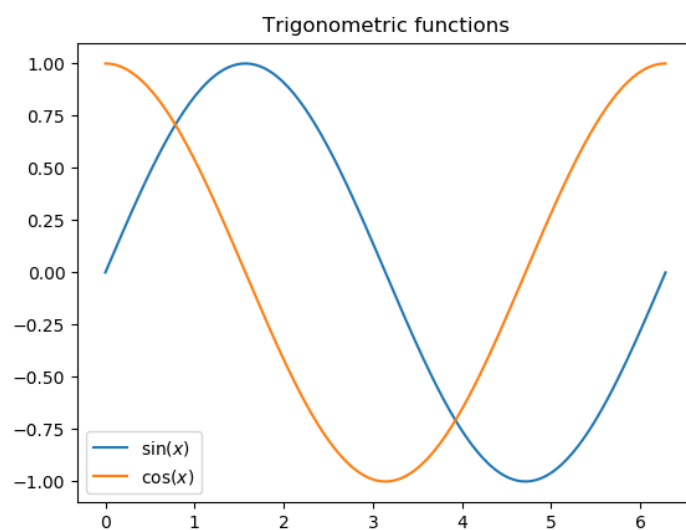
If we would like to have several plots on one figure we just call once more the function **plt.plot()**

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 arg = np.linspace(0, 2 * np.pi, 200)
5
6 plt.plot(arg, np.sin(arg))
7 plt.plot(arg, np.cos(arg)) # new line
8 plt.show()
```



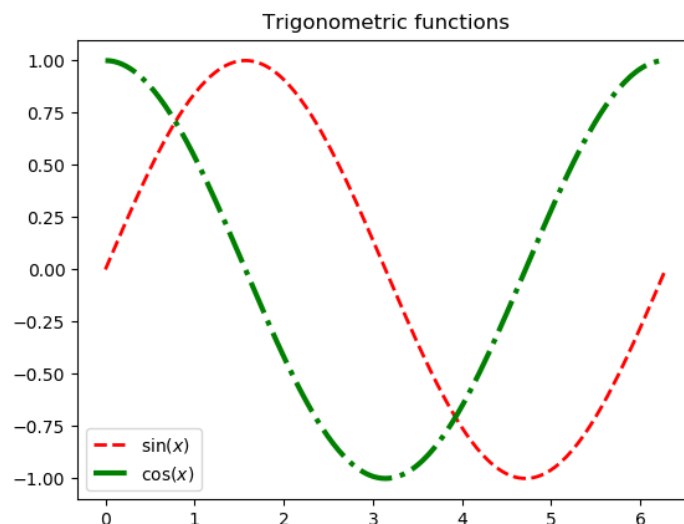
For adding title we just call the function **plt.title()**, for adding labels we modify parameters of the function **plt.plot()** and then call the function **plt.legend()** what is mandatory, otherwise the labels won't be shown.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 arg = np.linspace(0, 2 * np.pi, 200)
5
6 plt.title('Trigonometric functions')
7 plt.plot(arg, np.sin(arg), label=r'$\sin(x)$') # modified line
8 plt.plot(arg, np.cos(arg), label=r'$\cos(x)$') # modified line
9 plt.legend() # new line
10 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).

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 arg = np.linspace(0, 2 * np.pi, 200)
5
6 plt.title('Trigonometric functions')
7 plt.plot(arg, np.sin(arg), c='r', lw=2, ls='--',
8          label=r'$\sin(x)$') # modified line
9 plt.plot(arg, np.cos(arg), c='g', lw=3, ls='-.',
10          label=r'$\cos(x)$') # modified line
11 plt.legend()
12 plt.show()
```

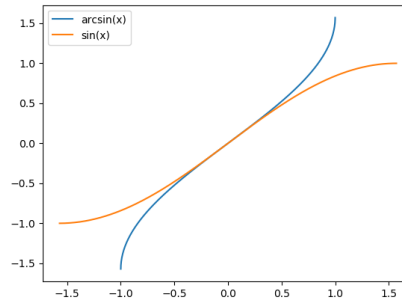


### Task:

On one plot make two 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()
```



## 9.2 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**.

```
1 import matplotlib.image
2 import matplotlib.pyplot as plt
3 biedronka = matplotlib.image.imread('biedronka.png')
4 plt.imshow(biedronka)
5 plt.show()
```



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

```
1 >>> type(biedronka)
2 <class 'numpy.ndarray'>
3 >>> biedronka.shape
```



```

4 (450, 600, 3)
5 >>> biedronka.dtype
6 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.ravel()** that flattens ndarrays and the second one is Matplotlib **plt.hist()** that builds histograms

```

1 plt.figure()
2 plt.hist(biedronka[:, :, 0].ravel(), 256, [0, 1], color='r')
3 plt.xlim(0,1)
4 plt.draw()
5 plt.figure()
6 plt.hist(biedronka[:, :, 1].ravel(), 256, [0, 1], color='g')
7 plt.xlim(0,1)
8 plt.draw()
9 plt.figure()
10 plt.hist(biedronka[:, :, 2].ravel(), 256, [0, 1], color='b')
11 plt.xlim(0,1)
12 plt.draw()

```

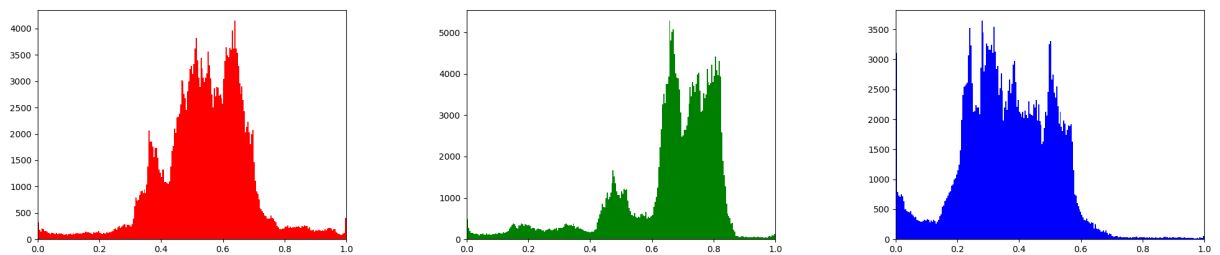


Figure 1: Colour distribution histograms

```

1 plt.figure()
2 plt.hist(biedronka[:, :, 0].ravel(), 256, [0, 1],
3         alpha=0.5, color='r')
4 plt.hist(biedronka[:, :, 1].ravel(), 256, [0, 1],
5         alpha=0.5, color='g')
6 plt.hist(biedronka[:, :, 2].ravel(), 256, [0, 1],
7         alpha=0.5, color='b')
8 plt.xlim(0, 1)
9 plt.show()

```

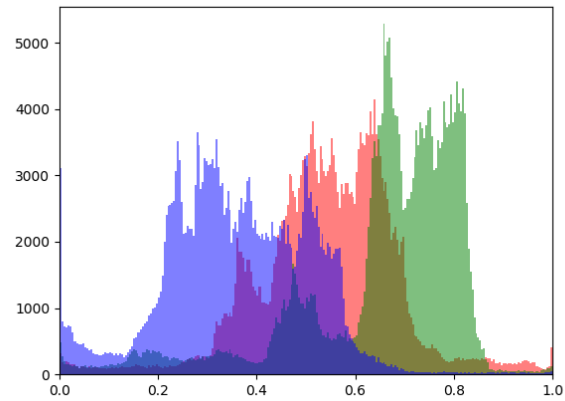


Figure 2: Colour distribution histograms

```

1 plt.figure()
2 biedronka[:biedronka.shape[0] // 2,
3           :biedronka.shape[1] // 2, :] = 0
4 biedronka[biedronka.shape[0] // 2:,
5           biedronka.shape[1] // 2:, :] = 1
6 plt.imshow(biedronka)
7
8 plt.show()

```

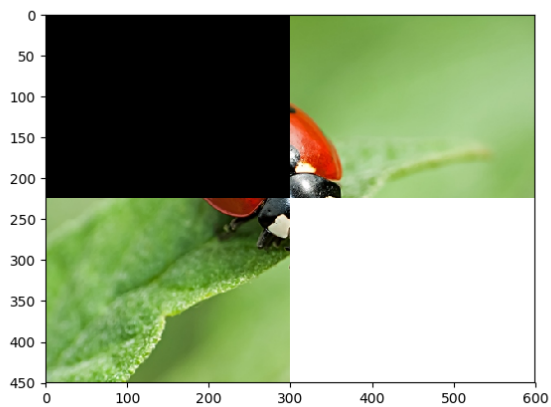


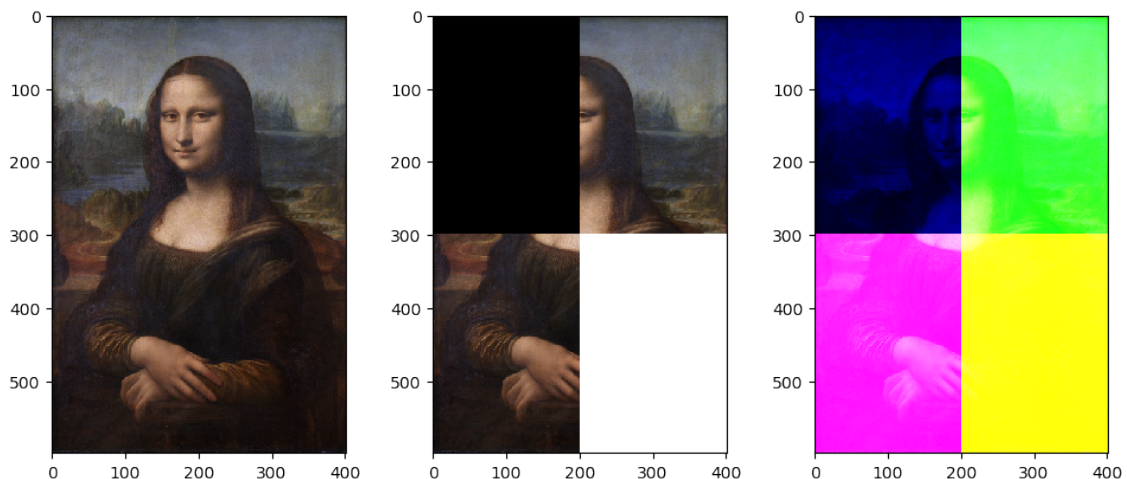
Figure 3: Colour distribution histograms

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.

```

1 monalisa = matplotlib.image.imread('monalisa.png')
2
3 plt.figure()
4 plt.imshow(monalisa)
5 plt.draw()
6
7 ml = monalisa.copy()    # first way of copying
8 plt.figure()
9 ml[:ml.shape[0] // 2, :ml.shape[1] // 2, :-1] = 0
10 ml[ml.shape[0] // 2:, ml.shape[1] // 2:, :-1] = 1
11 plt.imshow(ml)
12 plt.draw()
13
14 ml = monalisa[...]    # another way of array copying
15                        # actually the same as ml = monalisa[:, :, :]
16 plt.figure()
17 ml[:ml.shape[0] // 2, :ml.shape[1] // 2, [0, 1]] = 0
18 ml[ml.shape[0] // 2:, :ml.shape[1] // 2, [0, 2]] = 1
19 ml[:ml.shape[0] // 2, ml.shape[1] // 2:, [1]] = 1
20 ml[ml.shape[0] // 2:, ml.shape[1] // 2:, [0, 1]] = 1
21 plt.imshow(ml)
22 plt.draw()
23
24 plt.show()

```



**Task:**

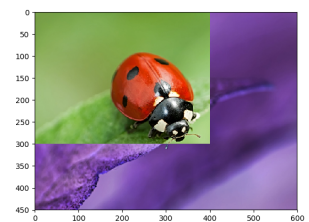
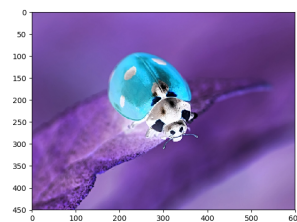
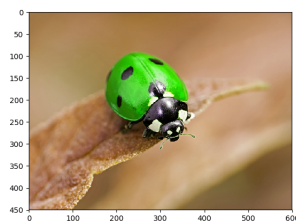
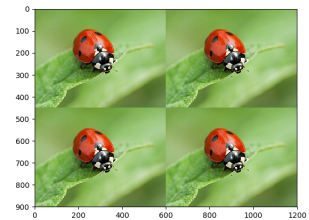
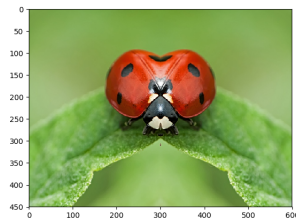
Make and draw a grayscale image from given coloured one.

**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



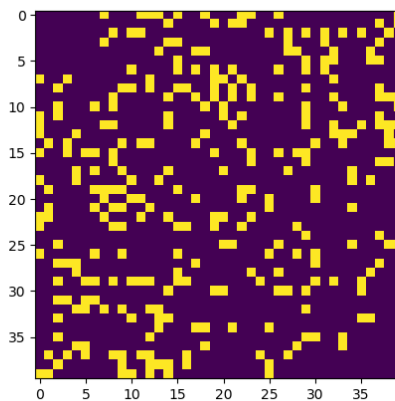
### 9.3 Two-dimensional plots

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.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import networkx as nx
4 import matplotlib.cm
5
6 G = nx.binomial_graph(n=40, p=0.2)
7 plt.figure('Graph')
8 nx.draw_circular(G, node_color=range(len(G)),
9                  cmap=matplotlib.cm.jet, with_labels=True)
10 plt.figure('Graph Matrix')
11 plt.imshow(nx.to_numpy_matrix(G))
12
13 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.