ncLab
Online STEM Laboratory

Introduction to Python Programming

Revision Nov-20-2012

About this Textbook

This free textbook is provided as a courtesy to NCLab users. Python is a modern high-level dynamic programming language that is used in many areas of business, engineering, and science today. After taking this course, you will have solid theoretical knowledge and vast practical experience with computer programming.

Become a Co-Author

We do not plan to publish the textbook with a commercial publisher since this would make it unnecessarily expensive for kids and students who are the main target audience. Feel free to contribute to the textbook with any material or suggestions. There is never enough illustrations and exercises, and there always are bugs to report. Translating the textbook into other languages would benefit thousands of kids worldwide. Instructions for contributors can be found below.

How to Contribute (for LaTeX and Git users)

The textbook is written in LaTeX, a high-quality typesetting system that you can learn and use in NCLab. In the future it will be possible to contribute to the textbook directly in NCLab, but at this time, the sources are stored in a public Git repository `nclab-textbook-python` at Github (http://github.com).

How to Contribute (for all others)

We will gladly accept new interesting exercises as well as new images of good quality that will make the textbook more interesting and fun. You can send those at any time via email to `pavel@femhub.com`.

List of Co-Authors

– Pavel Solin, University of Nevada, Reno (primary author), USA.
– Martin Novak, Czech Technical University, Prague, Czech Republic.
– Salih Dede, Coral Academy of Science High School, Reno, USA.
– Nazhmiddin Shapoatov, Sonoran Science Academy, Phoenix, USA.
– Artur Skonecki, Warsaw University of Technology, Warsaw, Poland.
– Joel Landsteiner, Cray Inc, USA.
– William Mitchell, NIST, USA.

– Venkata Rama Rao Mallela, Hyderabad, India.
– Steven Lamb, Philadelphia, USA.
– Norman Dunbar, Leeds, West Yorkshire, England.

Graphics Design: TR-Design `http://tr-design.cz`

Table of Contents

I Textbook

1 Introduction . . . 5
1.1 Objectives . . . 5
1.2 Compiled and interpreted programming languages . . . 5
1.3 Basic facts about Python . . . 5
1.4 Python programming in NCLab . . . 6
1.5 Launching a Python project . . . 6
1.6 Hello, World! . . . 6
1.7 Code, output, and descriptive cells . . . 7
2 Using Python as a Calculator . . . 8
2.1 Objectives . . . 8
2.2 Addition and subtraction . . . 9
2.3 Multiplication . . . 9
2.4 Division . . . 10
2.5 Modulo . . . 12
2.6 Powers . . . 12
2.7 Priority of operators . . . 13
2.8 Using empty characters makes your code more readable . . . 14
2.9 Using mathematical functions . . . 14
2.10 Fractions . . . 15
2.11 Random numbers . . . 16
2.12 Complex numbers . . . 17
3 Functions . . . 18
3.1 Objectives . . . 18
3.2 Defining new functions . . . 18
3.3 Passing arbitrary arguments . . . 20
3.4 Returning multiple values . . . 20
3.5 Using default arguments . . . 21
4 Colors and Plotting . . . 24
4.1 Objectives . . . 24
4.2 RGB colors . . . 25
4.3 Plotting polylines and polygons . . . 26
4.4 Plotting functions of one variable . . . 28
4.5 Labels, colors, and styles . . . 29
4.6 Scaling axes and showing grid . . . 31

4.7 Adjusting plot limits 32
4.8 Plotting multiple functions at once 33
4.9 Plotting parametric 2D curves 35
4.10 Plotting parametric 3D curves 36
4.11 Plotting functions of two variables 37
4.12 Plotting functions of two variables with WebGL 40
4.13 Plotting pie charts 42
4.14 Plotting bar charts 44
5 Variables 47
5.1 Objectives 47
5.2 Creating variables 48
5.3 Initializing multiple variables at once 49
5.4 Changing values of variables 50
5.5 Local and global variables 51
5.6 How to get in trouble 52
5.7 Variable shadowing 53
5.8 Overriding constants 54
6 Logic and Probability 54
6.1 Objectives 54
6.2 `True` and `False` 54
6.3 Boolean variables 56
6.4 Boolean operations 56
6.5 Monte Carlo methods 58
7 Conditional Loop 61
7.1 Objectives 61
7.2 Counting and conditional loops 62
7.3 Conditions and the `elif` statement 62
7.4 The `while` loop 64
7.5 The `break` statement 65
7.6 The `continue` statement 66
7.7 Abusing the `while` loop 67
7.8 Next trip into scientific computing 68
8 Strings 70
8.1 Objectives 70
8.2 Using quotes 70
8.3 Multiline strings and backslashes 70
8.4 Concatenation and repetition 72
8.5 Referring to letters by their indices 73
8.6 Parsing strings with the `for` loop 73

8.7 Slicing strings 75
9 Tuples, Lists, and Dictionaries 76
9.1 Objectives 76
9.2 Tuples 76
9.3 Lists 77
9.4 Dictionaries 81
10 More on Counting Loop 83
10.1 Objectives 83
10.2 The `range()` function revisited 83
10.3 The `for` loop combined with the `range()` function 84
10.4 The `for` loop combined with a list or tuple 84
10.5 The `for` loop combined with a dictionary 85
11 Exceptions 86
11.1 Objectives 86
11.2 Exceptions handling 87
11.3 The `assert()` function 87
11.4 The `raise` statement 88
11.5 Other types of exceptions 89
12 Object-Oriented Programming 90
12.1 Objectives 90
12.2 From procedural to object-oriented programming 91
12.3 Example: Moving to a different city 91
12.4 Classes, objects, and methods 92
12.5 An example from geometry 92
12.6 Defining and using class `circle` 93
12.7 Text string as a class 95
13 Class Inheritance 101
13.1 Objectives 101
13.2 Creating base class `geometry` 101
13.3 Deriving `polygon` from `geometry` 102
13.4 Deriving `circle` from `geometry` 103
13.5 Deriving `triangle` and `quad` from `polygon` 104
13.6 Deriving `rectangle` from `quad` 105
13.7 Deriving `square` from `rectangle` 105
13.8 Diagram of class structure 106
13.9 Creating sample instances 106
14 Python Libraries 109
14.1 Objectives 109
14.2 Python Standard Library 109

14.3 Pylab and Matplotlib 109
14.4 Scipy, Numpy and Sympy 110
15 Other Recommended Topics in Python 110
16 What Next? 110

II Programming Exercises

1 Introduction 113
1.1 Python is fun 113
1.2 Big word 113
1.3 Military exercise 113
2 Using Python as a Calculator 113
2.1 Grocery shopping 113
2.2 Oil change 114
2.3 Age average 114
2.4 Saving for a bike 114
2.5 Growing town 114
2.6 Coffee maker 114
2.7 Gardening 114
2.8 Setting tiles 114
2.9 Submerged soccer ball 115
2.10 Fastest runner 115
2.11 Triangle area 115
2.12 Math functions 115
2.13 Random numbers 115
2.14 Greatest common divisor 116
2.15 Fractions 116
3 Functions 116
3.1 Circle 116
3.2 Square 116
3.3 Rectangle 116
3.4 Sales prediction 116
3.5 Sales tax 117
3.6 Restaurant bill 117
3.7 Gas consumption conversion I (EU to US) 117
3.8 Gas consumption conversion II (US to EU) 117
3.9 Distance per tank of gas 117
3.10 Circuit board price 117
3.11 Gross pay 118

3.12 Stock gain 118
3.13 Temperature conversion I (Fahrenheit to Celsius) 118
3.14 Temperature conversion II (Celsius to Fahrenheit) 118
3.15 Time in seconds 118
3.16 Time in hours 119
3.17 Time last time 119
3.18 Distance 119
3.19 Angle conversion 119
3.20 Length conversion 119
3.21 Area conversion 119
3.22 Digital storage conversion 119
3.23 Savings account 120
4 Colors and plotting 120
4.1 Polygon 120
4.2 Custom plotting function 120
4.3 Planar curve 120
4.4 Random triangle 121
4.5 Wireframe plot 121
4.6 Solid surface plot 121
4.7 Contour plot 122
4.8 WebGL plot 122
4.9 Pie chart plot 122
5 Variables 122
5.1 Arithmetic sequence 122
5.2 Geometric sequence 123
5.3 Graph length 123
5.4 3D curve length 124
5.5 Triangle area 124
5.6 Roof area 124
6 Logic and probability 125
6.1 Even or odd 125
6.2 Mysterious parabola 125
6.3 Point in circle 125
6.4 Slot machine 125
6.5 Dice game 126
6.6 Cheat dice 126
6.7 Monte Carlo pie 126
6.8 Maximum of a function 127
6.9 Monte Carlo area 127

6.10 Trapezoids 128
7 Conditional Loop 129
7.1 Water jar 129
7.2 Throwing stones 129
7.3 Missing numbers 129
7.4 Trapezoids revisited 129
7.5 Infinite sum 130
8 Strings 130
8.1 Spelling 130
8.2 Inserting spaces 130
8.3 Counting chars 130
8.4 Counting words 130
8.5 Search and replace 131
9 Tuples, Lists, and Dictionaries 131
9.1 List reverse 131
9.2 String to list 131
10 More on Counting Loop 131
10.1 Analyze string 131
10.2 Multiplication table 131
10.3 Approximating π 132
10.4 Prime numbers 132
10.5 List maximizer 132
10.6 Fibonacci 132
10.7 Adding fractions 132
11 Exceptions 132
11.1 Sure plot 133
11.2 Syntax check 133
12 Object-Oriented Programming 133
12.1 Bank account 133
12.2 Word magic 134
12.3 Number magic 134
12.4 Vector magic 135
13 Class Inheritance 135
13.1 Number master 135
13.2 Vector master 136

III Review Questions

1 Introduction 139

2 Using Python as a Calculator 142
3 Functions 146
4 Colors and plotting 148
5 Variables 154
6 Logic and Probability 157
7 Conditional Loop 160
8 Strings 163
9 Tuples, Lists, and Dictionaries 165
10 More on Counting Loop 171
11 Exceptions 172
12 Object-Oriented Programming 173
13 Class Inheritance 175

Foreword

This course provides a gentle yet efficient introduction to Python – a modern high-level dynamic programming language that is widely used in business, science, and engineering applications. If this is your first time learning a programming language, then we recommend that you spend a few days with Karel the Robot before diving into Python. Karel the Robot is available in NCLab (http://nclab.com) and it will teach you what you will need most – *algorithmic thinking*. Once you learn that, your efficiency in acquiring any new programming language will improve dramatically. Moreover, Karel's syntax is very similar to Python, so the transition from Karel to Python is truly seamless.

In Python you will learn more applied concepts including mathematical operations and functions, 2D and 3D plotting, local and global variables, strings, tuples, lists, dictionaries, exceptions, object-oriented programming, and more. A strong companion of Python are its libraries. Besides the Standard Library that contains many built-in functions not present in lower-level languages such as Java, C, C++ or Fortran, Python also has powerful scientific libraries including Scipy, Numpy, Matplotlib, Pylab, Sympy and others. With these, you will be able to solve entry-level scientific and engineering problems. These libraries are used throughout the course.

Part I

Textbook

1 Introduction

1.1 Objectives

– Learn the difference between compiled and interpreted programming languages.
– Learn basic facts about Python and its powerful companion libraries.
– Understand how Python programming works in the cloud setting.
– Write and run your first Python program.

1.2 Compiled and interpreted programming languages

Compilation is a process where human-readable *source code (text)* is translated by means of a *compiler* and a *linker* into a *binary (executable) file* which then can be run on the concrete computer. The same source code, compiled on different hardware architectures, yields different binaries.

Interpreted (scripting) programming languages are more recent than the compiled ones. A program that is written using an interpreted language is "read" (parsed) at runtime – it is not compiled and there are no binaries. Programs written using compiled languages are usually more efficient than programs written using the interpreted ones because the former take better advantage of the underlying hardware. On the other hand, interpreted languages are usually more universal and easier to use. Compiled programming languages include Pascal, C, C++, Fortran, Java and many others. Examples of interpreted languages are Python, Lua, Perl and Ruby.

1.3 Basic facts about Python

Python is a powerful modern programming language that is used in many areas of business, engineering, and science. Its interpreted character along with a very intuitive syntax make Python an ideal language for beginners.

Python was conceived in the late 1980s and its implementation was started in December 1989 by Guido van Rossum in the Netherlands. Rossum also gave it a name originated in his favorite television series *Monty Python's Flying Circus*. Python 2.0 was released in October 2000 and Python 3.0 in December 2008. Python was awarded the TIOBE *Programming Language of the Year* award twice (2007, 2010), which is given to the language with the greatest growth in popularity over the course of a year.

Python is a multi-paradigm programming language. Rather than forcing programmers to adopt a particular style of programming, it permits several styles: *structured (procedural) programming* and *object-oriented programming* are fully supported, and there are a number of language features which support *functional programming*.

In this course you will learn the most important aspects of the language and you will be able to use the programming language to solve a large variety of problems.

Depending on your objectives, this course may be all you will ever need. We hope that you will like Python and want to learn more – and there is much more to learn out there. References to materials covering more advanced topics and object-oriented programming are given in Section 15.

1.4 Python programming in NCLab

In NCLab's Python Worksheet, Python code is typed into one or more *code cells*. The code is then sent as a text string to a remote server where it is interpreted, and the results are sent back to your web browser. It does not matter if you are using a desktop computer, laptop, netbook, tablet, or any other hardware platform. Your hardware is not used for computing, only in a limited way for graphics rendering.

1.5 Launching a Python project

There are multiple ways to launch a Python project. First, it is possible to clone and experiment with many existing displayed Python projects via File Manager → Project → Clone.

New Python worksheet can be launched via the Programming module on Desktop or via File Manager → Project → New → Python. Python worksheet launched in this way will contain a *welcome script* – simple Python program that can be run instantly by pressing the blue arrow button. The welcome script is meant to get you started quickly, and it can be turned off in Setting. Once turned off, next time you launch the Python worksheet, it will be empty. An empty Python worksheet is shown in Fig. 1.

1.6 Hello, World!

Click into the code cell and type `print "Hello, World!"`. Then click on the green arrow under the cell, and the text "Hello, World!" will be displayed in a new yellow *output cell* as shown in Fig. 2.

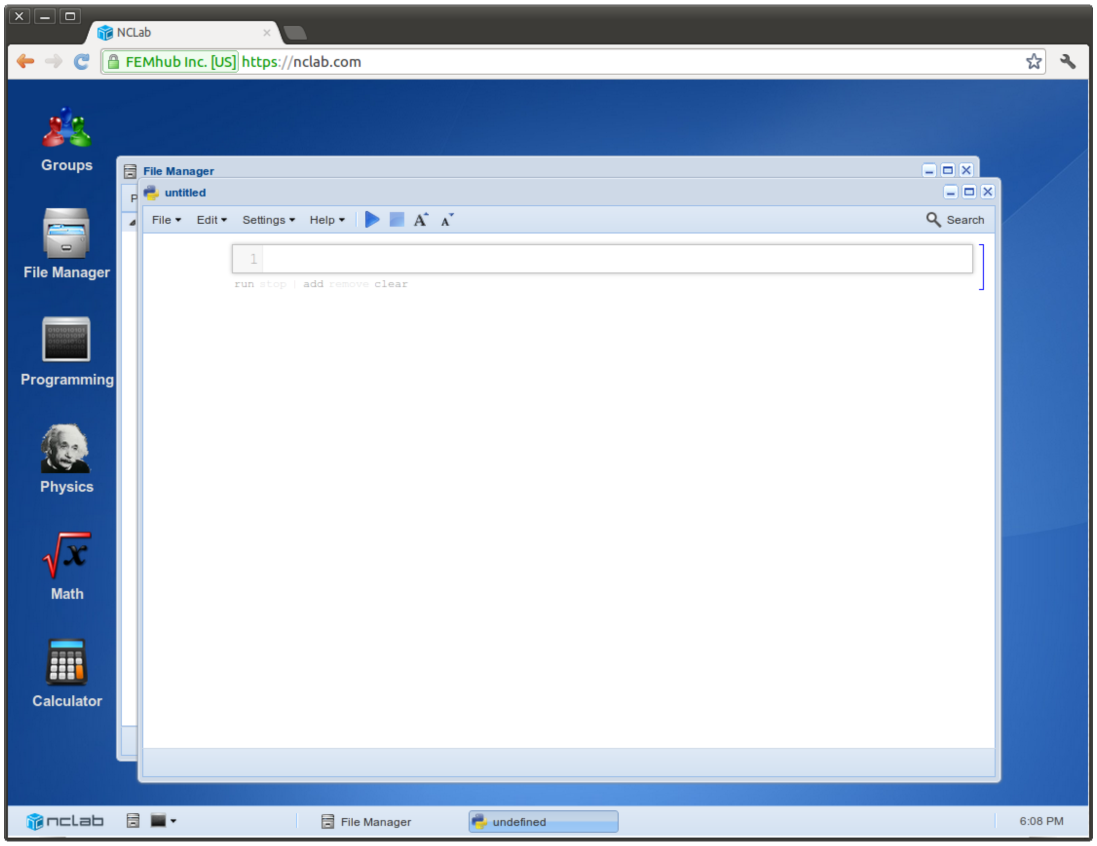


Fig. 1: Launching a new Python project.

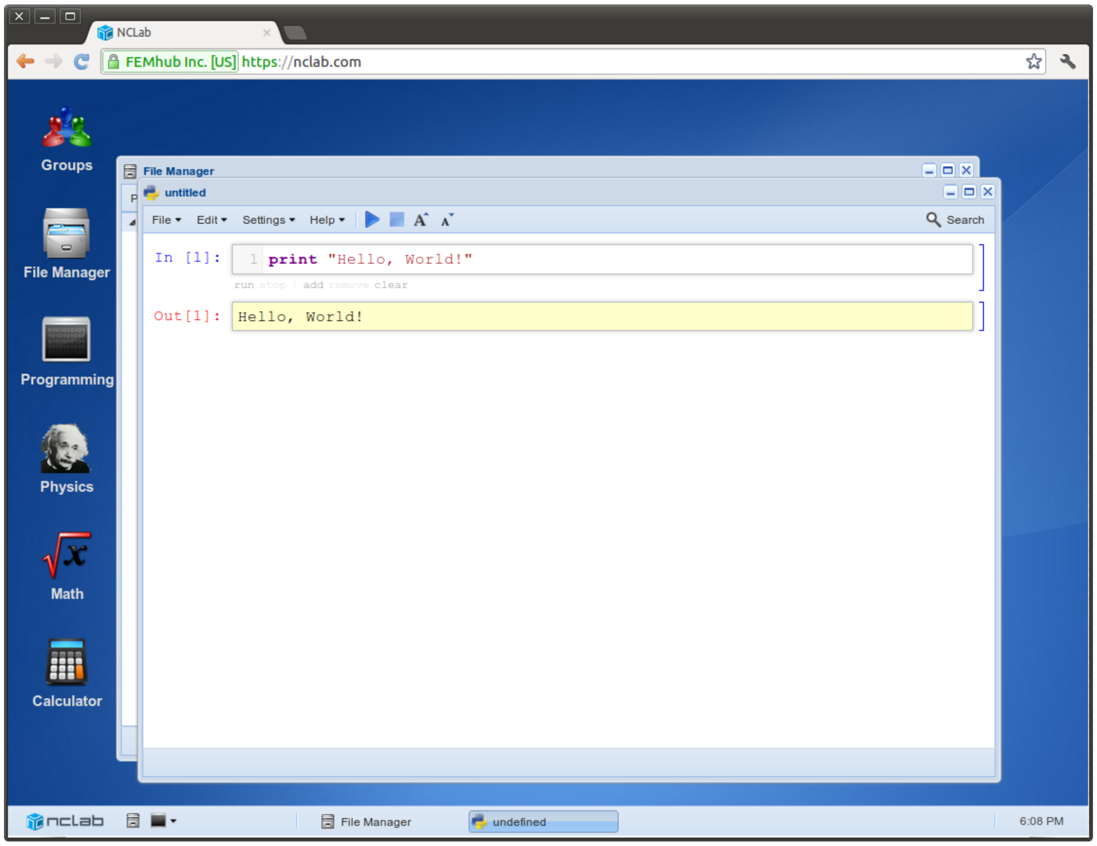
The blue and red labels on the left-hand side of cells can be turned off in Settings, allowing a bit more space for editing.

1.7 Code, output, and descriptive cells

There are three types of cells in the Python worksheet:

- *code cells* for entering computer code,
- *output cells* to display results,
- *descriptive cells* for textual and graphical descriptions.

Code cells as well as descriptive cells can be added via buttons located under any code or descriptive cell. Each cell can be removed by clicking on the red remove button below the cell. All output cells can be removed at once via *Remove all output* option in the Edit menu. Pressing the green arrow button in the upper menu will run all code



```
In [1]:  print "Hello, World!"
Out[1]:  Hello, World!
```

Fig. 2: Response received from the cloud is shown in a yellow output cell.

cells in the project. Clicking on the green arrow under a code cell will run only the contents of that particular code cell.

2 Using Python as a Calculator

2.1 Objectives

- Learn how to use Python for elementary as well as advanced math operations.
- Learn how to work with mathematical functions, fractions, random numbers and complex numbers.

Python can be used as an advanced scientific calculator – no need to own a TI-89 ($150 value). As a matter of fact, NCLab is much more powerful – a TI-89 hardly could compete with thousands of processors under NCLab's hood. In this textbook we will learn how to use this computing power. Let us begin with the simplest math operations. The rest of this section is fairly slow, so feel free to skip to Section 3 should it be too boring – you can always return to it later.

2.2 Addition and subtraction

Launch a new Python project and in the code cell type:

```
3 + 6
```

Then click on the green arrow under the cell. The output should be displayed quickly:

```
9
```

Of course you can add real numbers too:

```
3.2 + 6.31
```

Output:

```
9.51
```

Two numbers can be subtracted using the minus sign:

```
7.5 - 2.1
```

Output:

```
5.4
```

2.3 Multiplication

Multiplication is done using the asterisk '*' symbol as in

```
3 * 12
```

Output:

```
36
```

Indeed, real numbers can be multiplied as well:

```
3.7 * 12.17
```

Output:

```
45.029
```

2.4 Division

The forward slash '/' symbol is used for division:

```
30 / 5
```

Output:

```
6
```

However, we need to be careful. Look at this:

```
33 / 5
```

Output:

```
6
```

This doesn't look correct! Why did Python return `6` and not `6.6`? The answer is that since the numbers `33` and `5` are both integers, the '/' operator performs *integer division*, or *floored division*, and returns an integer. This is because like C, C++, and Fortran, Python is *strongly-typed*. This means that python will avoid implicitly changing between types of data. In order to perform floating-point division, the type of division that is more common in the sciences, at least one of the numbers must be represented as a real number.

Adding a decimal point after the number will accomplish this. Now, the result is a real number:

```
33. / 5
```

yields

```
6.6
```

An alternative way of turning an integer into a real number, which also works for variables, is using the function `float()`. Then the above division could be done as follows:

```
float(33) / 5
```

Output:

```
6.6
```

Once we understand the behavior, we can use it to our advantage. For example, we can calculate how many times a dozen fits into a thousand:

```
1000 / 12
```

Output:

```
83
```

By calculating `1000 - 83 * 12` we obtain `4` which is the remainder of the integer division. In fact the remainder can be calculated in a simpler way using the modulo operator that will be introduced in the next paragraph.

In summary:

Keep in mind that division is a tricky operation. Failure to convert at least one operand to a real number can be a source of mistakes that are very hard to find.

2.5 Modulo

The last of the common arithmetic operations is *modulo* (remainder after integer division). In Python modulo is represented via the percent symbol '%':

```
6 % 4
```

Output:

```
2
```

Modulo can be applied to real numbers as well:

```
12.5 % 2.0
```

Output:

```
0.5
```

2.6 Powers

For exponents, such as in 2^4, Python has a double-star symbol '**':

```
2**4
```

Output:

```
16
```

Both the base and the exponent can be real numbers:

```
3.2**2.5
```

Output:

```
18.31786887167828
```

But we have to be careful with negative numbers:

```
(-3.2)**2.5
```

Output:

```
Traceback (most recent call last):
  File "<nclab>", line 1, in <module>
ValueError: negative number cannot be raised to a fractional
power
```

2.7 Priority of operators

Python respects the priority of operators that is standard in mathematics:

- Round brackets '`(...)`'have the highest priority,
- then exponentiation '`**`',
- then multiplication '`*`', division '`/`' and modulo '`%`',
- the lowest priority have addition '`+`' and subtraction '`-`',
- operations with the same priority are evaluated from left to right – for example the result of `20 / 10 * 2` is `4`.

Note that no other brackets such as `{ }` and `[ ]` are admissible in mathematical expressions. The reason is that they have a different function in the programming language. To illustrate the priority of operations, we evaluate the following expression:

```
3**4 / 27 * 5 + 3 * 5
```

Output:

```
30
```

If we are not sure, it never hurts to use round brackets:

```
(3**4) / 27 * 5 + 3 * 5
```

Output:

```
30
```

2.8 Using empty characters makes your code more readable

Your code will be much more readable if you use empty characters on either side of arithmetic symbols, as well as after commas. Hence, you should never write things like

```
sin(x+y)+f(x,y,z)*5-2.4.
```

Instead, the same can be written in a much more reader-friendly form as

```
sin(x + y) + f(x, y, z) * 5 - 2.4.
```

2.9 Using mathematical functions

In order to calculate square roots, exponentials, sines, cosines, tangents, and many other math functions, the best way is to import Numpy. Numpy is a powerful Python library for numerical computations. To import it, just include the following line in your code:

```
from numpy import *
```

Here the symbol '`*`' stands for "everything". If you wanted to import just one or two functions, you could do that as well by just giving their names, separated by commas. After Numpy is imported, we can calculate, for example, e^2:

```
exp(2)
```

Output:

```
7.3890560989306504
```

Elementary functions (and constants) that one can import from Numpy are listed below. We also show their arguments for clarity, but the functions are imported without them. For example, the absolute value function is imported via `from numpy import abs`.

pi	π
abs(x)	absolute value of x
arccos(x)	inverse cosine of x
arccosh(x)	inverse hyperbolic cosine of x
arcsin(x)	inverse sine of x
arcsinh(x)	inverse hyperbolic sine of x
arctan(x)	inverse tangent of x
arctanh(x)	inverse hyperbolic tangent of x
arctan2(x_1, x_2)	arc tangent of x_1/x_2 choosing the quadrant correctly
cos(x)	cosine of x
cosh(x)	hyperbolic tangent of x
exp(x)	e^x
log(x)	natural logarithm of x
pow(a, b)	a^b (same as "a**b")
sin(x)	sine of x
sinh(x)	hyperbolic sine of x
sqrt(x)	square root of x
tan(x)	tangent of x
tanh(x)	hyperbolic tangent of x

In summary:

Python provides many readily available mathematical functions via the Numpy library. To use them, import them via the command `from numpy import *`

2.10 Fractions

Python makes operation with fractions easy via the `Fraction` function that is imported from the `fractions` library:

```
from fractions import Fraction
```

A fraction such as 4/7 can be defined simply as `Fraction(4, 7)`. Fractions can be used with the same operations as numbers, and the result of such an operation is a `Fraction`. For example

```
Fraction(2, 6) + Fraction(2, 3)
```

yields

```
Fraction(1, 1)
```

Another example:

```
Fraction(2, 6) / Fraction(2, 3)
```

results into

```
Fraction(1, 2)
```

The `fractions` library also provides the useful function `gcd()` that calculates the Greatest Common Divisor (GCD) of two integers. It is imported via

```
from fractions import gcd
```

For illustration let us calculate the GCD of 867 and 629:

```
gcd(867, 629)
```

The output is

```
17
```

2.11 Random numbers

Python provides a random number generator via the `random()` function that can be imported from the `random` library:

```
from random import random
```

This function returns a random real number between 0 and 1. For example,

```
random()
```

yields

```
0.871979925682207
```

Sometimes we need to generate random integers rather than real numbers. This is easy. For illustration, a random integer `n` between 1 and 3 can be generated via the code

```
a = random()
n = int(3*a + 1)
```

Here the function `int()` will erase the decimal part of the real number, converting it to an integer.

2.12 Complex numbers

Complex numbers are always represented as two floating point numbers, the real and imaginary part. Appending `'j'` or `'J'` to a real number makes it imaginary:

```
1j * 1J
```

Output:

```
(-1+0j)
```

This is one way to define complex numbers:

```
1 + 3j
```

Output:

```
(1+3j)
```

Another way is to use the command `complex`:

```
complex(1, 3)
```

Output:

```
(1+3j)
```

All arithmetic operations that are used for real numbers can be used for complex numbers as well, for example:

```
(1 + 2j) / (1 + 1j)
```

Output:

```
(1.5+0.5j)
```

To extract the real and imaginary parts of a complex number `z`, use `z.real` and `z.imag`. Use `abs()` to get the absolute value:

```
a = 3 + 4j
a.real
a.imag
abs(a)
```

Output:

```
3
4
5
```

3 Functions

3.1 Objectives

- Review what we know about functions from Karel the Robot.
- Learn that in Python, functions can accept input arguments.
- Learn to use default arguments and return multiple values.

3.2 Defining new functions

The purpose of creating custom functions is to make selected functionality easily reusable. In this way we can avoid code duplication, and bring more structure and transparency into our programs.

In Karel the Robot, custom functions were similar to commands but additionally, they could return values. For example, a function `turnsouth`, that turns the robot to face South and makes him return his GPS coordinates, looks as follows:

```
def turnsouth
    while not north
        left
    repeat 2
        left
    return [gpsx, gpsy]
```

In Python, the definition of a new function also begins with the keyword `def`, but we moreover have to use round brackets for input arguments and the colon '`:`' at the end of the line. For example, the following function adds two numbers and returns the result:

```
def add(a, b):
    return a + b
```

The round brackets in the function definition are mandatory even if no arguments are passed but the `return` statement can be omitted if not needed. The two lines of code above are a *function declaration* only – the function is not called. In order to call the function, we need to write one additional line:

```
print "5 + 3 is", add(5, 3)
```

This will produce the following output:

```
5 + 3 is 8
```

In most cases, functions are defined to process some input arguments and to return some output values. However, this does not apply always. The following function does not take any arguments and it does not return anything:

```
def print_hello():
    print "Hello!"
```

3.3 Passing arbitrary arguments

Python does not require that we specify the type of function arguments. What does it mean? The above function `add(a, b)` works for real numbers, complex numbers, vectors, strings, and any other objects where the operation '+' is defined. Let's try this:

```
def add(a, b):
    return a + b

word1 = "Good "
word2 = "Evening!"
print add(word1, word2)
```

Output:

```
Good Evening!
```

This makes Python very intuitive and easy to use.

3.4 Returning multiple values

Python functions can return multiple values which often comes handy. For example, the following function returns the second, third, and fourth powers of a number:

```
def powers(a):
    return a**2, a**3, a**4
```

This is how the function is used:

```
var1 = 2
print "Powers are", powers(var1)
```

Output:

```
Powers are (4, 8, 16)
```

We can also store the returned values in three separate variables:

```
var1 = 3
p2, p3, p4 = powers(var1)
print "Powers are", p2, p3, p4
```

Output:

```
Powers are 9 27 81
```

Again, for comparison let's see how this would be handled in C/C++. The asterisks in the code below are *pointers*, an additional programming concept that one needs to learn and utilize here:

```
void powers(double a, double* p2, double* p2, double* p3)
{
  *p2 = pow(a, 2);
  *p3 = pow(a, 3);
  *p4 = pow(a, 4);
  return;
}

int main()
{
  double p1, p2, p3;
  double a = 2;
  powers(a, &p2, &p2, &p3);
  printf("Powers are %g, %g, %g.\n", p2, p3, p4);
}
```

3.5 Using default arguments

Have you ever been to Holland? It is the most bicycle friendly place in the world. Imagine that you work for the Holland Census Bureau. Your job is to ask 10000 people how they go to work, and enter their answers into a database. The program for entering data into the database was written by one of your colleagues, and it can be used as follows:

```
add_database_entry("John", "Smith", "walks")
```

or

```
add_database_entry("Louis", "Armstrong", "bicycle")
```

or

```
add_database_entry("Jim", "Bridger", "horse")
```

etc. Since you are in Holland, it can be expected that 99% of people are using the bicycle. In principle you could call the function `add_database_entry()` to enter each answer, but with 9900 bicyclists out of 10000 respondents you would have to type the word `"bicycle"` MANY times.

Fortunately, Python offers a smarter way to do this. We can define a new function

```
def enter(first, last, transport="bicycle"):
    enter_into_database(first, last, transport)
```

This is a simple ("thin") *wrapper* to the function `add_database_entry()` that allows us to omit the third argument in the function call and autocomplete it with a *default argument* `"bicycle"`. In other words, now we do not have to type `"bicycle"` for Louis Armstrong or any other bicyclist:

```
enter("Louis", "Armstrong")
```

Only if we meet a rare someone who uses a car, we type

```
enter("Niki", "Lauda", "car")
```

Another example: Let us return to the function `add()`, and let us imagine that very often (but not always) the second number that we are adding is 10. Then it makes sense to write the function `add()` as follows:

```
def add(a, b=10):
    return a + b
```

The function will work as before when called with two arguments:

```
A = 5
B = 1
add(A, B)
```

Output:

```
6
```

But it can also be called with the second argument omitted:

```
A = 5
add(A)
```

Output:

```
15
```

Few rules to remember

(1) Default arguments need to be introduced **after** standard (non-default) arguments. In other words, the following code will result into an error:

```
def add(a=5, b):
    return a + b
```

This is the error message:

```
File "<nclab>", line 1
SyntaxError: non-default argument follows default argument.
```

(2) If multiple default arguments are present, they have to follow the non-default ones. If the number of arguments in the function call is less than the total number of default and non-default arguments, then first all non-default arguments are assigned, and then the default ones from left to right. To illustrate this, assume the function:

```
def add(x, a=2, b=3):
    return x + a + b
```

This function can be called as follows:

```
print add(1)
```

In this case, the value `1` is assigned to `x` and `a` and `b` take the default values. The output is

```
6
```

When the function is called with just two values,

```
print add(5, 6)
```

then the value `5` is assigned to `x`, `6` to `a`, and `b` takes the default value `3`. The output is

```
14
```

However, it is always a good idea to be as transparent as possible. Therefore a better code is

```
print add(1, a = 6)
```

The result is the same, `14`, but it is clear even to non-experts what is going on. Moreover, in this way we can also define the value of `b` without having to define the value of `a`:

```
print add(1, b = 6)
```

Now `x` is `1`, `a` has the default value `2` and the result is `9`.

4 Colors and Plotting

4.1 Objectives

- Understand how colors are composed of Red, Green and Blue components.
- Learn how to plot polygons, graphs of functions of one and two variables.
- Learn how to plot 2D and 3D curves and surfaces.
- Learn how to plot pie charts and bar charts.

4.2 RGB colors

According to the *additive color model* which is based on the human perception of colors, every color can be obtained by mixing the shades of Red, Green and Blue. These are called *additive primary colors* or just *primary colors*. The resulting color depends on the proportions of the primary colors in the mix. It is customary to define these proportions by a number between 0 and 1 for each primary color. So, the resulting color is a triplet of real numbers between 0 and 1. For the primary colors we have

Color	R	G	B
Red	1	0	0
Green	0	1	0
Blue	0	0	1

These colors are shown in the following Fig. 3.

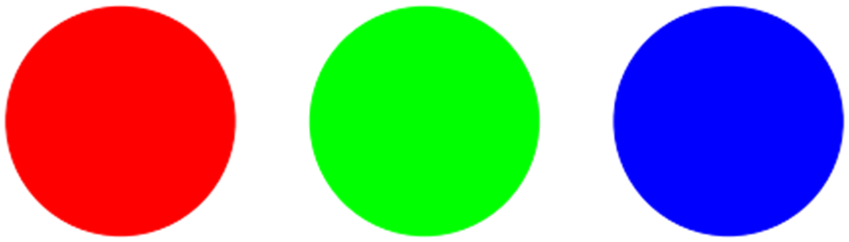


Fig. 3: Left: pure red color [1, 0, 0]. Middle: pure green color [0, 1, 0]. Right; pure blue color [0, 0, 1].

When the proportions of all three primary colors are the same, the result is a shade of grey. With [0, 0, 0] one obtains black, with [1, 1, 1] white. This is illustrated in Fig. 4.

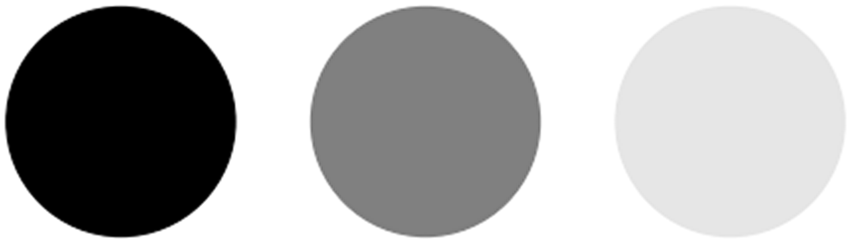


Fig. 4: Left: black color [0, 0, 0]. Middle: dark grey color [0.5, 0.5, 0.5]. Right; light grey color [0.9, 0.9, 0.9].

Guessing RGB codes is not easy. If you need to find the numbers representing your favorite color, the best way is to Google for "rgb color palette". You will find many pages that translate colors into RGB codes. Most of them will have the colors represented by integer numbers between 0 and 255 (as opposed to the 0 to 1 scale). This is because these integers are easier to represent using bits and Bytes. However, it is sufficient to divide all three numbers by 255 to translate the color into the original unit scale. Fig. 5 shows three "easy" colors cyan, pink and yellow along with their RGB codes.

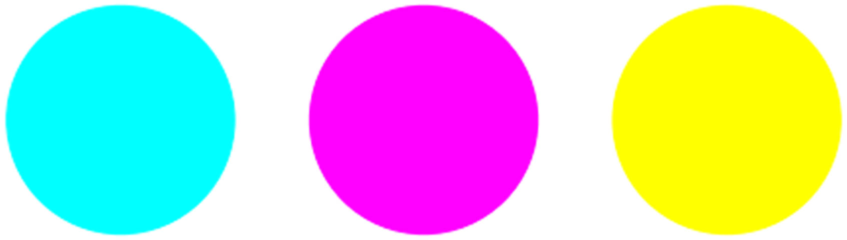


Fig. 5: Left: cyan color [0, 1, 1]. Middle: pink color [1, 0, 1]. Right; yellow color [1, 1, 0].

4.3 Plotting polylines and polygons

In Python, plotting is done via the Pylab library. The Pylab `plot()` function takes two arrays: x-coordinates and y-coordinates of points in the xy plane. Between the points, the curve is interpolated linearly. Let us illustrate this on a simple example with just five points [0, 0], [1, 2], [2, 0.5], [3, 2.5] and [4, 0]:

```python
from pylab import *
x = [0.0, 1.0, 2.0, 3.0, 4.0]
y = [0.0, 2.0, 0.5, 2.5, 0.0]
clf()
plot(x, y)
show()
```

The commands `clf()`,`plot()` and `show()` do clear the canvas, plot the graph, and fetch the image from server, respectively. The output is shown in Fig. 6.

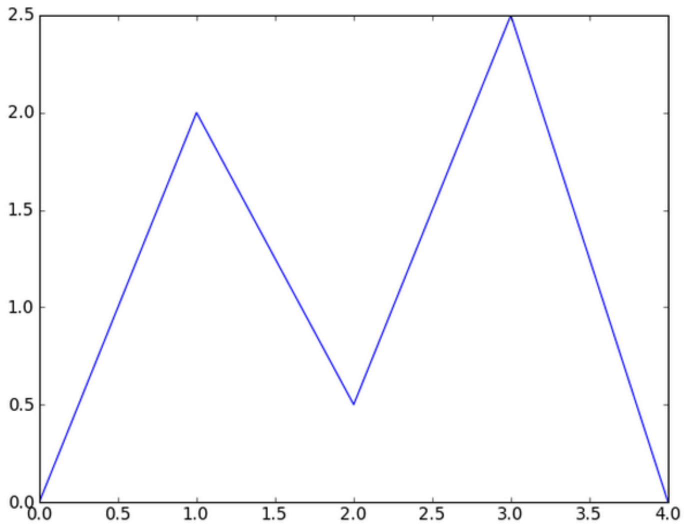


Fig. 6: Polyline with five points.

If the first and last points were the same, then the polyline would form a closed loop and one would obtain a polygon. The above plot was done using a default blue color. The color can be set by passing an optional parameter `color = [R, G, B]` into the `plot()` function. For illustration, changing the fifth line in the last script to `plot(x, y, color = [1, 0, 1])`, we obtain a pink polyline shown in Fig. 7.

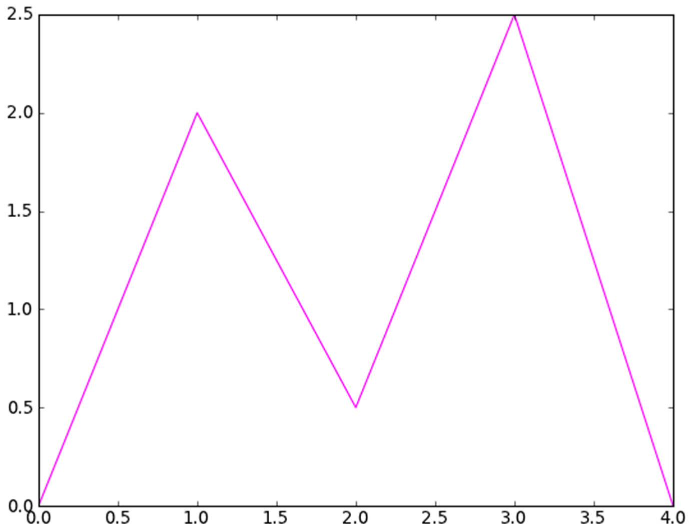


Fig. 7: Setting custom color.

4.4 Plotting functions of one variable

In the following we will discuss more options and show some useful techniques. Let's say, for example, that we want to plot the function $f(x) = \sin(x)$ in the interval $(0, 2\pi)$. The array of x-coordinates of equidistant points between 0 and π can be created easily using the function `linspace()`:

```
from numpy import *
x = linspace(0, 2*pi, 100)
```

Here 100 means the number of points in the division (including endpoints). Increasing this number will improve the resolution and vice versa. Next, the array of y-coordinates of the points is obtained via

```
y = sin(x)
```

The last part we already know:

```
clf()
plot(x, y)
show()
```

The output is shown in Fig. 8.

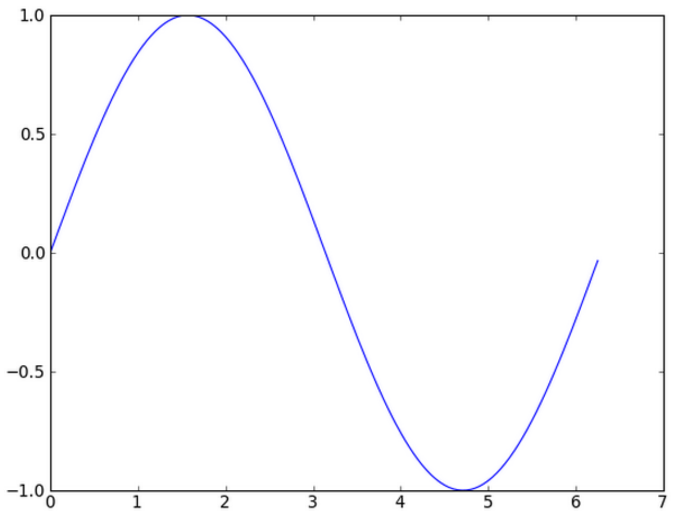


Fig. 8: Plotting $\sin(x)$ in interval $(0, 2\pi)$ with subdivision step 0.05.

4.5 Labels, colors, and styles

The plot can be made nicer by adding a label, and also the color and the line style can be changed. Let us start with adding a label:

```
plot(x, y, 'b-', label = "Solid blue line")
legend()
show()
```

The output is shown in Fig. 9.

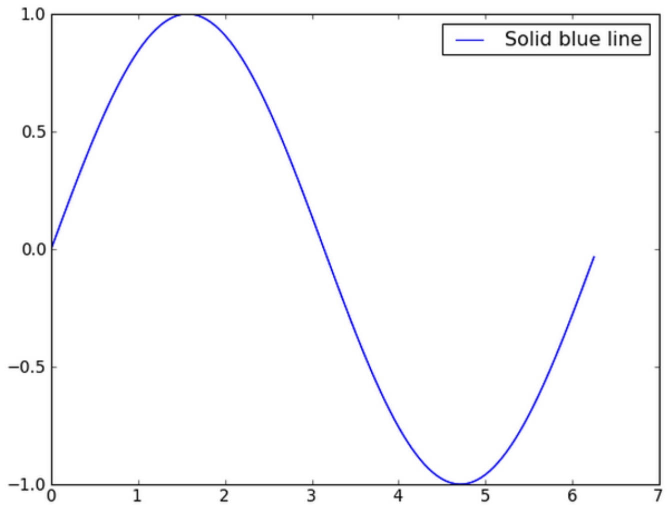


Fig. 9: Adding a label.

Next let us change the color to red and line style to dashed:

```
clf()
plot(x, y, 'r--', label = "Dashed red line")
legend()
show()
```

The output is shown in Fig. 10.

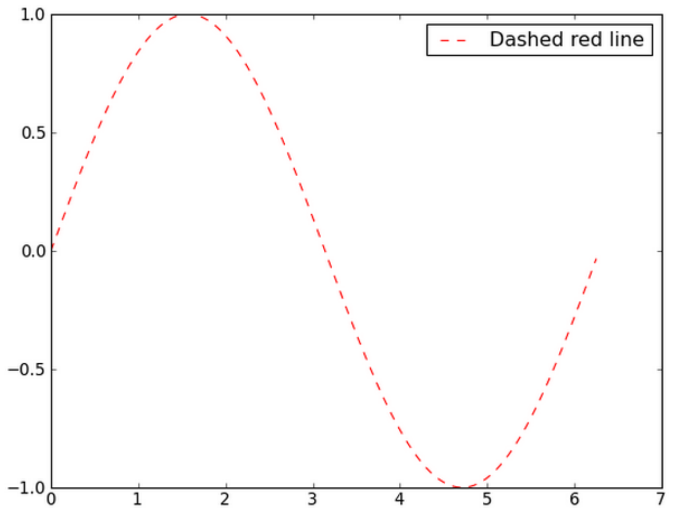


Fig. 10: Same graph using dashed red line.

The graph can be plotted using green color and small dots rather than a solid or dashed line:

```
clf()
plot(x, y, 'g.', label = "Dotted green line")
legend()
show()
```

The output is shown in Fig. 11.

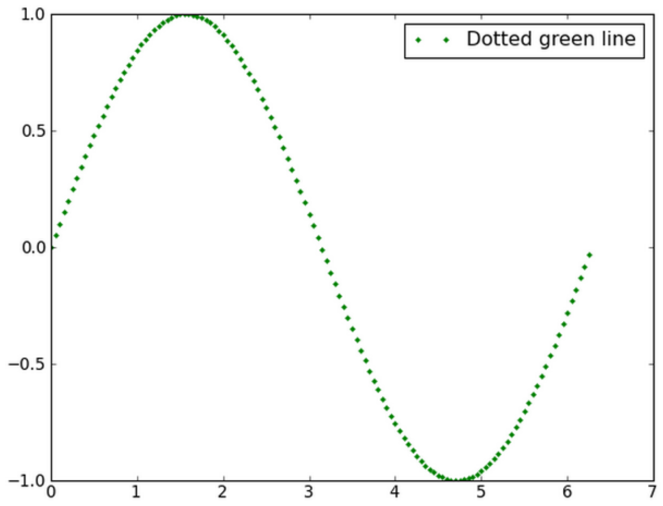


Fig. 11: Same graph using dotted green line.

Last let us stay with green color but make the dots larger:

```
clf()
plot(x, y, 'go', label = "Bigger green dots")
legend()
show()
```

The output is shown in Fig. 12.

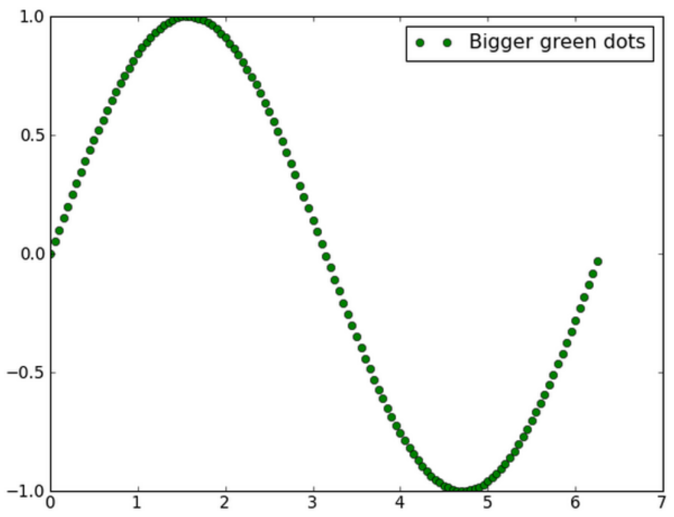


Fig. 12: Same graph using large green dots.

4.6 Scaling axes and showing grid

In the previous plots, function graphs were fitted into the display window, which means that the horizontal and vertical axes were scaled differently. This can be changed by including the `axis('equal')` command after calling `plot()`. Also, grid can be displayed by using the `grid()` command:

```
from numpy import *
from pylab import *
x = linspace(0, 2*pi, 100)
y = sin(x)
clf()
plot(x, y, label="sin(x)")
axis('equal')
grid()
legend()
show()
```

The output is shown in Fig. 13.

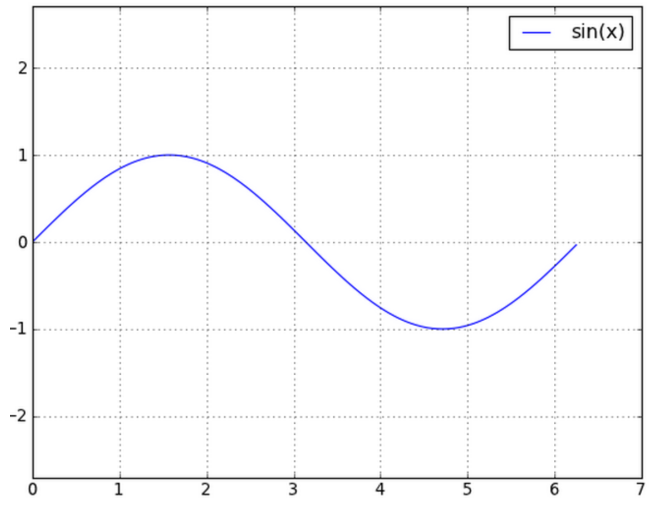


Fig. 13: Scaling axes equally and showing grid.

The scaling of axes can be returned to the automatic fit option via `axis('auto')` if needed.

4.7 Adjusting plot limits

Plot limits can be set using the `xlim()` and `ylim()` functions after `plot()`. For example, we can stretch the sine function from Fig. 13 to span the entire width of the canvas as follows:

```
from numpy import *
from pylab import *
x = linspace(0, 2*pi, 100)
y = sin(x)
clf()
plot(x, y, label="sin(x)")
xlim(0, 2*pi)
axis('equal')
grid()
legend()
show()
```

The output is shown in Fig. 14.

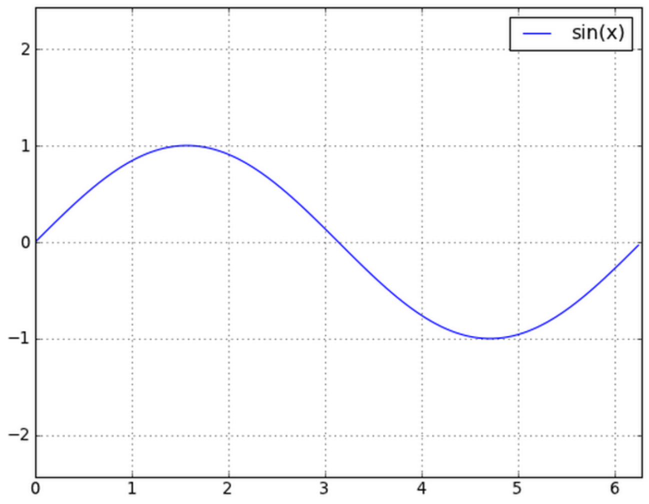


Fig. 14: Setting plot linits on the horizontal axis to 0 and 2π.

4.8 Plotting multiple functions at once

This can be done very easily, just do not use the `clf()` command between the plots and you can have as many graphs in one figure as needed. Let us do three:

```
from numpy import *
from pylab import *
x = linspace(0.5, 5, 100)
y1 = 1./x
y2 = 1. / (1 + x**2)
y3 = exp(-x)
axis=("equal")
clf()
plot(x, y1, label="y1")
plot(x, y2, label="y2")
plot(x, y3, label="y3")
legend()
show()
```

The output is shown in Fig. 15.

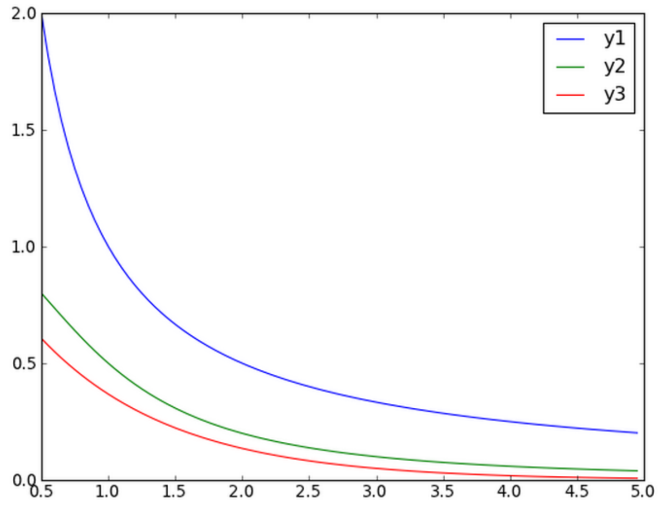


Fig. 15: Plotting graphs of functions $1/x$, $1/(1+x^2)$ and e^{-x} in interval $(0.5, 5)$.

The `plot()` command in Pylab is much more powerful, we just saw a small fraction of its functionality. For a complete list of options visit the Pylab page `http://www.scipy.org/PyLab`.

4.9 Plotting parametric 2D curves

The concept of plotting based on two arrays of x and y coordinates allows us to do much more than only plot graphs of functions of one variable. We can easily plot more general curves such as circles, spirals and others. Let us illustrate this on a spiral that is parameterized by

$$x(t) = t\cos(t), \quad y(t) = t\sin(t)$$

in the interval $(0, 10)$ for t. The complete code is

```
from pylab import *
from numpy import *
t = linspace(0, 10, 100)
x = t*cos(t)
y = t*sin(t)
clf()
plot(x, y)
show()
```

The output is shown in Fig. 16.

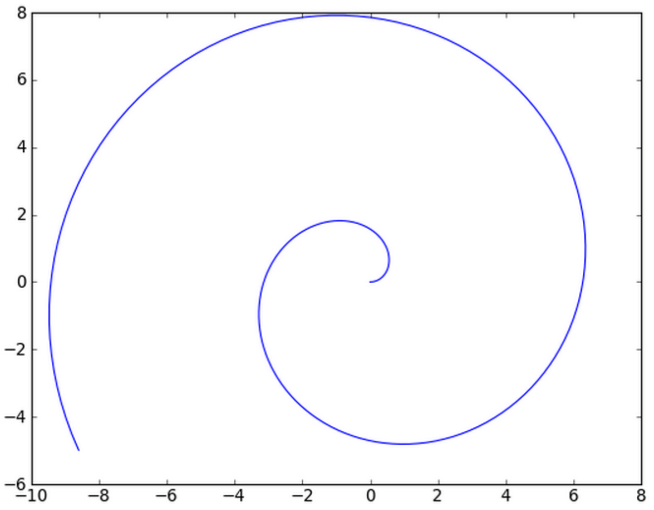


Fig. 16: Plotting a spiral.

4.10 Plotting parametric 3D curves

For 3D plots it is practical to use the `mplot3d` toolkit of the Python library Matplotlib. Let us begin with parametric 3D curves since this is analogous to how we handled planar curves in the previous paragraph. 3D curves are sequences of linearly interpolated 3D points represented via three arrays of x, y and z coordinates. As an example we will plot the curve $x(t), y(t), z(t)$ where

$$x(t) = (1 + t^2)\sin(2\pi t), \quad y(t) = (1 + t^2)\cos(2\pi t), \quad z(t) = t,$$

and where the parameter t lies in the interval $(-2, 2)$.

```python
# Import Numpy and Matplotlib:
from numpy import *
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# Set legend font size (optional):
mpl.rcParams['legend.fontsize'] = 15

# Setup the 3D plot:
fig = plt.figure()
ax = fig.gca(projection='3d')

# Define interval for parameter 't' and its
# division:
t = linspace(-2, 2, 100)

# Define the curve:
x = (1 + t**2) * sin(2 * pi * t)
y = (1 + t**2) * cos(2 * pi * t)
z = t

# Plot the curve and show the plot:
ax.plot(x, y, z, label='Parametric 3D curve')
ax.legend()
show()
```

The output is shown in Fig. 17.

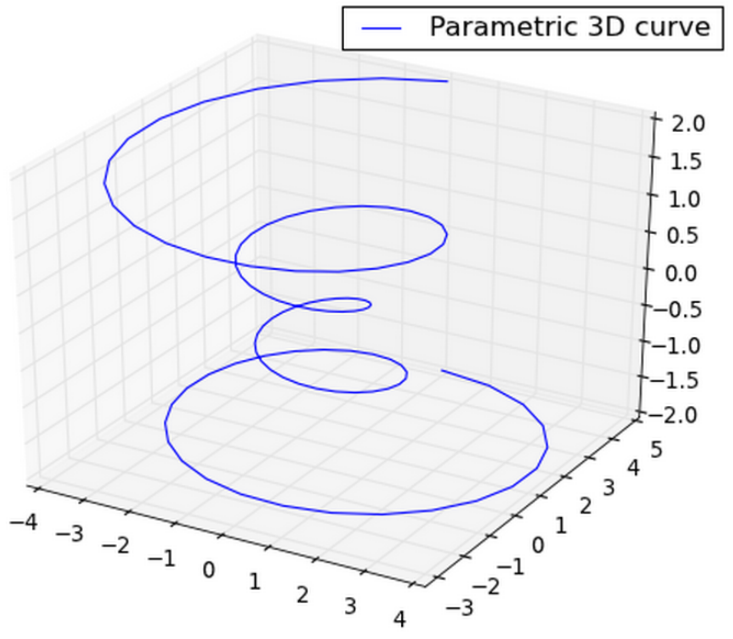


Fig. 17: Plotting a parametric 3D curve.

4.11 Plotting functions of two variables

There are several ways to plot graphs of functions of two variables, first let us do this with Matplotlib's `mplot3d` toolkit, then we will use WebGL. We will plot the graph of the function

$$f(x, y) = \sin(x)\sin(y)$$

in the square $(-\pi, \pi) \times (-\pi, \pi)$.

```
# Import Numpy and Matplotlib:
from numpy import *
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt

# Setup the 3D plot:
fig = plt.figure()
ax = fig.gca(projection='3d')
```

```
# Define intervals on the 'x' and 'y' axes and
# their divisions:
x = linspace(-pi, pi, 30)
y = linspace(-pi, pi, 30)

# Create Cartesian grid:
X, Y = meshgrid(x, y)

# Calculate values at grid points:
Z = sin(X) * sin(Y)

# Plot the data:
ax.plot_wireframe(X, Y, Z, rstride=1, cstride=1)

# Show the plot:
show()
```

The parameters `rstride` (row stride) and `cstride` (column stride) can be used to make the plot coarser when they are set to 2, 3, etc. The output is shown in Fig. 18.

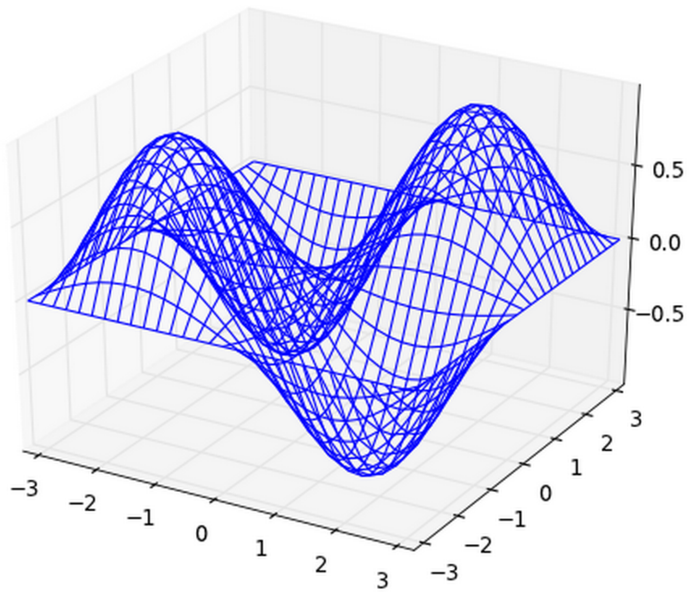


Fig. 18: Wireframe plot of the function $f(x, y)$.

Solid surface plot can be obtained by replacing in the above code `plot_wireframe` with `plot_surface`. The output is shown in Fig. 19.

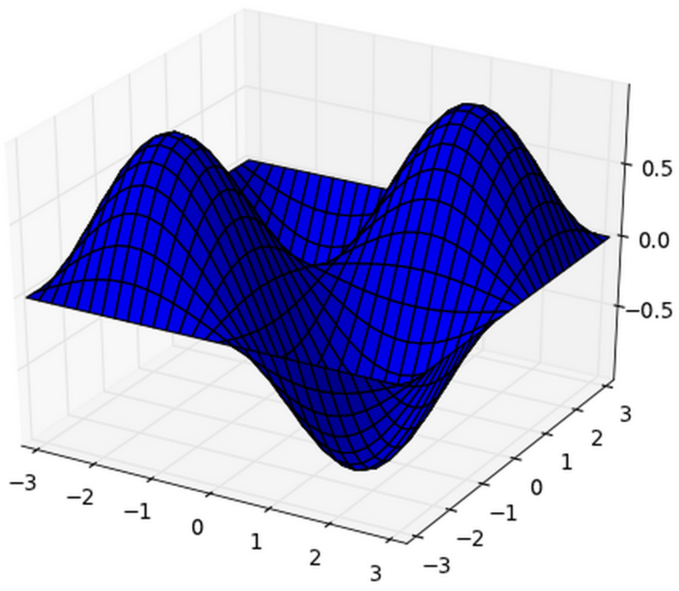


Fig. 19: Surface plot of the function $f(x, y)$.

Contour plot can be obtained by replacing in the above code the line

```
ax.plot_wireframe(X, Y, Z, rstride=1, cstride=1)
```

with

```
cset = ax.contour(X, Y, Z, num_contours)
ax.clabel(cset)
```

where `num_contours` is the number of contours to be shown. The output is shown in Fig. 20.

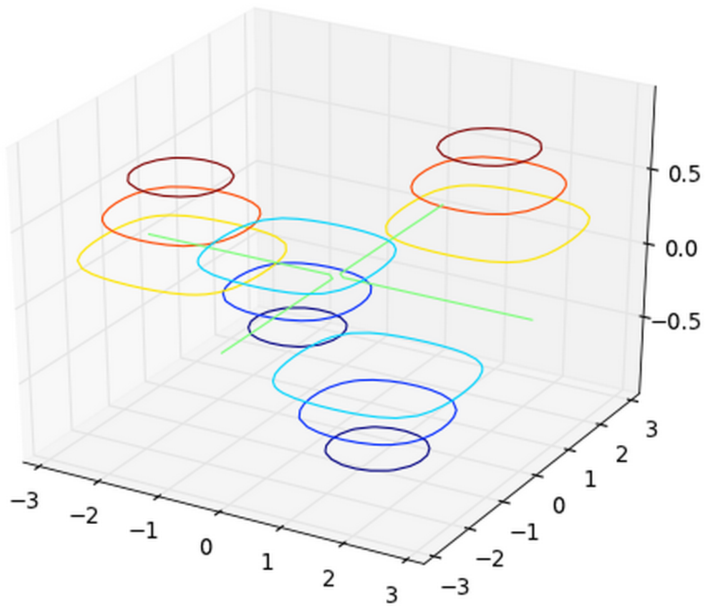


Fig. 20: Contour plot of the function $f(x, y)$.

We have only mentioned the most basic Matplotlib's functionality, for more options we refer to the tutorial at `http://matplotlib.sourceforge.net/mpl_toolkits/mplot3d`.

4.12 Plotting functions of two variables with WebGL

For this functionality, your browser has to support WebGL (most of modern browsers do, with the exception of Internet Explorer). See the introductory tutorial *Welcome to NCLab* on the page `http://femhub.com/nclab-tutorials/` for detailed instructions on how to enable WebGL. In the following example, we plot the graph of the function

$$f(x, y) = \sin\left(\sqrt{x^2 + y^2}\right)$$

in the square $(0, 10) \times (0, 10)$.

```
from numpy import sin, sqrt, arctan

# Define intervals on the x and y axes.
x0 = 0.0
x1 = 10.0
y0 = 0.0
y1 = 10.0
# Define the corresponding divisions.
nx = 100
ny = 100
# Define the function:
def f(x, y):
    return sin(sqrt(x**2 + y**2))
# Render the surface using WebGL.
lab.surface((x0, x1, nx), (y0, y1, ny), f)
```

The output is shown in Fig. 21.

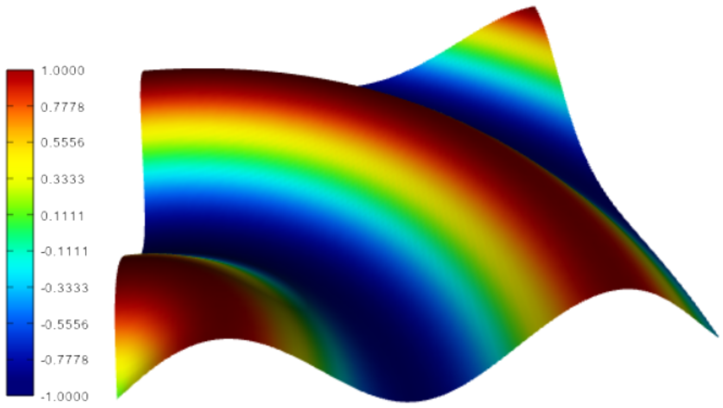


Fig. 21: Plotting functions of two variables with WebGL.

4.13 Plotting pie charts

Pie charts are an elegant way to report the breakdown of an ensemble into parts (percentages). We use the Pylab's function `pie` for that. The simplest example of a pie chart involves seven lines of code:

```
from pylab import *
data = [450, 550, 300]
labels = ['cats', 'dogs', 'mice']
clf()
axis('equal')
pie(data, labels=labels)
show()
```

The output is shown in Fig. 22.

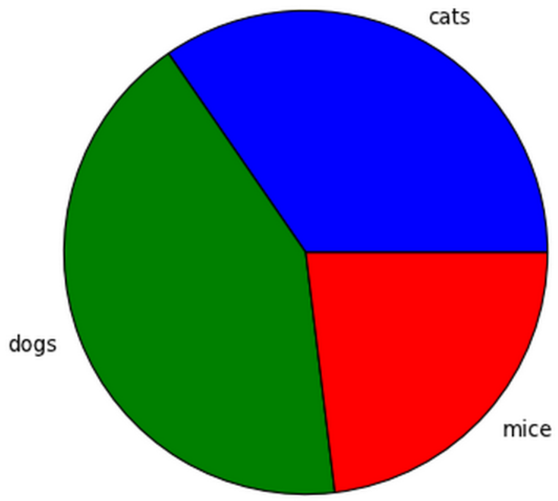


Fig. 22: Basic pie chart.

Next let us present a more advanced variant showing the production of an electronic company that sells TVs, computer monitors, cameras, printers and scanners. The script contains comments and additional details are explained under the code.

```
# Import libraries:
from pylab import *

# Remove eventual previous plots:
clf()

# We will have five sections:
labels = ['TVs', 'Monitors', 'Cameras', 'Printers', 'Scanners']

# Fractions in percent:
fractions = [15, 30, 5, 40, 10]

# Optional. "Monitors" section will be highlighted by moving
# out of the chart by 0.05:
explode = (0, 0.05, 0, 0, 0)

# Create the pie chart. The "autopct" parameter formats the
# numerical percentages. Shadow can be suppressed by setting
# it to False:
pie(fractions, explode=explode, labels=labels, \
    autopct='%1.1f%%', shadow = True)

# Create title:
color = [0.9, 0.9, 0.9]  # This is light-grey color.
title('Company\'s last year production', \
      bbox={'facecolor': color, 'pad': 20})

# Fetch the plot from the server:
show()
```

The output is shown in Fig. 23.

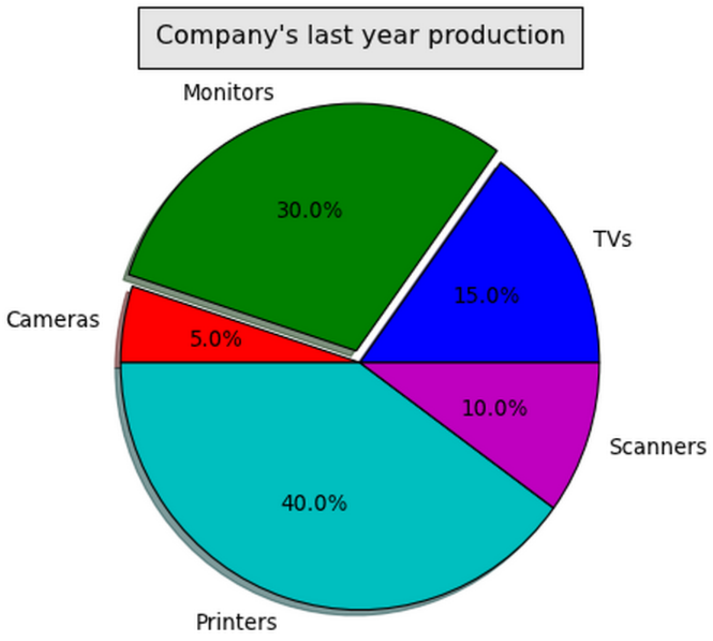


Fig. 23: Sample pie chart showing the breakdown of a company's production.

The `autopct` option determines the numerical format in which the numbers inside the pie chart will be displayed. The letter `f` stands for a floating-point number. The `1.1` means that one decimal digit will be present after the decimal point (`1.2` would print the percentages with two digits after the decimal point, etc.). The parameter `shadow` indicates whether a shadow will be used. Iy can be set to True or False. The `color` parameter in `title` defines a color using its RGB components which are between 0 (darkest) and 1 (lightest). The parameter `pad` defines padding.

4.14 Plotting bar charts

Bar charts are a great way to visualize time-dependent data such as, for example, the yearly growth of a company's revenues, number of daily visits to a web page, weekly rate of unemployment in the state, etc. The simplest bar chart involves 5 lines of code:

```
from pylab import *
data = [450, 550, 300]
clf()
bar([1, 2, 3], data)
show()
```

The array `[1, 2, 3]` specifies the positions on the horizontal axis where the bars begin. The default width of the bars is 0.8 and it can be adjusted via the parameter `width` of the `bar()` function. The output is shown in Fig. 24.

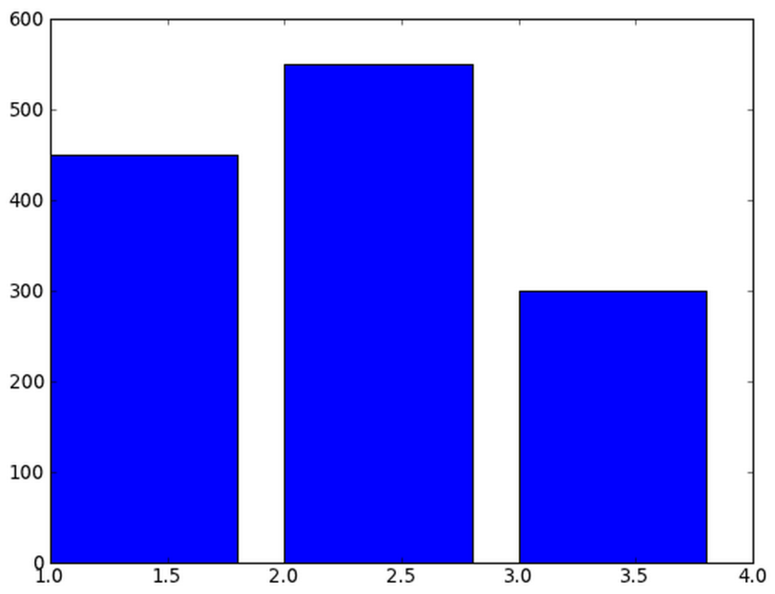


Fig. 24: Basic bar chart.

Next let us present a more advanced bar chart that shows a company's growth both in the total number of employees and the yearly increments.

```
# Import libraries:
from numpy import max
from pylab import *

# Number of years:
N = 5

# Totals and increments:
total = (20, 25, 32, 42, 61)
increment = [5, 7, 10, 19, 30]
```

```
# X-locations and width of the bars:
ind = np.arange(N)
width = 0.4

# Create figure and subplot:
fig = figure()
ax = fig.add_subplot(111)

# Define the bars:
rects1 = ax.bar(ind, total, width, color='y')
rects2 = ax.bar(ind + width, increment, width, color='b')

# Add descriptions:
ax.set_ylabel('Numbers')
ax.set_title('Number of employees')
ax.set_xticks(ind + width)
ax.set_xticklabels( ('2008', '2009', '2010', '2011', '2012') )

# Add numbers above bars:
def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2., \
                height + 0.02 * max(total), '%d'%int(height), \
                ha = 'center', va = 'bottom')
autolabel(rects1)
autolabel(rects2)

# Fetch image from server:
show()
```

The output is shown in Fig. 25.

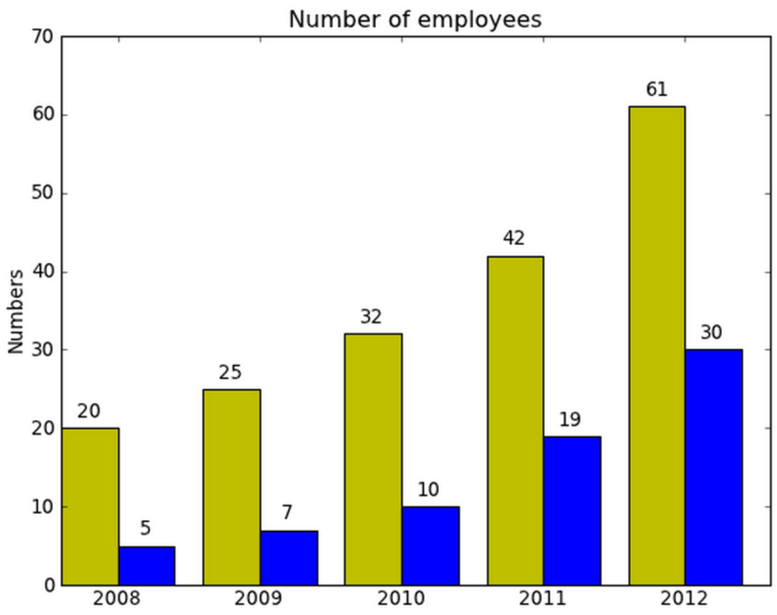


Fig. 25: Bar chart showing a company's growth.

Let us stop here for the moment, so that we can explore other aspects of Python programming. The functionality provided by Pylab/Matplotlib is so extensive that we could spend easily several months experimenting with it. Many tutorials and examples are available online. In particular, we encourage the reader to explore the original tutorial page at `http://matplotlib.org`.

5 Variables

5.1 Objectives

- Learn to assign values to variables.
- Understand dynamic type interpretation.
- Learn more about *local* and *global* variables.
- Understand why global variables should be avoided.
- Learn about *shadowing* of variables.

The reason for using variables is to store useful information for later use. This is the same across all programming languages. The way we work with variables in Python is almost the same as in Karel the Robot – all that we learned about variables with Karel applies here. Moreover, Python has interesting new features that we will explore in this section.

5.2 Creating variables

Variables can store Boolean values (True of False), text strings, integers, real numbers, and more. In Python, the type of a variable does not have to be indicated in advance (such as for example in C/C++). The type of a variable will be figured out at runtime based on the value that is stored in it. This is called *dynamic type interpretation*. Let us look at the following code:

```
a = 5
print a
a = "Hi!"
print a
a = 'Hello!'
print a
a = True
print a
```

The output is

```
5
Hi!
Hello!
True
```

Initially, the type of the variable `a` was an integer, then it was changed to a text string, and finally it became a Boolean variable. Being able to do things like this makes Python programming very flexible. However, we should not abuse our freedoms unless there is a valid reason for it.

Avoid abuse of dynamic type interpretation. A variable, once created for a purpose, should serve that purpose. If you need another variable for something else, just create another variable.

Also:

Have names of variables reflect their contents. For example, `phone_number` is a good name for a variable where phone numbers are stored. Using the name `x17` for such a variable is not against any rules, but your code becomes unreadable.

In general, do not be too inventive with names of variables nor with algorithms. Make sure that your code is crystal clear. Write comments as much as you can. You should not leave anything for the reader to figure out by himself/herself. Otherwise no one will understand your code – not even yourself when you return to it after some time.

5.3 Initializing multiple variables at once

In Python, several variables can be created at the same time and initialized with the same value:

```
x = y = z = 0.0
print "x =", x
print "y =", y
print "z =", z
```

Output:

```
x = 0.0
y = 0.0
z = 0.0
```

Any variable must be defined (have a value assigned) before it can be used. If we try to use a variable that the interpreter has not seen before, such as `w` in

```
x = w / 2.
```

then an error message will follow:

```
Traceback (most recent call last):
  File "<nclab>", line 2, in <module>
NameError: name 'w' is not defined
```

Initializing a new variable with existing one(s) is indeed possible:

```
w = (x + y + z + 1) / 2.
print "w =", w
```

yields

```
w = 0.5
```

5.4 Changing values of variables

The simplest way to increase the value of a numerical variable in Python by a given number is to use the '+=' command:

```
v = 1
v += 3
print "v =", v
```

Output:

```
v = 4
```

We can also subtract a number from a numerical variable:

```
v -= 1
print "New value of v is", v
```

Output:

```
New value of v is 3
```

We can multiply a numerical variable with a number:

```
v *= 4
print "Now v is", v
```

Output:

```
Now v is 12
```

And finally, we can divide a numerical variable with a number:

```
v /= 6
print "Finally, v is", v
```

Output:

```
Finally, v is 2
```

As we already saw, it is possible to use existing variables to assign a value to a new variable:

```
a = 1
b = 2.5
c = 0.5
d = (a + b) / c
print "d =", d
```

Output:

```
d = 7.0
```

Of course, apples cannot be mixed with oranges. When we try to add a number to a text string,

```
a = "My car is a Ferrari."
b = 3.5

c = a + b
```

then the interpreter rightfully complains:

```
Traceback (most recent call last):
  File "<nclab>", line 4, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

5.5 Local and global variables

Variables defined inside a function are *local* to that function. If we try to access such a variable outside the function, even though the function was already called, the variable is unknown. Look at the following code:

```
def add(a, b):
    c = a + b
    return c

print "Sum of 3 and 4 is", add(3, 4)
print c
```

The output is:

```
Sum of 3 and 4 is 7
Traceback (most recent call last):
  File "<nclab>", line 6, in <module>
NameError: name 'c' is not defined
```

In summary:

- *Keep variables as local as possible*. Mistakes such as using a variable in one part of the code and unintentionally overwriting it in some other part belong to the most difficult ones to find.
- *Avoid very long blocks of code*. If a function is longer than a few dozens of lines, probably it is about time to break it down to two or more simpler ones. Keeping functions short limits the space where variables can roam, making code healthier.

5.6 How to get in trouble

Once a variable is used in the main program, then it is *global*, meaning that you can use it in any function without passing it as an argument. Like in the following code:

```
val = 5.0

def msg():
    print "val =", val

msg()
```

Yuck! This code is not only ugly but also a source of future problems. Imagine that your programming project grows and the definition of the variable `val` is moved to a different module. Then, at some point you realize that you should get rid of the global variable `val` because using global variables is not a good programming practice. But when you remove it, the function `msg()` stops working!

On the other hand, it is very easy to write the code in a clean fashion:

```
def msg(out):
    print "val =", out

val = 5.0

msg(val)
```

Now the function `msg()` is well defined – one does not need to look elsewhere in the program to understand what it does. The output is:

```
val = 5.0
```

In summary:

Avoid using global variables in functions. Any function that operates on some data should always obtain all the data as its parameters.

5.7 Variable shadowing

Sometimes we may end up having two different variables of the same name in the code. Usually this happens in large software projects, but we can only illustrate it on a short program:

```
c = 5

def add(a, b):
    c = a + b
    print "The value of c inside is", c
    return c

add(1, 2)
print "The value of c outside is", c
```

The output is

```
The value of c inside is 3
The value of c outside is 5
```

We say that the global variable `c` defined in the first line of the code is in the function `add()` *shadowed* by the local variable `c`. Simply said, local variables in functions have priority. What actually happens when the above code is interpreted, is that the interpreter gives to the two `c` variables different names. For the interpreter, they are two completely different objects.

5.8 Overriding constants

In Python, constants such as `pi` or `e` are in fact just global variables and in principle there is no problem redefining their values in our program. Of course this is not a good programming practice as our code becomes confusing. A variable `pi` should contain the value of π as everyone expects.

But, mistakes happen. If we define a function where we create a local variable `pi`, this is in fact another example of variable shadowing and our code will work fine:

```
from numpy import sqrt, pi
def hypotenuse(x, y):
    pi = sqrt(x**2 + y**2)
    return pi
print hypotenuse(3, 4)
print pi
```

The output is

```
5.0
3.14159265359
```

6 Logic and Probability

6.1 Objectives

- Review Boolean expressions, operations, and variables.
- Learn the Monte Carlo method of scientific computing.
- Encounter the `for` loop in Python for the first time.

Logic is always a useful review topic, so let's return to it briefly. Fortunately, logic in Python is virtually the same as it is in Karel the Robot.

6.2 `True` and `False`

Python has a built-in data type, `bool`, for expressing True and False values. This is short for Boolean. Certain expressions in Python, those that use *comparison operators*, evaluate to `True` or `False`. These expressions are used to determine the relationship between two values.

Let's begin with creating an integer variable and printing it:

```
a = 1
print a
```

Output:

```
1
```

Now type

```
print a > 0
```

Output:

```
True
```

In the above example, > is a comparison operator. It is used to evaluate whether the value on the left is greater than the value on the right.

Although the syntax looks a bit unusual, this is a correct Python code – since `a` is equal to `1`, the Boolean expression `a > 0` is `True`. This is the value that was printed by the interpreter.

We can try it the other way round:

```
print a < 0
```

Output:

```
False
```

`True` and `False` are the two possible values of logical (Boolean) expressions. In the next paragraph we will learn to store them in variables, and in Paragraph 6.4 we will use them with logical operations.

6.3 Boolean variables

Similar to numbers and strings, Boolean values can be stored in variables. For example:

```
b = a < 0
print b
```

Output:

```
False
```

Boolean expressions involving numbers often contain the following operators:

Symbol	Meaning
>	greater than
>=	greater than or equal to
<=	less than or equal to
<	less than
==	equal to
!=	not equal to
<>	not equal to (same as !=)

A common source of confusion for new Python programmers is the difference between the '==' and '=' operators. The *assignment operator* '=' is used to assign a value to a variable. The *comparison operator* '==' is used to compare two values and determine if they are equal.

6.4 Boolean operations

When solving real-life problems, our algorithms often contain conditions that include more than one logical expression. For example, consider an algorithm that calculates the area of a triangle with edge lengths a, b and c. These three numbers are provided by the user. Clearly, the result is undefined if any of the three numbers is negative or zero. It is also undefined if the three numbers are positive but do not satisfy the triangle inequality (the sum of the lengths of two shorter edges must be greater than the length of the longest edge). Hence, before we calculate the area, we need to introduce the following logical variables:

$A = a > 0$
$B = b > 0$

$C = c > 0$
$D = a + b > c$
$E = a + c > b$
$F = b + c > a$

The numbers a, b, c form an admissible input if the following Boolean expression is True:

$$A \textit{ and } B \textit{ and } C \textit{ and } (D \textit{ or } E \textit{ or } F)$$

We can see that logical operations indeed are useful. Now let us revisit them one by one, starting with logical *and*:

If A and B are logical expressions, then A *and* B is True only if A as well as B are True, otherwise it is False:

```
a = 1
v1 = a > 0
v2 = a < 5
print v1 and v2
```

Output:

```
True
```

Logical *or* of expressions A and B is True if at least one of A, B is True. Otherwise it is False:

```
v3 = a > 10
print v1 or v3
```

Output:

```
True
```

Negation is a logical operation that changes the value of expression A to False if A is True, and vice versa:

```
v4 = not v1
print v4
```

Output:

```
False
```

Sometimes logical expressions can be more complicated, but this is no problem as we always can use round brackets:

```
v5 = not ((v1 and v2) or (v3 and not v4))
```

6.5 Monte Carlo methods

Monte Carlo methods are popular methods of scientific computing. They employ large numbers of random values to calculate approximately results that are difficult or impossible to obtain exactly. Sounds complicated? On the contrary, Monte Carlo methods are very simple! For demonstration, let us calculate the area of an ellipse.

An ellipse centered at the origin $(0, 0)$ whose major and minor half-axes a and b coincide with the x and y axes is given by the formula

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1.$$

In other words, all points (x, y) in the plane that satisfy this equation form the ellipse. It is also true that all points in the plane that satisfy

$$\frac{x^2}{a^2} + \frac{b^2}{y^2} < 1$$

form the interior of the ellipse. Obviously the ellipse is enclosed in the rectangle $(-a, a) \times (-b, b)$. The idea of the Monte Carlo algorithm is very simple: We will fill the rectangle with random points, and count how many of them fall into the ellipse. The latter number will be denoted by m. With a sufficiently large n, such as $100,000$ or more, the ratio of m and n will be close to the ratio of the area of the ellipse E and the area of the rectangle $R = 4ab$. Hence we obtain a simple formula

$$\frac{m}{n} \approx \frac{E}{R}$$

from which we can express the area of the ellipse as

$$E \approx R\frac{m}{n}.$$

Now we can start coding!

The program

First, let us write a Boolean function `is_inside(x, y, a, b)` that returns `True` if the point (x, y) lies inside the ellipse, and `False` otherwise:

```
# Check whether point (x, y) is
# inside an ellipse with half-axes
# a and b:
def is_inside(x, y, a, b):
    # Avoid eventual problems with
    # integer-integer division:
    x = float(x)
    y = float(y)
    # Check if point is inside:
    if (x/a)**2 + (y/b)**2 < 1:
        return True
    else:
        return False
```

Notice the way we make sure that the divisions x/a and y/b yield correct results even if all these variables are integers.

In the next step we import the `random()` function and write the Monte Carlo algorithm. For this, we need to introduce the counting loop in Python first.

The `for` loop

Recall that to make Karel the Robot increase the value of the variable `a` by one `n` times, we wrote:

```
repeat n
    a += 1
```

Python does not have the `repeat` command but the same can be achieved by

```
for i in range(n):
    a += 1
```

The `for` loop in Python is much more powerful than the `repeat` loop in Karel and it will be discussed in detail in Section 10. Now let us return to our Monte Carlo method:

```
# Import libraries:
from random import random

# Monte Carlo algorithm for
# ellipse area calculation:
def area(a, b, n):
    # Initialize m by zero:
    m = 0
    # Repeat n times:
    for i in range(n):
        # Random number between 0 and 1:
        x = random()
        # Transform it between (-a, a):
        x = (x - 0.5) * 2 * a
        # Random number between 0 and 1:
        y = random()
        # Transform it between (-b, b):
        y = (y - 0.5) * 2 * b
        # If the point (x, y) lies in
        # ellipse, increase m by one:
        if is_inside(x, y, a, b):
            m += 1
    # Area of rectangle (-a, a)x(-b, b):
    print m
    R = 4.0 * a * b
    # Calculate the approximate area:
    E = R * m / n
    # Return it:
    return E
```

The code is ready and we can test it. Let us do it for $a = 2, b = 1$ and various values of n:

```
# Test the code:
a = 2
b = 1
print "a =", a
print "b =", b
n = 10000
A = area(a, b, n)
print "For n =", n, "area =", A
n = 100000
A = area(a, b, n)
print "For n =", n, "area =", A
n = 1000000
A = area(a, b, n)
print "For n =", n, "area =", A
```

The corresponding output is:

```
a = 2
b = 1
7836
For n = 10000 area = 6.2688
78473
For n = 100000 area = 6.27784
785078
For n = 1000000 area = 6.280624
```

This is not a bad result, our approximation is accurate to two decimal digits! This can be checked using a simple formula for the area of an ellipse, that you can find on Wikipedia. Do this as your homework! According to this formula, the exact area of the above ellipse is $2\pi = 6.283185307179586$.

7 Conditional Loop

7.1 Objectives

- Review the difference between counting and conditional loops.
- Review what we know about conditions from Karel the Robot.
- Learn about the new `elif` statement in Python.
- Learn how to terminate a loop with the `break` statement.
- Learn to use the `continue` statement to skip the rest of the loop.

– Use the `while` loop to make another trip to scientific computing.

7.2 Counting and conditional loops

Every procedural programming language has two types of loops that serve different purposes: The *counting loop* (`repeat` in Karel, `for` in Python and many other languages) is used when the number of cycles is known in advance. Such as when we need to write the first 10 even integers.

On the contrary, the *conditional loop* (`while` in Karel, Python, and many other languages) is used when it is not known how many cycles will be needed. Instead of the number of repetitions, the conditional loop has a condition that must be fulfilled for the loop to go on. It is used, for example, when we need to keep halving a real number while the result is greater than 10^{-5}.

We first saw the `for` loop in Paragraph 6.5 about the Monte Carlo method, it will be used with text strings in Section 8, and additional details will be provided in Section 10. In the following paragraphs of this section we will focus on conditions and the conditional loop.

In summary:

Always use the counting loop when you know how many repetitions will be done. Only if you do not know, use the conditional loop.

7.3 Conditions and the `elif` statement

Conditions in Python are almost the same as in Karel the Robot. The `if - else` statement has been used in the previous sections, so there is no need to explain it here again. But there is one practical new feature in Python – the `elif` statement that simplifies cases with more than two options. If you think that "elif" sounds a lot like "else if" then you are absolutely right! In general

```
if <logical_expression_1>:
    <do_action_1>
elif <logical_expression_2>:
    <do_action_2>
else:
    <do_action_3>
```

means exactly the same as

```
if <logical_expression_1>:
    <do_action_1>
else:
    if <logical_expression_2>:
        <do_action_2>
    else:
        <do_action_3>
```

Clearly, the latter involves more indentation. The advantage of the `elif` statement becomes more visible as the number of cases grows. Let us therefore show a concrete example with five cases:

Five boxes of apples

Imagine that we have five wooden boxes: Box A is for apples that weight under 5 ounces, box B for apples that weight at least five but less than 10 ounces, box C for apples that weight at least 10 but less than 15 ounces, etc. Apples weighting 20 ounces or more will all be put into box E. We need to write a function `box(weight)` that, given the weight of an apple, chooses the correct box for it and returns the corresponding letter:

```
def box(weight):
    if weight < 5:
        return "A"
    elif weight < 10:
        return "B"
    elif weight < 15:
        return "C"
    elif weight < 20:
        return "D"
    else:
        return "E"
```

Note the mandatory colon ':' after every `if`, `elif` and `else` statement. Here is how the same code would look like without the `elif` statement:

```
def box(weight):
    if weight < 5:
        return "A"
    else:
        if weight < 10:
            return "B"
        else:
            if weight < 15:
                return "C"
            else:
                if weight < 20:
                    return "D"
                else:
                    return "E"
```

In summary:

The `elif` statement simplifies conditions with multiple cases.

7.4 The `while` loop

The `while` loop in Python has the following general format:

```
while <logical_expression>:
    <sequence_of_commands>
```

There is a mandatory indent for the body of the loop, same as for conditions and functions. Moreover note the mandatory color `':'` after the Boolean expression.

As a simple example, let us generate and print random numbers between zero and one as long as they are less than 0.75. If a number is greater or equal to 0.75, stop. Since we do not know how many repetitions will be done, the conditional loop is the proper way of handling this task:

```
from random import *
a = random()
while a < 0.75:
    print a
    a = random()
```

Sample output:

```
0.383771890647
0.068637471541
0.0850385942776
0.682442836322
0.168282761575
0.121694025066
0.185099163429
0.606740825997
0.209771333501
0.186191737708
0.608456569449
0.591009262698
```

Running the program once more gives another output:

```
0.637423013735
0.533383497321
0.240830918488
0.343139647492
0.718014397238
0.698931548709
0.363558456088
```

In summary:

The `while` loop should be used only when the number of repetitions is not known in advance.

7.5 The `break` statement

The `break` statement can be used to exit a loop at any time, no questions asked. If multiple loops are embedded, then it only exits the closest one that contains it. For illustration, let us rewrite the above program to use the `break` statement:

```
from random import *
while True:
    a = random()
    if a >= 0.75:
        break
    print a
```

Sample output:

```
0.486690683395
0.232518153456
0.38084453626
```

The `break` statement can be useful sometimes and we should know about it. However, let's keep the following in mind:

Combining `while True:` with the `break` statement in this way is quite popular among Python programmers. The technique should not be abused though since it is not a perfect example of structured programming – it is just a shortcut.

7.6 The `continue` statement

Sometimes we know that finishing all commands in the loop's body is not necessary. Python has the `continue` command for this. To illustrate its use, let us write a program that will generate triplets of random numbers between zero and one until all three of them are greater than 0.9. We also want to know how many attempts were needed.

Clearly, after the first number is generated and it is less or equal to 0.9, it does not make any sense to generate the other two. That's when we use the `continue` command. If the first number is greater than 0.9 but the second one is less or equal to 0.9, it is time to use the `continue` command once again. Here is the code:

```
from random import *
counter = 0
while True:
    counter += 1
    a1 = random()
    if a1 <= 0.9:
        continue
    a2 = random()
    if a2 <= 0.9:
        continue
    a3 = random()
    if a3 <= 0.9:
        continue
    print "Finally the desired triplet:"
    print a1, a2, a3
    print "It took", counter, "attempts to get it!"
    break
```

Sample output:

```
Finally the desired triplet:
0.921077479893 0.956493808495 0.917136354634
It took 1168 attempts to get it!
```

7.7 Abusing the `while` loop

The following code has seen the light of the world too many times:

```
counter = 0
while counter < 20:
    # Do something with the counter, here
    # for simplicity we just print it:
    print counter
    # Increase the counter:
    counter += 1
```

Although there is nothing syntactically wrong with this code, and it even produces the desired result, it is an example of bad programming. The person who wrote such a code probably did not understand why programming languages have two types of loops. The code emulates a counting loop and whoever reads it, will be confused. Why was the `while` loop used when the number of repetitions was known in advance? The correct code for this situation is:

```
for counter in range(20):
    print counter
```

7.8 Next trip into scientific computing

As our programming skills get stronger, we are able to tackle more difficult problems. Now it is our task to find an angle x that is equal to its own cosine. In other words, we need to solve the equation

$$\cos(x) = x.$$

This equation cannot be solved on paper. But we will see that it can be solved with the help of the computer rather easily. As the first step, let us visualize the graphs so that we know where approximately the solution can be expected:

```
from numpy import pi, cos
from pylab import *
x = linspace(0, pi/2, 100)
y = cos(x)
z = x
clf()
plot(x, y, label="cos(x)")
plot(x, z, label="x")
legend()
show()
```

The output is shown in Fig. 26.

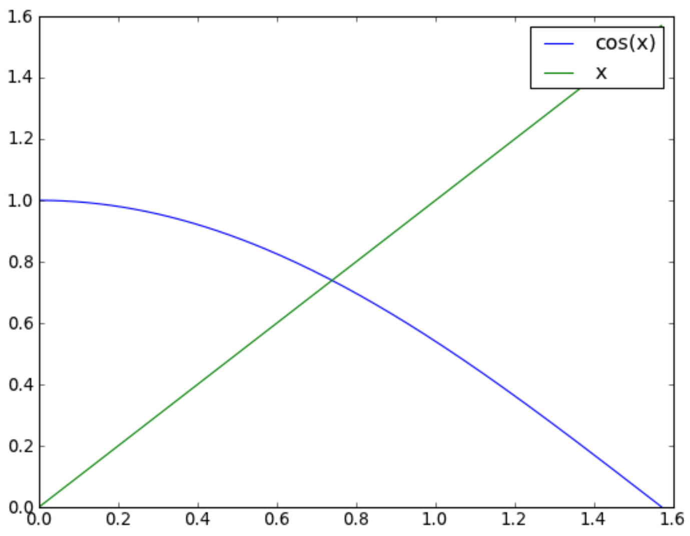


Fig. 26: Graphs of the functions $\cos(x)$ and x in the interval $(0, \pi/2)$.

The solution that we are after is the value of x where the two graphs intersect. Looking at the picture, clearly the solution is somewhere between 0.6 and 0.8. An engineer would not be extremely happy with such an answer though – we need to make this guess much more accurate.

Looking at the graphs again, we can see that in order to reach the intersection point, we can depart from zero and march with very small steps to the right while the value of $\cos(x)$ is greater than the value of x. We stop as soon as $\cos(x)$ becomes less than x. Clearly, the `while` loop needs to be used as we do not know exactly how many steps will be taken.

Let's do this. Our step will be called `dx`, and our marching point on the x-axis will be called simply `x`. We will also count steps via the variable `n`. Choosing `dx = 1e-6` will give us the result accurate to five decimal digits:

```python
from numpy import cos
x = 0
dx = 1e-6
n = 0
while cos(x) > x:
    x += dx
    n += 1
print "Result is approximately", x
print "Steps made:", n
```

Output:

```
Result is approximately 0.739086
Steps made: 739086
```

In other words, the `while` loop ran 739086 times! Can you imagine doing this on your pocket calculator?

It is worth mentioning that the numerical method that we used was rather naive and in fact it took quite a bit of CPU time. Scientific computing is an exciting field where researchers develop new methods that can solve problems like this more accurately and using less CPU time. In particular, for our problem there are much more powerful methods that can do the same or better job using just five or six steps (!). One of them is the *Newton's method*. If you like what we did here, we encourage you to explore more math functionality and in particular numerical methods in NCLab. There are displayed projects for the Newton's method and other methods to solve nonlinear equations and nonlinear equation systems.

8 Strings

8.1 Objectives

- Learn basic operations with single and multiline strings.
- Learn to use quotes and backslashes, and to concatenate and repeat strings.
- Learn to refer to letters by indices, parse strings, and slice them.

By a *string* we mean a text surrounded by double or single quotes, such as `"this is a string"` or `'this is a string as well'`. Strings are useful to make outputs more informative, but they have many other uses as well. For example, they can represent data in databases such as in a phone book. We need to understand them well, as well as various operations that we can do with them.

8.2 Using quotes

Since we have already seen usage of simple strings before, let us now understand how to use quotes in strings. The safest way is to use them with a backslash:

```
print "I said \"yes\"."
```

Output:

```
'I said "yes".'
```

Another example:

```
print "It doesn\'t matter."
```

Output:

```
"It doesn't matter."
```

8.3 Multiline strings and backslashes

If we want to use multiline strings, the best way is to enclose them in triple quotes:

```
edgar = """
Once upon a midnight dreary, while I pondered weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
`'Tis some visitor,' I muttered, `tapping at my chamber door -
Only this, and nothing more.'
"""
print edgar
```

Output:

```

Once upon a midnight dreary, while I pondered weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
`'Tis some visitor,' I muttered, `tapping at my chamber door -
Only this, and nothing more.'

```

Yes, there is one empty line on top and one on bottom. The reason is that Python inserted newline symbols there. If we want to avoid these empty lines, we have to include a backslash after the first triple quote, and also at the end of the last line before the closing triple quote. The backslash prevents Python from inserting a newline symbol into the string. Let us show one more example on backslashes. The string

```
"""\
This is the first line,
and this is the second line.\
"""
```

will render as

```
'This is the first line,\nand this is the second line.'
```

When a backslash is included at the end of the first line of text,

```
"""\
This is the first line,\
and this is the second line.\
"""
```

we obtain

```
'This is the first line,and this is the second line.'
```

As we see, we need to watch for empty characters. Inserting one before the second backslash

```
"""\
This is the first line, \
and this is the second line.\
"""
```

brings us to the desired outcome:

```
'This is the first line, and this is the second line.'
```

8.4 Concatenation and repetition

Strings can be concatenated (glued together) with the '+' operator, and repeated with the '*' operator. For example,

```
word = 'Help' + 'me!'
print "I yelled" + 3 * word
```

yields

```
I yelledHelpme!Helpme!Helpme!
```

Again, empty spaces matter. So let's try again:

```
word = '\"Help' + ' me!\" '
print "I yelled " + 3 * word
```

will render

```
I yelled "Help me!" "Help me!" "Help me!"
```

8.5 Referring to letters by their indices

Individual letters forming a string can be accessed via indices. The indices start from zero. It is also handy to use the index `-1` for the last index, `-2` for the one-before-last etc. Typing

```
word = "breakfast"
print "First character:", word[0]
print "Second character:", word[1]
print "Last character:", word[-1]
print "One-before-last character:", word[-2]
```

produces the following output:

```
First character: b
Second character: r
Last character: t
One-before-last character: s
```

Before we move on, make sure that you remember that

Indices in Python start from zero.

8.6 Parsing strings with the `for` loop

The length of a string is obtained using the function `len()`. For illustration, the code

```
word = "breakfast"
n = len(word)
print "Length of the string:", n
```

yields

```
Length of the string: 9
```

Since every string has a known length, the right loop to parse strings is the `for` loop. The following code parses the string `word` letter by letter and prints all the letters:

```
word = "breakfast"
for c in word:
    print c
```

The same can be achieved using the `range()` function that we already know from Paragraph 6.5:

```
word = "breakfast"
n = len(word)
for i in range(n):
    print word[i]
```

Output:

```
b
r
e
a
k
f
a
s
t
```

To see that the two above programs are equivalent, we need to understand a bit more about the `for` loop. First, a text string is a sequence of characters. The `for` loop can parse sequences using the general scheme `for <element> in <sequence>`. The function `range(n)` creates a sequence of integer numbers `0`, `1`, ..., `n-1`. Let's try this with `n = 3`:

```
print range(3)
```

Output:

```
[0, 1, 2]
```

Another example:

```
print range(10)
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

So, `for i in range(n)` goes through integer indices `i = 0,1`,...,`n-1` and for each one, `word[i]` is the next character in the text string.

8.7 Slicing strings

Python makes it easy to access substrings – this is called *slicing*:

```
w1 = "bicycle"
w2 = w1[0:3]
print w2
```

Output:

```
bic
```

Omitting the first index in a slice defaults to zero:

```
w3 = w1[:2]
print w3
```

Output:

```
bi
```

Omitting the second index defaults to the length of the string:

```
w4 = w1[2:]
print w4
```

Output:

```
cycle
```

This is enough about strings for now, but we will return to them again later. In particular, Python provides many useful built-in operations with text strings that will be shown in Paragraph 12.7

9 Tuples, Lists, and Dictionaries

9.1 Objectives

– Learn to store data in tuples, lists, and dictionaries.
– Learn the differences between these data types, and their best use cases.
– Learn useful built-in functions to operate with these data structures efficiently.

Tuples, lists, and dictionaries are the most important data structures in Python, and the language offers a wide range of functionality to work with them efficiently. They become especially powerful in combination with the `for` loop that was already mentioned in Paragraphs 6.5 and 8.6, and that will be discussed in more detail in Section 10.

9.2 Tuples

Tuples are perfect for sequences of data that does not change – such as the names of weekdays, or names of months. To define a tuple, enclose comma-separated items in round brackets:

```
months = ('January', 'February', 'March', 'April', 'May', \
'June','July', 'August', 'September', 'October', 'November', \
'December')
```

All items in this tuple are text strings, but a tuple can be heterogeneous – it can contain text strings, integers, real numbers, other tuples, etc.

Working with tuples is a lot similar to working with text strings. To begin with, items in a tuple can be referenced by indices. Remember that indices always start from zero:

```
print "First month:", months[0]
print "Second month:", months[1]
print "Third month:", months[2]
```

Output:

```
First month: January
Second month: February
Third month: March
```

As in text strings, index `-1` refers to the last item, `-2` to the one before last, etc.:

```
print "One before last month:", months[-2]
print "Last month:", months[-1]
```

Output:

```
One before last month: November
Last month: December
```

We can *slice* tuples same as we sliced text strings:

```
months[2:5]
```

Output:

```
('March', 'April', 'May')'
```

As in text strings, the length of a tuple is obtained using the function `len()`:

```
len(months)
```

Output:

```
12
```

9.3 Lists

Lists are similar to tuples, and all indexing and slicing operations work in the same way. The only difference is that we can change a list by adding and deleting entries, we can reorder items in a list, etc. A tuple, once created, cannot be changed.

A list is created by enclosing its items into square brackets. For example, an empty list `L` is defined as follows:

```
L1 = []
```

A list `L2` consisting of the first five prime numbers is created as follows:

```
L2 = [2, 3, 5, 7, 11]
```

A list `cls` containing ten students in a class is created via:

```
cls = ['John', 'Pam', 'Emily', 'Jessie', 'Brian', \
'Sam', 'Jim', 'Tom', 'Jerry', 'Alex']
```

Let's say that after some time, Emily moves to a different city:

```
del cls[2]
print cls
```

Output:

```
['John', 'Pam', 'Jessie', 'Brian', 'Sam', 'Jim',
'Tom', 'Jerry', 'Alex']
```

After some time, a new student Jack moves in:

```
cls.append('Jack')
print cls
```

Output:

```
['John', 'Pam', 'Jessie', 'Brian', 'Sam', 'Jim',
'Tom', 'Jerry', 'Alex', 'Jack']
```

The function `pop()` deletes an item and returns it for further use (as opposed to `del` which just deletes the item):

```
name = cls.pop(2)
print name
print cls
```

Output:

```
Jessie
['John', 'Pam', 'Brian', 'Sam', 'Jim',
'Tom', 'Jerry', 'Alex', 'Jack']
```

New item can be inserted at an arbitrary position using the function `insert()`:

```
cls.insert(3, 'Daniel')
print cls
```

Output:

```
['John', 'Pam', 'Brian', 'Daniel', 'Sam', 'Jim',
'Tom', 'Jerry', 'Alex', 'Jack']
```

A list can be sorted via the function `sort()`:

```
cls.sort()
print cls
```

Output:

```
['Alex', 'Brian', 'Daniel', 'Jack',
'Jerry', 'Jim', 'John', 'Pam', 'Sam', 'Tom']
```

The function `reverse()` reverses a list:

```
cls.reverse()
print cls
```

Output:

```
['Tom', 'Sam', 'Pam', 'John', 'Jim',
'Jerry', 'Jack', 'Daniel', 'Brian', 'Alex']
```

The function `count()` counts the number of occurences of an item in the list:

```
cls.count('Jerry')
```

Output:

```
1
```

The function `index()` returns the index of the first occurence of an item:

```
cls.index('Jerry')
```

Output:

```
5
```

If the item is not found, error is thrown:

```
cls.index('Kevin')
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 2, in <module>
ValueError: 'Kevin' is not in list
```

Another useful operation with lists is their *zipping*. The `zip()` function takes two lists as arguments and creates a new list of pairs. Example:

```
A = [1, 2, 3]
B = ['x', 'y', 'z']
print zip(A, B)
```

Output:

```
[(1, 'x'), (2, 'y'), (3, 'z')]
```

If the lists are not equally-long, the superfluous items in the longer list are skipped:

```
A = [1, 2, 3, 4, 5]
B = ['x', 'y', 'z']
print zip(A, B)
```

Output:

```
[(1, 'x'), (2, 'y'), (3, 'z')]
```

9.4 Dictionaries

Sometimes we want to store information for people. Such information may be their phone number, age, address, hobbies, family status and so on. Python's *dictionary* is a perfect tool to do this. The people's names are said to be *keys* and the additional information are the corresponding *values*.

Cars in a used car store can serve as another example: For each car, the key can be its VIN (or something else that is unique to each car). The corresponding value can be a list containing the price of the car, age, mileage, gas consumption, etc.

An empty dictionary D is defined as follows (note the curly brackets):

```
D = {}
```

An example of a phone book could be:

```
phonebook = {'Peter Parson': 8806336, 'Emily Everett': 6784346, \
'Lewis Lame': 1122345}
```

Once a new phone book was created, we may want to add a new person to it:

```
phonebook['Silly Sam'] = 1234567
print phonebook
```

Output:

```
{'Peter Parson': 8806336, 'Emily Everett': 6784346,
'Lewis Lame': 1122345, 'Silly Sam': 1234567}
```

We can also delete a person from the phonebook:

```
del phonebook['Peter Parson']
print phonebook
```

Output:

```
{'Emily Everett': 6784346, 'Lewis Lame': 1122345,
'Silly Sam': 1234567}
```

We can check if a key is in the dictionary:

```
if phonebook.has_key('Silly Sam'):
    print "Silly Sam's phone number is", phonebook['Silly Sam']
else:
    print "Silly Sam is not in the phonebook."
```

Output:

```
Silly Sam's phone number is 1234567
```

To get the list of all keys, we type:

```
phonebook.keys()
```

Output:

```
['Emily Everett', 'Lewis Lame', 'Silly Sam']
```

Similarly, we can also get the list of all values:

```
phonebook.values()
```

Output:

```
[6784346, 1122345, 1234567]
```

It is important to realize that **dictionaries are not ordered** in any special way. While a list can be sorted, a dictionary can not. Dictionaries are designed to store keys and values, and to be able to get the value for a given key quickly. The length of a dictionary can be obtained using the function `len()`:

```
len(phonebook)
```

Output:

```
3
```

There are additional functions that one can use with dictionaries. However, what we mentioned so far is enough for an introductory course and we refer to the Python tutorial[1] for more details.

10 More on Counting Loop

10.1 Objectives

– Review the `range()` function.
– Understand when the Python `for` loop should be used.
– Learn that the `for` loop can go over any list, tuple or dictionary.

10.2 The `range()` function revisited

The `range()` function was first mentioned in Paragraph 6.5. After learning about tuples, lists and dictionaries in Section 9, the reader will not be surprized when we state that this function produces a Python list of integers. For example:

```
range(10)
```

produces the following output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

It is possible to start the list from a nonzero integer:

[1] http://docs.python.org/tutorial

```
range(3, 10)
```

Output:

```
[3, 4, 5, 6, 7, 8, 9]
```

The `for` loop in Python is a *counting loop* analogous to the `repeat` loop in Karel the Robot. Thus it should be used when we know how many repetitions there will be. However, in Python, the counting loop is not restricted to integer numbers only. It is much more general – *it goes over the items of aa arbitrary list or tuple*.

10.3 The `for` loop combined with the `range()` function

When used in combination with the `range()` function, it goes over integer indices. Example:

```
for i in range(5, 10):
    print i
```

Output:

```
5
6
7
8
9
```

As usual, note indentation of the body of the `for` loop, which has the same purpose as indentation in the `while` loop – tell the interpreter which commands should be repeated.

10.4 The `for` loop combined with a list or tuple

Remember our tuple of months? We can use the `for` loop to print them:

```
for m in months:
    print m
```

Output:

```
January
February
March
April
May
June
July
August
September
October
November
December
```

The `for` loop works in the same way with lists – it does not distinguish between these two data types at all.

10.5 The `for` loop combined with a dictionary

Remember that it is easy to extract keys and values from a dictionary using the `keys()` and `values()` functions? In either case, the result of this operation is a *Python list*, which in turn can be used with the `for` loop. That was easy. Let us do something more interesting – perhaps convert an English-Spanish dictionary into a Spanish-English one:

```
ES = {'city': 'ciudad', 'fire': 'fuego', 'sun': 'sol'}
SE = {}
for english_word in ES:
    spanish_word = ES[english_word]
    SE[spanish_word] = english_word

print "Spanish-English dictionary:"
print SE
```

The output of this program is:

```
Spanish-English dictionary:
{'ciudad': 'city', 'sol': 'sun', 'fuego': 'fire'}
```

That was easy, wasn't it?

11 Exceptions

11.1 Objectives

– Understand what exceptions are, when they can occur, and how to catch them.
– Learn to never underestimate the user.
– Understand `ZeroDivisionError` and other exceptions.
– Learn to use the `assert()` function and catch the `AssertionError` exception.
– Learn how to raise exceptions using the `raise` statement.

Imagine that you have to write a Python function to evaluate

$$f(x) = \frac{1}{9 - x^2}.$$

Of course, this is simple!

```
def f(x):
    return 1. / (9 - x**2)
```

But, we can almost guarantee that the first user will call the function with $x = 3$ or $x = -3$. What will happen?

```
Traceback (most recent call last):
  File "<nclab>", line 1, in <module>
ZeroDivisionError: float division by zero
```

We can prevent the code from crashing in the function `f()` by returning `nan` ("not a number") that can be imported from Numpy:

```
from numpy import nan

def f(x):
    if x != -3 and x != 3:
        return 1. / (9 - x**2)
    else:
        print "Warning: Division by zero detected."
        return nan
```

However, this is not a perfect solution either. When called with `3` or `-3`, our function will not crash, but a program that uses it will crash some place else.

Situations like this (dividing by zero, taking square root of a negative number etc.) cause *runtime exceptions*. An exeption, when not caught and handled, will terminate our program with an error. However, Python offers an elegant way to catch exceptions, that we will explain in the next paragraph.

11.2 Exceptions handling

In contrast to *predicting* what possibily could go wrong and including conditions to detect wrong input, in exceptions handling we let it simply happen. Does it sound careless? Let us return to the original simple form of the function `f(x)`:

```
def f(x):
    return 1. / (9 - x**2)
```

This time, we will call it in such a way that an exceptional situation (here division by zero) is captured. For this we use the `try` command:

```
try:
    result = f(a)
except ZeroDivisionError:
    print "Division by zero detected, performing emergency \
    scenario!"
else:
    print "Evaluation of f(a) was successful, going on \
    normally."
```

This allows us to perform any emergency scenario *outside of the function* `f(x)`, which would not be possible otherwise.

Before continuing, we recommend that you type this code into the code cell, including the definition of the function `f(x)`, and run it with values that crash the function `f(x)` and with values that do not. If you are a newcomer to exceptions handling, this is an interesting experience. Feels a bit like letting the airplane crash, but save all passengers afterwards.

11.3 The `assert()` function

The `assert()` function raises an exception whenever the Boolean statement used as its argument is `False`. This is a powerful tool to raise our own exceptions. These exceptions can be caught using the `try` command as all other exceptions:

```
def f(x):
    assert(x != 3 and x != -3)
    return 1. / (9 - x**2)

try:
    result = f(a)
except AssertionError:
    print "Assertion failed, performing emergency scenario!"
else:
    print "Assertion passed, going on normally."
```

When called with `a` equal to `1`, this code gives:

```
Assertion passed, going on normally.
```

When called with `a` equal to `3`, this is what happens:

```
Assertion failed, performing emergency scenario!
```

11.4 The `raise` statement

Python does not have a command to stop program execution. Sometimes this is useful though, such as when the user calls our function with incorrect data. To illustrate this, let us return to the function `f()` that was defined above, but now we include a `raise` command in it:

```
def f(x):
    if x == 3 or x == -3:
        raise ValueError("Division by zero detected.")
    return 1. / (9 - x**2)
```

When the function `f(x)` is called with `3` or `-3`, the output will be

```
Traceback (most recent call last):
  File "<string>", line 7, in <module>
  File "<nclab>", line 3, in f
ValueError: Division by zero detected.
```

Incorrect values of user input are the most common reason to raise exceptions on our own. So far we have seen `ZeroDivisionError`, `AssertionError` and `ValueError`. There are additional types of exceptions in Python – let us mention the most important ones in the next paragraph.

11.5 Other types of exceptions

There are many different exceptions including

- `ValueError` (wrong value encountered in a function or expression),
- `MemoryError` (program runs out of memory),
- `Name Error` (local or global name is not found),
- `OverflowError` (result of an arithmetic operation is too large to be represented),
- `SyntaxError` (parser encounters a syntax error),
- `IndentationError` (parser finds incorrect indentation),
- `UnboundLocalError` (reference is made to a local variable in a function or method, but no value has been bound to that variable),

and others. All of them can be caught using the `try` command.

Let us experiment with `MemoryError` for a moment. The following code will try to allocate in the memory a 2D array of real numbers of size $n \times n$ with `n = 1000000`:

```
from numpy import zeros

def array(n):
    a = zeros((n, n))
    return a

A = array(1000000)
```

The size of this array would be around 8 TB (one TeraByte is 1024 GigaBytes), which is more runtime memory than all NCLab currently has. Since no exceptions handling is present, Python will throw a `MemoryError` exception and the code will simply crash:

```
Traceback (most recent call last):
  File "<string>", line 6, in <module>
  File "<nclab>", line 4, in array
MemoryError
```

Can we implement such a program in a smarter way? Of course! The following implementation catches the `MemoryError` exception if it happens, and tells the user that so much memory is not available:

```
from numpy import zeros

def array(n):
    a = zeros((n, n))
    return a

try:
    A = array(1000000)
except:
    print "There was not enough memory for array A!"
    print "Please reduce array size and retry."
else:
    print "Array A was created successfully."
```

Now the output is:

```
There was not enough memory for array A!
Please reduce array size and retry.
```

If you are interested in learning more about exceptions, visit http://docs.python. org/ library/exceptions. html. Before we move to the next section, let's just remember:

Never underestimate the user. Whenever your program accepts user input, make sure that you check its sanity.

12 Object-Oriented Programming

12.1 Objectives

– Understand the philosophy of object-oriented (OO) programming.
– Understand the difference between OO and procedural programming.
– Learn basic concepts – classes, objects, and methods.
– Learn the syntax of defining and instantiating classes in Python.

Since the very beginning of this course, we have put emphasis on designing simple algorithms and writing clean, transparent code. This included:

– Creating custom functions for functionality that can be reused.
– Keeping all variables as local as possible

Object-oriented programming brings these good programming practices to perfection: It will help us to write crystal clear, elegant, and very safe code.

12.2 From procedural to object-oriented programming

The programming style we have been using so far is called *procedural programming*. The name, of course, comes from using *procedures* to solve various tasks that lead to the solution of the problem at hand. Procedure is an activity, a sequence of operations done with data that is supplied to it. *Procedures do not own the data they operate with.*

This, however, means that the data must be stored somewhere else. In other words, the range of validity of the data extends beyong the borders of the function where it is processed. Can you see an analogy to using local and global variables? It is great to keep variables local, and in the same way it would be great to make data local to procedures that operate with them. But wait, this is called *object-oriented programming!*

12.3 Example: Moving to a different city

Here is a real-life example demonstrating the difference between procedural and object-oriented programming. Imagine that you are moving and all your things need to be packed, boxed, loaded on a truck, hauled a 1000 miles, unloaded, and brought into your new home.

Procedural approach: You (the procedure) go rent a truck – this is the data that the procedure does not own. Then you pack everything, load it on the truck, drive to the other city, unload your things, and carry them into your new home. Then you go return the truck (that's the return statement at the end of the procedure).

Object-oriented approach: You call a moving company "C". This means that you just created an object C. Note that the object C owns the truck. They will come to your home and use methods C.pack(), C.box(), C.load(), C.haul(), C.unload(), C.carry() and perhaps even C.unpack() to solve your problem. Notice the following benefits of the object-oriented approach:

– The work is being done by the company, you do not have to worry (delegation of work).
– You do not have to be able to drive a truck (better code structure).
– You do not need to know how the company does the moving (reduced flow of data).
– You can concentrate on solving problems in your area of expertise (greater efficiency).
– You do not risk crashing the truck (safer programming).

12.4 Classes, objects, and methods

Let us stay with the moving example for a little while longer. The moving company that you heard of, or that you saw in an advertisement, is a *class* or in other words a *concept*. Formally, we say that *class* is an entity comprising data and functionality that uses the data. But a concept will not get your things moved. For that you need a concrete team of movers. This is called an *object* or an *instance of the class*.

When the movers appear at your doorstep, that's when an instance of the class is created. When they pack, box, and load your things, the object is using *methods* of the class. Technically, methods belong to the class because that's where they are defined, but often we say that "an objects uses its methods to ...". Once again, methods are *defined in a class* but *used by instances of the class*. Methods of a class can operate on data owned by the class (the truck) as well as on data that does not belong to the class (your things that are being moved.

Similarly with variables – variables are defined in a class, but they do not have concrete values there. In the class definition, they are part of a concept. Only after an instance of the class is created, the variables are filled with concrete values.

12.5 An example from geometry

Let's see how all this is implemented in Python. We will say goodbye to the movers and play with some simple geometrical concepts instead. Let us create a class `circle`. The data owned by this class will be:

- Radius `R`,
- center with coordinates `Cx, Cy`,
- two arrays `ptsx`, `ptsy` to plot the circle.

The methods of this class will include

- `__init__(self, r, cx, cy)` ... constructor,
- `area(self)` ... calculate and return its area,
- `perimeter(self)` ... calculate and return its perimeter,
- `draw(self)` ... draw itself.

Do not be disturbed by the funny way the constructor `__init__` is defined in Python, or by the argument `self` that is present in all methods. Constructors are used in all object-oriented languages to pass external parameters (such as the radius and the center point in this case) into newly created instances. In other words, each instance of the class `circle` can have a different radius and a different center.

The keyword `self` must be included in each method of the class and this is a reference to the concrete instance through which all data and methods of the instance are accessed. Such a reference is present in other object-oriented languages as well - including C++ where the programmer does not have to mention it explicitly like in Python. It is added there by the compiler.

12.6 Defining and using class `circle`

This is how the class `circle` is defined in Python:

```
from numpy import pi, cos, sin
from pylab import clf, plot, legend, axis

class circle:
    # Constructor:
    def __init__(self, r, cx, cy, n = 100):
        self.R = r
        self.Cx = cx
        self.Cy = cy
        self.n = n
        # And now the points for plotting:
        self.ptsx = []
        self.ptsy = []
        da = 2*pi/self.n
        for i in range(n):
            self.ptsx.append(self.Cx + self.R * cos(i * da))
            self.ptsy.append(self.Cy + self.R * sin(i * da))
        self.ptsx.append(self.Cx + self.R)
        self.ptsy.append(self.Cy + 0)

    # Method to return circle area:
    def area(self):
        return pi * self.R**2

    # Method to return perimeter:
    def perimeter(self):
        return 2 * pi * self.R
    # Method to plot the circle:
    def draw(self, label):
        axis('equal')
        plot(self.ptsx, self.ptsy, label = label)
        legend()
```

Notice how the variables owned by the class are always used with the prefix `self`. With such a nice class in hand, it is easy to draw various circles, and calculate their areas and perimeters:

```
# Create an instance:
C1 = circle(1, 0, 0)
print "Are and perimeter of circle 1:", \
C1.area(), C1.perimeter()
C2 = circle(0.5, 1, 0)
print "Are and perimeter of circle 2:", \
C2.area(), C2.perimeter()
C3 = circle(0.25, 1.5, 0)
print "Are and perimeter of circle 3:", \
C3.area(), C3.perimeter()
```

```
clf()
C1.draw("First circle")
C2.draw("Second circle")
C3.draw("Third circle")
show()
```

The output (text and Fig. 27) is shown below.

```
Area and perimeter of circle 1: 3.14159265359 6.28318530718
Area and perimeter of circle 2: 0.785398163397 3.14159265359
Area and perimeter of circle 3: 0.196349540849 1.57079632679
```

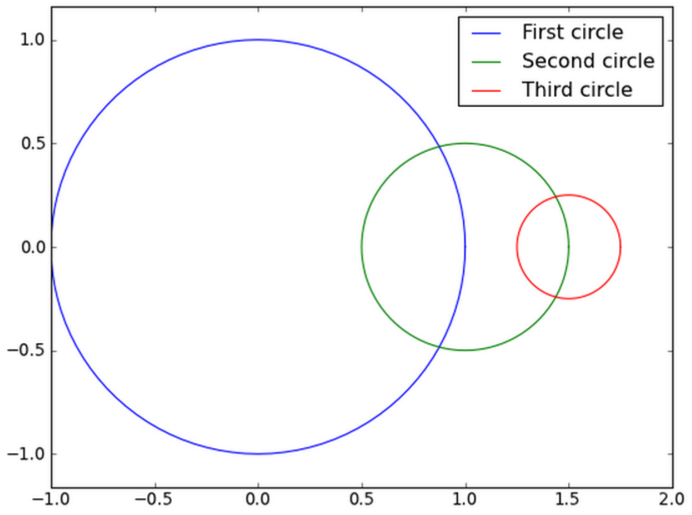


Fig. 27: Three instances of the class `circle`.

12.7 Text string as a class

We first met text strings in Section 8. Now that we know how classes are defined and used, we can reveal that text string is a class! In fact it has many great methods that really make working with texts easy and fun. Let us show some of them. Below, arguments enclosed in brackets are optional, and if they are used, then without the brackets:

Method `capitalize()`

Return a copy of the string with only its first character capitalized.

Example:

```
str = "sentences should start with a capital letter."
print str.capitalize()
```

Output:

```
Sentences should start with a capital letter.
```

Method `count(sub[, start[, end]])`

Return the number of occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Example:

```
str = "John Stevenson, John Lennon, and John Wayne."
print str.count("John")
```

Output:

```
3
```

Method `find(sub[, start[, end]])`

Return the lowest index in the string where substring `sub` is found, such that sub is contained in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Return `-1` if `sub` is not found.

Example:

```
str = "This happened during the summer."
print str.find("sum")
```

Output:

```
25
```

Method `index(sub[, start[, end]])`

Like find(), but raise ValueError when the substring is not found.

Method `isalnum()`

Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

Example:

```
str1 = "mypassword123"
if str1.isalnum():
    print str1, "is alphanumeric."
else:
    print str1, "is not alphanumeric."
str2 = "mypassword123+"
if str2.isalnum():
    print str2, "is alphanumeric."
else:
    print str2, "is not alphanumeric."
```

Output:

```
mypassword123 is alphanumeric.
mypassword123+ is not alphanumeric.
```

Method `isalpha()`

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

Example:

```
str1 = "John"
if str1.isalpha():
    print str1, "is alphabetic."
else:
    print str1, "is not alphabetic."
str2 = "My name is John"
if str2.isalpha():
    print str2, "is alphabetic."
else:
    print str2, "is not alphabetic."
```

Output:

```
John is alphabetic.
My name is John is not alphabetic.
```

Method `isdigit()`

Return true if all characters in the string are digits and there is at least one character, false otherwise.

Example:

```
str1 = "2012"
if str1.isdigit():
    print str1, "is a number."
else:
    print str1, "is not a number."
str2 = "Year 2012"
if str2.isdigit():
    print str2, "is a number."
else:
    print str2, "is not a number."
```

Output:

```
2012 is a number.
Year 2012 is a number.
```

Method `join(seq)`

Return a string which is the concatenation of the strings in the sequence seq. The base string (object whose method is used as separator).

Example:

```
str = "..."
str1 = "This"
str2 = "is"
str3 = "the"
str4 = "movie."
print str.join([str1, str2, str3, str4])
```

Output:

```
This...is...the...movie.
```

Method `lower()`

Return a copy of the string converted to lowercase.

Example:

```
str = "She lives in New Orleans."
print str.lower()
```

Output:

```
she lives in new orleans.
```

Method `replace(old, new[, count])`

Return a copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

Example:

```
str = "First word, second word, and third word."
print str.replace("word", "thing")
```

Output:

```
First thing, second thing, and third thing.
```

Method `swapcase()`

Return a copy of the string with uppercase characters converted to lowercase and vice versa.

Example:

```
str = "The abbreviation is MENSA"
print str.swapcase()
```

Output:

```
tHE ABBREVIATION IS mensa
```

Method `title()`

Return a titlecased version of the string: Words start with uppercase characters, all remaining cased characters are lowercase.

Example:

```
str = "this is the title of my new book"
print str.title()
```

Output:

```
This Is The Title Of My New Book
```

Method `upper()`

Return a copy of the string converted to uppercase.

Example:

```
str = "this is the title of my new book"
print str.upper()
```

Output:

```
THIS IS THE TITLE OF MY NEW BOOK
```

For additional useful text string methods see the Python documentation at `http://docs.python.org/release/2.5.2/lib/string-methods.html`.

Before we leave this section, notice that lists, tuples, dictionaries and many other data structures in Python are classes as well.

13 Class Inheritance

13.1 Objectives

– Understand the benefits of inheritance in object-oriented programming.
– Learn the syntax of creating descendant classes in Python.
– Learn how methods of parent class are called from the descendant.

To illustrate the inheritance process in object-oriented programming, we will build a small library of geometrical objects that contains triangles, quadrilaterals, rectangles, squares, and circles. We want all these classes to have the same functionality as the class `circle` from Paragraph 12.6. This means, they should be able to:

– Calculate their area,
– calculate their perimeter,
– draw their geometry.

Notice one interesting thing: The drawing functionality will be the same for all these classes. Specifically, all of them will have the coordinate arrays `ptsx`, `ptsy` and the plotting method `draw()`. Other than that their functionality will differ. This is where *class inheritance* comes handy.

13.2 Creating base class `geometry`

Instead of repeating the same drawing functionality in all classes, we create a base class `geometry` as follows:

```
from pylab import clf, plot, legend, axis

# Base class:
class geometry:
    # Constructor:
    def __init__(self):
        self.ptsx = []
        self.ptsy = []

    # Plotting method:
    def draw(self, label):
        axis('equal')
        plot(self.ptsx, self.ptsy, label = label)
        legend()
```

This base class really cannot do much, but this is fine. We will use it to derive several new (descendant) classes. The benefit of inheritance is that if we decide in the future to change the plotting mechanism, we will just have to change it at one place – in the base class `geometry` – and the change will propagate automatically into all its descendants. Remember that humans are extremely prone to mistakes. Having the same code at several different places in the program is about the worst thing a programmer can ever do.

13.3 Deriving `polygon` from `geometry`

There is more to the story. Triangles, quadrilaterals, rectangles and squares are all *polygons*. Hence let us create the class `polygon` to be a descendant of the base class `geometry`. This class will represent general non-convex counter-clockwise (CCW) oriented polygons. Notice the syntax of the inheritance: one types `class polygon(geometry):` instead of just `class polygon`:

```python
# Class polygon derived from class geometry,
# represents general, possibly non-convex
# polygon. Boundary needs to be CCW oriented.
class polygon(geometry):
    # Constructor (here L is list of vertices):
    def __init__(self, L):
        self.ptsx = []
        self.ptsy = []
        # Convert L into plotting arrays:
        for pt in L:
            self.ptsx.append(pt[0])
            self.ptsy.append(pt[1])
        # To close the loop in plotting:
        self.ptsx.append(L[0][0])
        self.ptsy.append(L[0][1])
    # Method to calculate area of a general
    # oriented polygon:
    def area(self):
        # Calculate minimum y-coordinate:
        ymin = min(self.ptsy)
        # The area of an oriented polygon is the sum of
        # areas of oriented trapezoids under its edges:
        m = len(self.ptsx) - 1    # Number of trapezoids.
        s = 0
        for i in range(m):
            s += (self.ptsx[i+1] - self.ptsx[i]) \
                 * (self.ptsy[i+1] + self.ptsy[i] \
                 - 2*ymin) / 2.
        return -s

    # Method to calculate perimeter of a general
    # oriented polygon:
    def perimeter(self):
        l = 0
        m = len(self.ptsx) - 1
        for i in range(m):
            l += sqrt((self.ptsx[i+1] - self.ptsx[i])**2
                 + (self.ptsy[i+1] - self.ptsy[i])**2)
        return l
```

Notice that we did not have to define the method `draw()` here.

13.4 Deriving `circle` from `geometry`

Next let us derive the class `circle` from the class `geometry`. Obviously it cannot be derived from `polygon` because circles are not polygonal geometries:

```
# Class circle derived from class geometry:
class circle(geometry):
    # Constructor (here r is radius, (cx, cy) the center point,
    # and n plotting subdivision):
    def __init__(self, r, cx, cy, n = 100):
        self.R = r
        self.Cx = cx
        self.Cy = cy
        self.n = n
        # Create array of points for plotting:
        self.ptsx = []
        self.ptsy = []
        da = 2*pi/self.n
        for i in range(n):
            self.ptsx.append(self.Cx + self.R * cos(i * da))
            self.ptsy.append(self.Cy + self.R * sin(i * da))
        self.ptsx.append(self.Cx + self.R)
        self.ptsy.append(self.Cy + 0)

    # Method to calculate circle area:
    def area(self):
        return pi * self.R**2

    # Method to calculate perimeter:
    def perimeter(self):
        return 2 * pi * self.R
```

Again, notice that we did not have to define the method `draw()`.

13.5 Deriving `triangle` and `quad` from `polygon`

The `polygon` class defined above is very general. Of course we could use it directly to render triangles, quads, rectangles, and squares. But these geometries are simpler and thus we also want their instantiation to be simpler. Let us begin with the class `triangle`.

This class is very simple – it takes three points as parameters, forms a list containing these three points, and then the list is passed into the constructor of the parent class `polygon`. Notice the way the constructor of the `polygon` class is called from its descendant `triangle` via `polygon.__init__(self, L)`:

```
# Class triangle derived from class polygon:
class triangle(polygon):
    # Constructor (here a, b, c are the vertices):
    def __init__(self, a, b, c):
        L = [a, b, c]
        polygon.__init__(self, L)
```

Class `quad` is created analogously:

```
# Class quad derived from class polygon:
class quad(polygon):
    # Constructor (here a, b, c, d are the vertices
    # in CCW orientation):
    def __init__(self, a, b, c, d):
        L = [a, b, c, d]
        polygon.__init__(self, L)
```

13.6 Deriving `rectangle` from `quad`

Next, rectangle is a special case of quad, so let us derive the `rectangle` class from the `quad` class:

```
# Class rectangle derived from class quad.
# Represents rectangle (0, a) x (0, b):
class rectangle(quad):
    # Constructor:
    def __init__(self, a, b):
        quad.__init__(self, [0, 0], [a, 0], [a, b], [0, b])
```

13.7 Deriving `square` from `rectangle`

Taking this one step further, square is a special case of rectangle. Therefore we can derive the `square` class from the `rectangle` class:

```
# Class square derived from class rectangle.
# Represents square (0, a) x (0, a):
class square(rectangle):
    # Constructor. Here a is the edge length:
    def __init__(self, a):
        rectangle.__init__(self, a, a)
```

13.8 Diagram of class structure

The inheritance structure of our classes is depicted in Fig. 28.

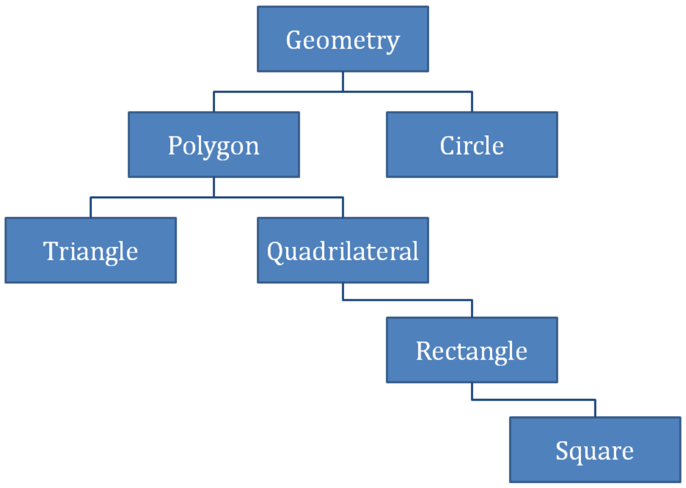


Fig. 28: Graphical representation of the structure of our classes.

13.9 Creating sample instances

Finally let us create sample instances of the classes defined above, inquire about their areas and perimeters, and let them plot themselves:

```
# Create a triangle:
T = triangle([-2, -0.5], [0, -0.5], [-1, 2])
print "Area and perimeter of the triangle:", \
T.area(), T.perimeter()

# Create a quad:
Q = quad([-3, -1], [0, -1], [-1, 1], [-2, 1])
print "Area and perimeter of the quad:", \
Q.area(), Q.perimeter()

# Create a rectangle:
R = rectangle(3, 1)
print "Area and perimeter of the rectangle:", \
R.area(), R.perimeter()

# Create a square:
S = square(1.5)
print "Area and perimeter of the square:", \
S.area(), S.perimeter()

# Create a circle:
C = circle(2.5, 0, 0.5)
print "Area and perimeter of the circle:", \
C.area(), C.perimeter()

# Plot the geometries:
clf()
T.draw("Triangle")
Q.draw("Quad")
R.draw("Rectangle")
S.draw("Square")
C.draw("Circle")
ylim(-3, 4)
legend()
show()
```

First, here is the textual output:

```
Area and perimeter of the triangle: 2.5 7.38516480713
Area and perimeter of the quad: 4.0 8.472135955
Area and perimeter of the rectangle: 3.0 8.0
Area and perimeter of the square: 2.25 6.0
Area and perimeter of the circle: 19.6349540849 15.7079632679
```

The graphical output is shown in Fig. 29.

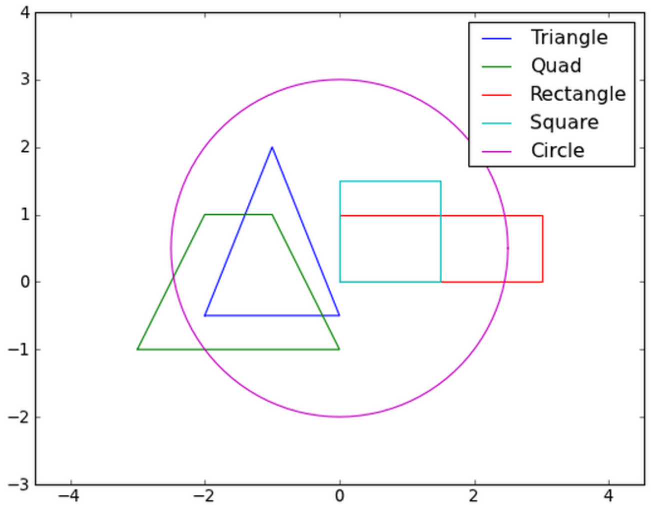


Fig. 29: Visualization of the instances created above.

Bézier curves

:
Bézier curves are the most widely used approach to define curves in engineering design. Let us begin with the linear case. Linear Bézier curve is a straight line that connects two control points P_1 and P_2. All Bézier curves are parameterized from the interval $[0, 1]$, so in the linear case the exact mathematical definition is

$$B(t) = P_1 + t(P_2 - P_1)$$

You can easily check that the curve starts at P_1 (substitute $t = 0$ to the above formula), and that it ends at P_2 (substitute there $t = 1$).

Quadratic Bézier curves are defined using three control points P_1, P_2 and P_3. Again they are parameterized from $[0, 1]$, and their definition is

$$B(t) = (1-t)^2 P_1 + 2(1-t)tP_2 + t^2 P_3.$$

Cubic Bézier curves are defined using four control points P_1, P_2, P_3 and P_4. They are parameterized from $[0, 1]$ as usual, and their definition is

$$B(t) = (1-t)^3 P_1 + 3(1-t)^2 t P_2 + 3(1-t)t^2 P_3 + t^3 P_4.$$

14 Python Libraries

14.1 Objectives

- Understand that using libraries enhances your problem solving skills.
- Learn basic facts about Scipy, Numpy, Pylab, Matplotlib, and Sympy.
- Learn where you can obtain more information about these libraries.
- Learn that Internet is the best resource for Python programmers.

Using libraries is an indivisible part of Python programming. As opposed to commercial products, Python libraries are free. We have used some of them already, such as Pylab and Matplotlib in Section 4 and Numpy in Section 7.

14.2 Python Standard Library

The Python Standard Library has vast functionality related to built-in functions, built-in types, built-in constants, built-in exceptions, string services, numerics and mathematics modules, files, data compression, cryptographic services, and many others. Just introducing a table of contents would take several pages. To learn more, to visit `http://docs.python.org/library/`. Also, a long list of packages for Python programmers can be found at `http://pypi.python. org/pypi/`.

14.3 Pylab and Matplotlib

The Pylab library can be used for numerical computation but as there are other and stronger tools, we recommend to use it mainly for plotting, along with the Matplotlib library. For more details on Pylab visit `http://www.scipy.org/PyLab`. This page is not dedicated to plotting though, more on concrete plotting functionality can be found on the home page of Matplotlib, `http://matplotlib.sourceforge.net`. This page contains an exhaustive overview of plotting-related functionality, many examples, gallery, and documentation.

14.4 Scipy, Numpy and Sympy

Scipy is a general library for scientific computing with Python. Numpy can be viewed as its "heart" containing numerical algorithms and methods. These methods comprise interpolation, integration, optimization, Fourier transforms, signal processing, linear algebra, eigenvalue problems, graph theory routines, statistics, ordinary and partial differential equations, image processing, etc. To learn more, visit `http://www.scipy.org` and `http://numpy.scipy.org/`.

Sympy is an excellent library for symbolic mathematics covering high-school algebra, calculus, matrix algebra and differential equations. To learn more about Sympy, visit its home page (`http://www.sympy.org`).

15 Other Recommended Topics in Python

There are many topics that we have decided to skip in the first reading of the Python language. There is more to learn to almost every command, function aned concept described in this document. If you like Python, your nexr resource should be the original Python tutorial at `http://docs.python.org`.

16 What Next?

Congratulations, you made it! We hope that you enjoyed the textbook and the exercises. If you can think of any way to improve the application Karel the Robot or this tutorial, we would be very happy to hear from you. If you have an interesting new game or exercises for Karel, please let us know as well.

Although you may feel like an Almighty Programmer right now, we would recommend staying humble. Even the most experienced programmers are learning new things all the time. There is much more to Python that we managed to cover in this introductory textbook. You already know about the Internet resources where you can learn more.

Alternatively, you may dive into a next programming language! We would recommend Javascript since this is the most popular language for web development. Of course there are many more languages to explore, including C/C++, Java, Perl, Ruby, Lua and others.

In any case, our team wishes you good luck, and keep us in your favorite bookmarks!

Your Authors

Part II

Programming Exercises

1 Introduction

All exercises from this textbook can be cloned in NCLab through the *Project → Clone* menu. Solution Manual is available in NCLab starting with Basic Version. To warm up, in this section we will experiment with the `print` command.

1.1 Python is fun

Write a program that prints the phrase "Python is fun"

1. As separate words on three lines, one word per line, aligned to the left.
2. On one line.
3. On one line, inside a box made up of the characters '=' and '|'.

1.2 Big word

Write a program that prints the word "Python" in large block letters as shown below:

```
XXXXX Y   X XXXXX X   X XXXXX X   X
X   X Y   X   X   X   X X   X XX  X
XXXXX XXXXX   X   XXXXX X   X X X X
X         X   X   X   X X   X X  XX
X     XXXXX   X   X   X XXXXX X   X
```

1.3 Military exercise

Save the program from the previous exercise under the name "George Smith" and adjust it to print the name of a famous American general who during the World War 2 commanded U.S. Army Troops in North Africa and Europe.

2 Using Python as a Calculator

In this section we will practice using Python's interactive shell for simple as well as advanced arithmetic operations.

2.1 Grocery shopping

Today you went grocery shopping and you bought:

1. Two cans of an energy drink for \$1.56 a piece.
2. Three bottles of milk \$2.34 a piece.
3. Four French baguettes \$3.41 a piece.
4. Five packs of chewing gum \$0.99 a piece.

Tax in the amount of 8% applies to items 1 and 4. Calculate the total cost of your purchase!

2.2 Oil change

You decided to do oil change today since your favorite place offers 20% off. The oil costs \$45 plus 8% tax. The work is 25 dollars. Do not forget to take the 20% off the total. How much will you pay for this oil change?

2.3 Age average

You go on a hike with a group of people of different ages. Your are (say) 15 years old. Two other people are 21 years old, one is 35, one is 42, and one is 55. Calculate the average age in the group!

2.4 Saving for a bike

Your savings account grows at a rate of 3% annually. Three years ago you started by inserting \$1,250 into the account. One year later you added another \$500, and a year ago you withdrew \$300. How much money is in your savings account now?

2.5 Growing town

For the last 20 years, the population of a city has been growing steadily by 5% per year, and 20 years ago it had 10,000 inhabitants. How many people live in the city now? Round the result to be an integer number.

2.6 Coffee maker

How many cups of coffee will you be able to make from a one kilogram pack when 11 grams are needed for one cup? How many grams will be left?

2.7 Gardening

You are buying plants for an orchard restoration project. How many plants can you purchase with \$250 if the price of one is \$17? How much money will you have left?

2.8 Setting tiles

You are setting tiles in a room of rectangular shape whose edges measure 19 and 27 feet. The tiles are 1.25×1.25 foot squares and they will be aligned with the walls in the simplest possible pattern. You need to leave a quarter inch between the tiles for grout. How many whole tiles are you going to use? Hint: integer part of a real number `x` is obtained via `int(x)`.

2.9 Submerged soccer ball

The famous Archimedes' law states that the upward buoyant force exerted on a body immersed in a fluid is equal to the weight of the fluid the body displaces. Imagine that you submerge a soccer ball – neglecting its own weight, this is the force that you need to employ to keep it under water! Let's calculate this force, assuming that the ball is a perfect sphere of diameter 22 cm. The density of water is 1000 kg/m^3 and gravitational acceleration is 9.81 ms^{-2}.

2.10 Fastest runner

Arguably, the fastest human runner in history achieved a speed of 12.1 m/s. Calculate how much time in seconds such a runner would need to cross a football field at this speed, running diagonally from one corner to the opposite one. The measures of a football field are 109.7 m and 48.8 m.

2.11 Triangle area

Heron's formula is a famous formula that allows you to calculate the area S of a general triangle knowing its edge lengths a, b and c:

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

where $p = (a+b+c)/2$. Calculate in this way the area of a triangle with sides 2.5 m, 3.7 m and 4.1 m!

2.12 Math functions

Calculate the following values:

$$a = e^5,$$
$$b = \sin(\pi/3),$$
$$c = \cos(\pi/4),$$
$$d = \log(10).$$

Here log is the natural logarithm.

2.13 Random numbers

Random numbers are generated using the function `random()` from the `random` library. This function returns a random real number between 0 and 1. Use this function to generate a random integer between 10 and 20!

2.14 Greatest common divisor

Python has a built-in function `gcd()` to calculate the greatest common divisor (GCD) of two integers. It is imported via `from fractions import gcd`. Use the function to find the GCD of 1554 and 2331!

2.15 Fractions

Python makes operations with fractions very simple via the `Fraction` function that is imported from the `fractions` library. Then a fraction `1/3` is defined simply as `Fraction(1, 3)` and so on. Fractions can be used with the same operations as numbers, and the result of such an operation is a `Fraction`. Now to your task: Use Fractions to calculate

$$\frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \frac{1}{5} + \frac{1}{6} - \frac{1}{7}.$$

3 Functions

In this section we will practise defining and using custom functions.

3.1 Circle

Write a Python function `circle(R)` that accept an arbitrary radius `R` as argument, and returns the area and the perimeter of a circle with radius `R`.

3.2 Square

Write a Python function `squarearea(a)` that returns the area of a square whose edge is `a` cm long. You know that most of the time the edge length wil be 1 cm, so make this function callable without any arguments.

3.3 Rectangle

Write a Python function `rectanglearea(a, b)` that returns the area of a rectangle whose edges are a and b cm long. You know that most of the time one of the edges will measure 3 cm, so make the function callable with only one argument.

3.4 Sales prediction

The East Coast sales division of a company usually generates P percent of total sales each year. Write a function `sales_prediction(P, S)` to predict how much the East Coast division will generate if the company has S dollars in sales the next year. Use your function with the numbers $P = 62\ \%$ and $S = 4.6$ million dollars.

3.5 Sales tax

Write a function `sales_tax(P, ST, CT)` that calculates the total sales tax on a P dollars purchase. Assume the state sales tax is ST percent, and the county sales tax is CT percent. Use the function with the following numbers: $P = 52$ dollars, $ST = 5$ %, $CT = 2$ %.

3.6 Restaurant bill

Write a function `restaurant_bill(M, P, T)` that computes the tax and tip on a restaurant bill for a patron with M dollars meal charge. The tax is P percent of the meal cost. The tip is T percent of the total after adding the tax. Your function should return the tax amount, tip amount, and the total bill. Use your function with the numbers $M = 44.50$ dollars, $P = 6.75$ % and $T = 15$ %.

3.7 Gas consumption conversion I (EU to US)

In Europe, gas consumption of a car is reported in liters per 100 kilometers. In the U.S. it is reported in miles per gallon. Write a function `conversion_eu_to_us(C)` that converts a given European gas consumption C into the U.S. scale. One mile is 1.609344 kilometers, and one gallon is 3.78541178 liters.

3.8 Gas consumption conversion II (US to EU)

In Europe, gas consumption of a car is reported in liters per 100 kilometers. In the U.S. it is reported in miles per gallon. Write a function `conversion_us_to_eu(C)` that converts a given U.S. gas consumption C into the European scale. One mile is 1.609344 kilometers, and one gallon is 3.78541178 liters.

3.9 Distance per tank of gas

A car with a G gallon gas tank averages A miles per gallon when driven in town and B miles per gallon when driven on the highway. Write a function `distance(G, A, B)` that returns the distance the car can travel on one tank of gas when driven in town and when driven on the highway. Use your function with $G = 20$ gallons, $A = 21.5$ miles per gallon, $B = 26.8$ miles per gallon.

3.10 Circuit board price

An electronics company sells circuit boards at a P percent profit. Write a function `circuit_ board_price(P, D)` that calculates the selling price of a circuit board that costs them D dollars to produce. Use your function with $P = 40$ % and $D = 12.67$ dollars.

3.11 Gross pay

A particular employee earns E dollars annually. Write a function `gross_pay(E)` that determines and prints what the amount of his gross pay will be for each pay period if he is paid twice a month (24 pay checks per year) and if he is paid bi-weekly (26 checks per year). Use your function with $E = 32,500$ dollars.

3.12 Stock gain

Kathryn bought N shares of stock at a price of A dollars per share. One year later she sold them for B dollars per share. Write a function `stock_gain(N, A, B)` that calculates and displays the following:

– The total amount paid for the stock.
– The total amount received from selling the stock.
– The total amount of money she gained.

Use your function with the values $N = 600$, $A = 21.77$ dollars and $B = 26.44$ dollars.

3.13 Temperature conversion I (Fahrenheit to Celsius)

Write a function `temperature_conversion_FC(F)` that converts temperature `F` from Fahrenheit to Celsius and returns the result. The corresponding formula is

$$C = \frac{5}{9}(F - 32).$$

3.14 Temperature conversion II (Celsius to Fahrenheit)

Write a function `temperature_conversion_CF(C)` that converts temperature `C` from Celsius to Fahrenheit and returns the result. The corresponding formula is

$$F = \frac{9}{5}C + 32.$$

3.15 Time in seconds

Write a function `time_in_seconds(H, M, S)` where H, M, S are integer numbers representing hours, minutes and seconds. Convert this time period into seconds, and return it.

3.16 Time in hours

Write a function `time_in_hours(H, M)` where H, M are integer numbers representing hours and minutes. Convert this time period into hours and return it. For example, `time_in_hours(1, 30)` will return `1.5`.

3.17 Time last time

Write a function `time_conversion(T)` that takes a period of time `T` in seconds (an integer number), and converts it to hours, minutes and seconds. It should return the corresponding three values H, M, S. The number H should be an integer, and both S and M should be integers between 0 and 59.

3.18 Distance

Write a function `distance(x1, y1, x2, y2)` that calculates and returns the distance d of two points (x_1, y_1) and (x_2, y_2) using the formula

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

3.19 Angle conversion

Write a function `angle_conversion(A)` that converts the angle A from degrees to radians and returns the result. Use the fact that π radians equals to 180 degrees. In order to work with `pi`, you need to import it from Numpy first. Hence, begin your program with the line "`from numpy import pi`".

3.20 Length conversion

Write a function `length_conversion(Y, F, I)` that converts a length given in terms of yards, feet and inches into meters. Return the result. Recall that one yard is 0.9144 meters, there are three feet in a yard, and 12 inches per foot.

3.21 Area conversion

Write a function `area_conversion(SF)` that converts an apartment area from square feet SF to square meters. Return the result.

3.22 Digital storage conversion

Write a function `memory_conversion(GB, MB, KB, B)` that converts a digital storage size consisting of GB gigabytes, MB megabytes, KB kilobytes, and B bytes to bytes. Return the result. Recall that one kilobyte equals to 1024 bytes, one megabyte equals to 1024 kilobytes, and one gigabyte is the same as 1024 megabytes.

3.23 Savings account

Write a Python function `savings(amount, p, num_years)` that calculates how much money will be in your savings account after `num_years` with an initial amount `amount`, if every year the amount increases by `p` percent.

4 Colors and plotting

In this section we will practice plotting of various objects including polygons, function graphs, 2D and 3D curves, surfaces, and even pie and bar charts.

4.1 Polygon

Write a Python function `polygon(R, n)` that returns two arrays `x` and `y`. These arrays will contain the x and y coordinates of the vertices of a polygon with n equally-long edges, that is inscribed into a circle of center $[0, 0]$ and radius $R > 0$. Here `n > 2` is an integer number and `R > 0` is a real number. The polygon should be oriented counter clock-wise and the first vertex should be $[R, 0]$. Use this function to plot a sample polygon with parameters `R = 2`, `n = 6`. Make sure that axes are scaled equally.

4.2 Custom plotting function

Consider an arbitrary Python function `f(x)` that for a real number `x` returns a real value. For example, such a function could be $f(x) = e^{-x}$:

```
def f(x):
    return exp(-x)
```

Write a Python function `plotfun(a, b, f, n)` that will return two arrays `xpts` and `ypts`. The former will contain x-coordinates of `n`-point linear subdivision of the interval `[a, b]`, the latter will contain the corresponding function values. Use the function `plotfun` to plot the graph of the function `f(x)` defined above, in the interval $[-1, 1]$. Axes should be scaled equally.

4.3 Planar curve

Consider two arbitrary Python functions `f(t)` and `g(t)` that for a real number `t` in the interval $[0, 1]$ return a real value. For example, such functions could be $f(t) = t^2 \cos(10\pi t)$ and $g(t) = t^2 \sin(10\pi t)$:

```python
def f(t):
    return cos(2*pi*t)

def g(t):
    return sin(2*pi*t)
```

Write a Python function `planarcurve(f, g, n)` that returns two arrays `xpts` and `ypts`. These arrays will contain 'x' and 'y' coordinates of `n` points lying on the planar curve $[f(t), g(t)]$. These points correspond to a linear subdivision of the parameterization interval $[0, 1]$ with `n` points. Use the function `planarcurve` to plot the curve corresponding to the functions `f(t)` and `g(t)` defined above. Axes should be scaled equally.

4.4 Random triangle

The function `random()` from the `random` library returns a random number between 0 and 1. Write a Python function `tria(a, b, c, d)` that returns six numbers x1, y1, x2, y2, x3, y3. These numbers represent vertices [x1, y1], [x2, y2], [x3, y3] of a random triangle that lies in the rectangle (a, b)x(c, d). Plot the triangle. Make sure that axes are scaled equally.

4.5 Wireframe plot

Write Python function `wireframeplot(f, x1, x2, y1, y2, nx=30, ny=30)` where `f(x, y)` is a real function of two variables, `(x1, x2)x(y1, y2)` is the plotting rectangle, and `nx`, `ny` are plotting subdivisions in the x and y axial directions. Test it on the function

$$f(x, y) = e^{-x^2-y^2}$$

in the square $(-3, 3) \times (-3, 3)$.

4.6 Solid surface plot

Write Python function `solidsurfaceplot(f, x1, x2, y1, y2, nx=30, ny=30)` where `f(x, y)` is a real function of two variables, `(x1, x2)x(y1, y2)` is the plotting rectangle, and `nx`, `ny` are plotting subdivisions in the x and y axial directions. Test it on the function

$$f(x, y) = 5 - (x^2 + y^2)$$

in the square $(-2, 2) \times (-2, 2)$.

4.7 Contour plot

Write Python function `contourplot(f, x1, x2, y1, y2, num_contours, nx =30, ny=30)` where `f(x, y)` is a real function of two variables, `(x1, x2)x(y1, y2)` is the plotting rectangle, `num_contours` is the number of contours to be displayed, and `nx`, `ny` are plotting subdivisions in the x and y axial directions. Display contour plot of the function

$$f(x, y) = xy$$

in the square $(-1, 1) \times (-1, 1)$ with 30 contours.

4.8 WebGL plot

Solve this exercise only if WebGL works on your computer (WebGL Tester can be found on NCLab front page). Write Python function webglplot(f, x1, x2, y1, y2, nx=30, ny=30) where f(x, y) is a real function of two variables, (x1, x2)x(y1, y2) is the plotting rectangle, and nx, ny are plotting subdivisions in the x and y axial directions. Test it on the function

$$f(x, y) = e^{-x^2-y^2}$$

in the square $(-2, 2) \times (-2, 2)$.

4.9 Pie chart plot

A family spends each month 25% of their budget on clothing, 20% on food, 10% on car payments, 20% on mortgage, 5% on utilities, and they save 20%. Plot a pie chart representing their spendings, and highlight the savings part via the `explode` function.

5 Variables

In this section we reinforce our understanding of local and global variables.

5.1 Arithmetic sequence

Arithmetic sequence is a progression of n numbers $a_1, a_2, \ldots, a_n$ that increase by the same difference d. For example,

$$5, 7, 9, 11, 13, 15, 17$$

is an arithmetic sequence with $n = 7$, $a_1 = 5$ and $d = 2$. It holds

$$a_n = a_1 + (n - 1)d.$$

There is a well known formula for the sum of such a sequence:

$$S = \frac{n}{2}(a_1 + a_n).$$

And now the exercise: Write a function `summation_arithmetic(n, a1, d)` that takes arbitrary values of n, a_1 and d as arguments, and calculates and returns the sum of the corresponding arithmetic sequence!

5.2 Geometric sequence

Geometric sequence is a progression of n numbers a_1, a_2, ..., a_n where the next number is calculated by multiplying the last one with a quotient q. For example,

$$1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$$

is a geometric sequence with $n = 5$, $a_1 = 1$ and $q = 1/2$. It is

$$a_n = a_1 \cdot q^{n-1}.$$

If $q \neq 1$, then the formula for the sum of such a sequence is

$$S = a_1 \frac{1 - q^n}{1 - q}$$

Write a function `summation_geometric(n, a1, q)` that takes arbitrary values of n, a_1 and $q \neq 1$ as arguments, and calculates and returns the sum of the corresponding geometric sequence!.

5.3 Graph length

Write a Python function `graphlength(f, a, b, n)` that calculates approximately the length of the graph of a function f(x) in the interval (a, b). Hint: Subdivide the interval (a, b) into 'n' equally long parts and approximate the graph in each one via a linear segment. These segments need to coincide with the function f(x) at all division points. Clearly, the answer will become more accurate as the number 'n' grows. Therefore, experiment with various values of 'n' until you are confident that the length that you calculated is accurate to two decimal digits!

5.4 3D curve length

Write a Python function `curvelength(f, g, h, n)` that calculates approximately the length of a 3D curve [f(t), g(t), h(t)] that is parameterized using three functions f(t), g(t), h(t) defined in the interval (0, 1). Hint: Subdivide the interval (0, 1) into 'n' equally long parts and approximate the curve in each one via a linear segment. Use your function to calculate approximately the length of a spiral [cos(10*t), sin(10*t), t]. Experiment with the number 'n' until you are certain that your result is accurate to two decimal digits.

5.5 Triangle area

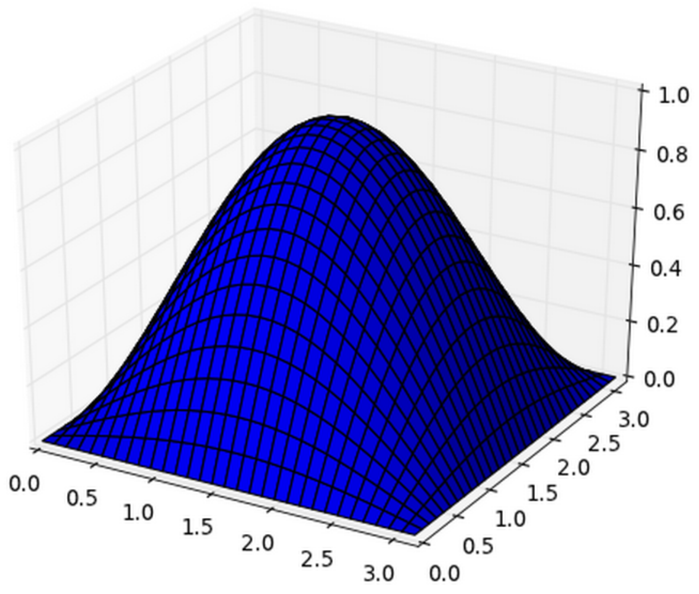
Write a Python function `trianglearea(v1, v2, v3)` that for three 3D points (lists) calculates the area of the corresponding triangle. Hint: Use Heron's formula

$$A = \sqrt{p(p-a)(p-b)(p-c)}$$

where p is half of the triangle's perimeter and a, b, c are its edge lengths.

5.6 Roof area

Write a Python function `roofarea(f, x1, x2, y1, y2, nx, ny)` that calculates approximately the area of a roof. The building has a rectangular footprint (x1, x2)x(y1, y2), and the roof is given via a function f(x, y).



This task may seem difficult but we can use similar trick as in Exercises 05.03 and 05.04: Split the rectangle (x1, x2)x(y1, y2) into nx times ny equally-sized small rectangles. Moreover, split each of these small rectangles diagonally into two triangles (there are two diagonals but it does not matter which one you choose). Over each triangle, span a linear plane that in the corners coincides with the function f(x, y). Use the function from Exercise 05.05 to calculate the triangle area. After adding the area of all these triangles together, you will have an approximation of the roof surface. Return it as a result of the function. Similarly to Exercises 05.03 and 05.04, experiment with various values of nx and ny until you are confident that the surface that you calculated is exact to two decimal digits! Test your program on the function f(x, y) = sin(x)*sin(y) in the interval (0, Pi)x(0, Pi) that is shown below.

6 Logic and probability

In this section we practice usage of logical operations and useful operations with random numbers.

6.1 Even or odd

Write a Python function `checknumber(n)` where `n` is an arbitrary integer number, that returns `True` if `n` is even and `False` otherwise.

6.2 Mysterious parabola

Consider a quadratic equation $ax^2 + bx + c = 0$ where a, b, c are real numbers and $a \neq 0$. Write a Python function `hasrealroots(a, b, c)` that returns `True` if the equation has at least one real root, and `False` otherwise.

6.3 Point in circle

Write a Python function `liesincircle(R, x, y)` that returns `True` if the point with planar coordinates `[x, y]` lies in a circle of radius `R` whose center is at the origin `[0, 0]`. If the point lies on the border of the circle or outside, the function should return `False`.

6.4 Slot machine

Write a Python function `slotmachine()` that simulates a simple slot machine by returning three random integers between 0 and 9. Use the function `random()` from the `random` library that returns a random real number between 0.0 and 1.0. Hint: Split the interval (0, 1) into 10 equally-long parts. Return 0 if the generated random number falls into the first one, 1 if it falls into the second one, etc. Integer part of a real number `x` is obtained via `int(x)`.

6.5 Dice game

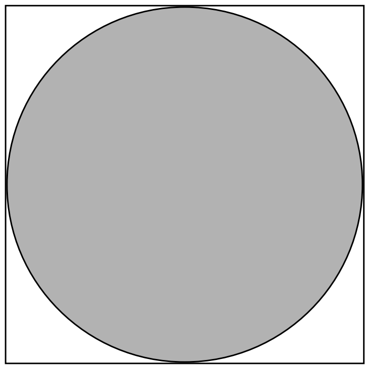
Write a Python function `dicegame()` that simulates a throw of two dice by returning a pair of random numbers 1 - 6. Make sure that every number has the same probablity. Use the function `random()` from the `random` library that returns a random real number between 0.0 and 1.0. Hint: Split the interval (0, 1) into six equally-long parts. Return 1 if the generated random number falls into the first one, etc. Integer part of a real number `x` is obtained via `int(x)`.

6.6 Cheat dice

Write a Python function `cheatdice(p1, p2, p3, p4, p5, p6)` that simulates a throw of two dice by returning a pair of random numbers 1 - 6. However, this time, you are using artificially altered dice where the probability of obtaining 1 is p1 instead of 1/6, probability of obtaining 2 is p2 instead of 1/6, etc. The sum of all p1, p2, ..., p6 equals to 1.0. Again use the function `random()` from the `random` library. Hint: Split the interval (0, 1) into six parts which now will not be equally-long. Their lengths will be p1, p2, ..., p6. If the generated random number falls into the first one, return 1, etc.

6.7 Monte Carlo pie

Probabilistic methods are often called *Monte Carlo methods*. Write a Python function `calculate_pi(n)` that will calculate and return an approximation of π using the following algorithm: Generate n random points inside the square $(-1, 1) \times (-1, 1)$. Count the number m of points that lie in the circle or radius $r = 1$ centered at the origin, as shown in the figure below.



Clearly the ratio of m and n is approximately the same as the ratio of the area C of the circle to the area S of the square.

$$\frac{m}{n} \approx \frac{C}{S}$$

At least for large values of n such as 10,000 this will be close. Assume that the number π satisfies $C = \pi r^2$. Knowing that $S = 4$, we can calculate

$$\frac{m}{n} \approx \frac{\pi}{4}$$

and thus

$$\frac{4m}{n} \approx \pi.$$

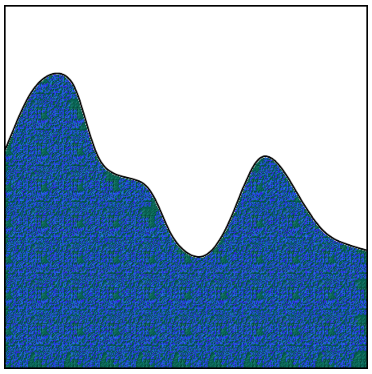
Hence, use the last formula to calculate an approximation of π using the numbers n and m.

6.8 Maximum of a function

Write a Python function `maxfun(f, a, b, n)` to calculate and return an approximate maximum of the function $f(x)$ in the interval (a, b). Hint: Cover the interval (a, b) with n equidistant points. Loop over the points, evaluate the function f at every one of them, and find the maximum value.

6.9 Monte Carlo area

Write a Python function `area_under_graph(f, a, b, max, n)` to calculate and return an approximation of the area under the graph of the function $f(x)$ in the interval (a, b). By the area under a graph we understand the area between the x-axis and the graph in the interval (a, b) as illustrated in the figure below.



Hint: Use the function `maximum(f, a, b, n)` from Exercise 6.8 to find an approximate maximum M of the function f in the interval (a, b). Generate n^2 random points in the rectangle $(a, b) \times (0, M)$. By m denote the number of points which lie under the function graph. Clearly, for larger n^2 such as 10000 the ratio of m and n^2 will be approximately the same as the ratio of the area A under the graph and the area R of the rectangle $(a, b) \times (0, M)$. Hence we obtain

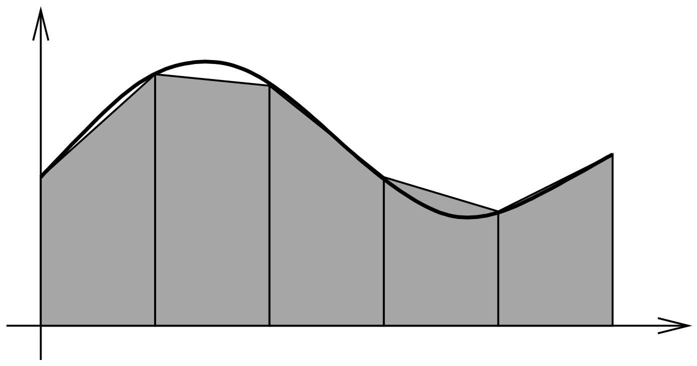
$$\frac{m}{n^2} \approx \frac{A}{R}.$$

From here, it is easy to express A as

$$A \approx \frac{Rm}{n^2}$$

6.10 Trapezoids

Write a Python function `trapezoids(f, a, b, n)` to calculate and return an approximate area under the graph of the function $f(x)$ in the interval (a, b). First split the interval (a, b) into n equally-long parts, and then add together the areas of all trapezoids under the graph of the function f, as shown in the image below.



You can notice that in some cases the entire trapezoid does not lie under the curve, but this is OK. It is important that the upper two vertices of each trapezoid always lie on the function graph. Hint: The area of a trapezoid with basis (c, d) and function values $f(c),\ f(d)$ at the upper vertices is

$$area = (d - c)\frac{f(c) + f(d)}{2}.$$

Test your code with the function $f(x) = \sin(x)$ in the interval $(0, \pi)$. In this case the exact area under the graph is $area = 2$.

7 Conditional Loop

In this section we solve problems that include iterations (repetitions) whose number is not known in advance.

7.1 Water jar

There is a jar of volume V that is full of water. Every day, P per cent of the water volume evaporates. Write a function `waterjar(V, P, R)` that prints the remaining water volume day by day. The process stops when the volume is less than or equal to a residual volume R. Your function should return the number of days the process took. Include a condition at the beginning of the function that will output a warning and return zero if $R > V$ or $V <= 0$ or $R <= 0$.

7.2 Throwing stones

Imagine that you stand on a very high bridge H meters above a river and throw a stone with velocity v in horizontal direction. The coordinates of the flying stone as functions of time t are $x(t) = tv$ and $y(t) = H - 0.5gt^2$ where $g = 9.81$ kgms^{-2} is the gravitational acceleration. Use these formulas to write a function `flyingstone(t, H, v)` that for any time instant `t` returns the x and y coordinates of the flying stone. Then calculate and plot the trajectory of the stone since the moment it leaves your hand until it falls into the river! Hint: Proceed by choosing a small time step `dt` such as 0.01 seconds, and use the function `flyingstone(t, H, v)` to generate a sequence of points for the `plot()` command.

7.3 Missing numbers

Write a function `missing_numbers()` that defines a variable $e = 1.0$ and performs a loop in which the value of e is divided by 10.0. The loop should run while $1.0 - e < 1.0$. Count how many times the loop will run and return this number. Note: Since $1.0 - e$ is no longer less than 1.0 in the computer arithmetic, this clearly shows that the computer arithmetic does not contain all real numbers!

7.4 Trapezoids revisited

In this exercise we will reuse the function `trapezoids(f, a, b, n)` from Exercise 6.10. Write a function `accurate_area(f, a, b, err)` that begins with approximationg the area using just one trapezoid: `trapezoids(f, a, b, 1)`. Then write a loop in which the number n is doubled and new approximation of the area is calculated. The loop should run while the absolute value of the difference between the last two approximations is greater than err. Return two values: the last (most accurate) area approximation, and the number of trapezoids used. Test your function on $f(x) = \exp(-x)$ in the interval $(0, 100)$ and $err = 10^{-8}$. (The exact area in this case is $area = 1$.)

7.5 Infinite sum

It is a well known fact in mathematics that the sum of the infinite sequence

$$\frac{1}{1} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dots$$

is a finite number. But what is this number? Let's find out! Write a Python function `sum(err)` that will keep adding the numbers in the above sequence together until the last increment is less than `err`. Then stop, and return two values: the latest value of the sum and the number of entries added.

8 Strings

In this section we will practice operations with text strings.

8.1 Spelling

Write a function `spell(str)` that prints the string `str` one character per line, and returns the number of characters in the string.

8.2 Inserting spaces

Write a function `insert_spaces(str)` that returns a new string that is obtained by inserting an empty character between all letters in `str`.

8.3 Counting chars

Write a function `countchars(str, L)` that counts all occurrences of the character (one-letter string) `L` in the string `str`, and return their number.

8.4 Counting words

Write a function `countwords(str, word)` that will count all occurences of the string `word` in the string `str`, and return their number. You can assume that `len(str)` is always greater or equal to `len(word)`.

8.5 Search and replace

Write a function `searchandreplace(str, word1, word2)` that will find all occurrences of the string `word1` in the string `str`, and replace them with the string `word2`. Return the new string and the number of replacements made.. Hint: Do not change the string `str` - instead, create a new string `str2` and add into it what is needed.

9 Tuples, Lists, and Dictionaries

In this section we will solve problems related to tuples, lists and dictionaries.

9.1 List reverse

Write a Python function `listreverse(L)` that reverts the list `L` and returns the result. Use of the Python built-in list reverse function is not allowed.

9.2 String to list

Write a Python function `str2list(str)` that for any text string `str` returns the list of words that it contains, in the order they appear. Repetitions are allowed. Words can be separated by empty character ' ', comma, period, question mark or exclamation mark.

10 More on Counting Loop

In this section we strengthen our understanding of the counting loop by solving problems that include repetitions whose number is known in advance.

10.1 Analyze string

Write a Python function `analyze_string(str)` that for the text string `str` returns the number of decimals '0' - '9'.

10.2 Multiplication table

Write a Python function `multable(N)` that returns a 2D array A containing the multiplicative table of the numbers $1, 2, \ldots, N$. In other words, $A[i-1][j-1] = ij$ for all $i, j = 1, 2, ..., N$. Hint: Use the Numpy command `zeros` to create the 2D array:

```
from numpy import zeros
A = zeros((N, N))
```

Entry at position `r`, `s` in A is accessed via `A[r][s]`. Keep in mind that indices start from zero.

10.3 Approximating π

The number π can be written as an infinite series

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots\right).$$

Write a Python function `approx_pi(n)` that returns an approximation of π using a truncated series with n terms.

10.4 Prime numbers

Write a Python function `primes(n)` that for a positive number `n` returns the list of first `n` prime numbers starting with 2.

10.5 List maximizer

Write a Python function `maximizer(L1, L2)` that takes two lists of real numbers and finds a number `n1` in `L1` and a number `n2` in `L2` such that the product of `n1` and `n2` is maximal. Return the numbers `n1` and `n2`.

10.6 Fibonacci

Write a Python function `fibonacci(n)` that returns a list of integer numbers representing the Fibonacci sequence. Recall that this sequence starts with 0, 1 and each next number is the sum of the two last numbers in the sequence.

10.7 Adding fractions

Write a Python function `addfractions(L)` where L is a list of integer pairs (two-element lists). Each integer pair represents the numerator and denominator of a fraction. Your function should add all these fractions together and obtain the numerator and denominator of the resulting fraction. Further, the numerator and denominator should be divided by the greatest common divisor to reduce the fraction to the simplest possible form. Hint: The function fraction.gcd(numerator, denominator) can be used to find the greatest common divisor of two integers.

11 Exceptions

In this section we put exceptions handling to practical use.

11.1 Sure plot

Implement a custom function `sureplot(f, a, b, label, n = 100)` where `f(x)` is a user-supplied real function, (a, b) is an interval, 'label' is a text string label for the plot, and n is the plotting subdivision of the interval (a, b). Implement exception handling in such a way that if the user's function is not defined for some x, a warning is printed and the corresponding point is not added to the plot. But the rest of the graph will be plotted normally. For example, if the user wants to plot the function log(x) in the interval (-1, 1) then the left half of the graph will be missing but the rest will be displayed correctly.

```python
def f(x):
    return log(x)
```

in the interval `(-1, 1)` then the left half of the graph should be missing but the rest will be displayed correctly. Hint: Use mathematical functions from the "math" module ("from math import log").

11.2 Syntax check

Implement a Boolean function `syntaxcheck(str)` that evaluates any user-supplied text expression `str`. If the code in `str` contains a syntax error, return False. Otherwise return True. Hint: Text string `str` can be evaluated in Python via the command `exec(str, )`. Do not worry about the second argument - it serves for passing data into the evaluated expression. Check Python tutorial for more details if interested.

12 Object-Oriented Programming

In this section we practise our understanding of object-oriented programming.

12.1 Bank account

Implement a Python class `Account` to simulate a bank account. It should have two variables `savings` and `checking`, and the following methods:

– Constructor `__init__(self)` that creates an empty checking and savings account.
– `add_to_checking(self, amount)`: Add `amount` to checking.
– `add_to_savings(self, amount)`: Add `amount` to savings.
– `get_balance_checking(self)`: Return the balance in the checking account.
– `get_balance_savings(self)`: Return the balance in the savings account.

- `transfer_to_checking(self, amount)`: Transfer `amount` from savings to checking and return True. If there are insufficient funds, do not perform the operation, write a warning, and return False.
- `transfer_to_savings(self, amount)`: Transfer `amount` from checking to savings and return True. If there are insufficient funds, do not perform the operation, write a warning, and return False.
- `withdraw_from_checking(self, amount)`: Withdraw `amount` from checking and return True. If there are insufficient funds, do not perform the operation, write a warning, and return False.
- `withdraw_from_savings(self, amount)`: Withdraw `amount` from savings and return True. If there are insufficient funds, do not perform the operation, write a warning, and return False.

12.2 Word magic

Implement a Python-dictionary-based smart translator class `Wordmagic` that will have the following methods:

- Constructor `__init__(self)` that creates an empty dictionary.
- `add(self, word, tran)`: Boolean method that adds a pair of text strings into the dictionary. The former is the key, the latter the corresponding value. If `word` is already present in the dictionary, let the user know via a warning message, do not add it, and return `False`. Otherwise return `True`.
- `read(self, d)`: Method that imports existing dictionary.
- `translate(self, word)`: Method that returns a Boolean and a string. If `word` is present in the keys, return True and its translation. Otherwise print a warning, and return False and an empty string.
- `export(self)`: Method that returns the dictionary.
- `reverse(self)`: Method that swaps all keys and values in the dictionary.

12.3 Number magic

Implement a Python class `Numbermagic` whose data is just one integer number, and whose methods are as follows:

- Constructor `__init__(self, num)` that takes one integer number as parameter and stores it in the object. If the user supplies a non-integer number, print a warning and round it to an integer. Hint: Use Python bbuilt-in function `isinstance(n, int)` to determine whether the number 'n' is integer.
- `insert(self, num)`: Method that replaces the number stored in the object with `num`. As in constructor, make sure that the number is an integer.

– `ispositive(self)`: Boolean method that returns `True` if the number stored in the object is greater than zero, `False` otherwise.
– `iseven(self)`: Boolean method that returns `True` if the number stored in the object is even, `False` otherwise.
– `isprime(self)`: Boolean method that returns `True` if the number stored in the object is a prime, `False` otherwise.

12.4 Vector magic

Implement a Python class `Vectormagic` whose data are two three-dimensional vectors. All vectors are represented by lists of length three. The class has the following methods:

– Constructor `__init__(self, u, v)` that takes two vectors as parameters and stores them in the object. If the length of any of the two vectors is not three, raise a ValueError exception.
– `insert(self, u, v)`: Method that replaces the vectors in the object with new ones. Make sure that u, v are 3D vectors same as in the constructor.
– `norm(self, z)`: Method to calculate the Euclidean norm of vector `z`.
– `areparallel(self)`: Boolean method that returns `True` if the vectors are parallel, `False` otherwise. Note: If you are going to compare a real number against zero, use a small tolerance, such as `1e-8`.
– `arenormal(self)`: Boolean method that returns `True` if the vectors are perpendicular, `False` otherwise. Again: If you are going to compare a real number against zero, use a small tolerance, such as `1e-8`.
– `innerproduct(self)`: Returns one real number which is the inner product of the two vectors.
– `vectorproduct(self)`: Returns a vector (list of length three) which is the vector product of the two vectors.

13 Class Inheritance

In this section we solve problems involving creating descendants of classes in object-oriented programming.

13.1 Number master

Create a descendant `Numbermaster` of the class `Numbermagic` from Exercise 12.3 that in addition to the functionality contained in the class `Numbermagic` has one new method:

- `factorize(self)`: Method that returns a Python list containing primes `p1, p2, ..., pn` that form the prime number factorization of the number stored in the object (repetitions are allowed).

13.2 Vector master

Create a descendant `Vectormaster` of the class `Vectormagic` from Exercise 12.4 that in addition to the functionality contained in the class `Vectormagic` has one new method:

- `project(self, w)`: If the two vectors contained in the object form a plane, the method calculates and returns the projection of the vector `w` into this plane. Otherwise the method prints a warning and returns a zero vector.

Part III

Review Questions

1 Introduction

1. *What is the difference between compiled and scripting programming languages?*

   A1 Programs written in compiled languages are binary files, programs written in scripting languages are text files.
   A2 Compiled languages are easier to use than scripting languages.
   A3 Programs written in scripting languages are usually more efficient than programs written in compiled languages.
   A4 Scripting languages do not require a compiler.

2. *What languages take better advantage of the underlying hardware architecture and why?*

   A1 Scripting languages because they are less hardware-dependent.
   A2 Compiled languages because the executable files are tailored to the concrete hardware.
   A3 Compiled languages because they do not require a linker.
   A4 Scripting languages because they do not require a compiler.

3. *Give three examples of compiled programming languages.*

   A1 Fortran, C++ and Lua.
   A2 Python, Ruby and C.
   A3 C, C++ and Fortran.
   A4 Python, Lua and Perl.

4. *Give three examples of interpreted programming languages.*

   A1 Pascal, Perl and Python.
   A2 Lua, C and C++.
   A3 Perl, Python and Ruby.
   A4 C, C++ and Fortran.

5. *Where does the name "Python" of the programming language come from?*

   A1 The snake, *Python regius*.
   A2 TV show in Great Britain.
   A3 Brand name of remote start systems.
   A4 Brand name of aquarium products.

6. *When was the implementation of Python started?*

   A1 1989
   A2 1995
   A3 2000
   A4 2005

7. *Name three programming styles that Python permits.*

   A1 Compiled, interpreted, scripting.
   A2 Procedural, object-oriented, functional.
   A3 Procedural, object-oriented, binary.
   A4 Compact, literal, extended.

8. *How can displayed Python projects be cloned?*

   A1 In Python worksheet through the *File* menu.
   A2 Through File Manager's *Project* menu.
   A3 Using the icon *Displayed projects* on Desktop.
   A4 Python projects cannot be cloned.

9. *How can new Python project be launched?*

   A1 Through the *Program* menu.
   A2 Through File Manager's *Settings* menu.
   A3 Through File Manager's *Project* menu.
   A4 Through the Programming module on Desktop.

10. *How are Python programs processed in NCLab?*

   A1 They are translated into Javascript and run in your web browser.
   A2 They are interpreted on a remote server.
   A3 They are interpreted on your PC computer / laptop / tablet.
   A4 They are translated into Flash and run in your web browser.

11. *What are the types of cells that a Python worksheet can contain?*

   A1 Error message cells.

A2 Output cells.
A3 Code cells.
A4 Descriptive cells.

12. *How can all code cells in a Python worksheet be evaluated at once?*

A1 Type Evaluate in the last code cell and hit ENTER.
A2 Click on the green arrow under the last code cell.
A3 Click on *Evaluate all* in the *File* menu.
A4 Click on the green arrow button in the upper menu.

13. *If your Python worksheet has multiple code cells, how can a single code cell be evaluated?*

A1 Click on `save` under the code cell.
A2 Click on the green arrow button in the upper menu.
A3 Click on the green arrow under the code cell.
A4 Click on the red button in the menu.

14. *What is the way to add a new code cell?*

A1 Via a button located under any code cell or descriptive cell.
A2 Click on *New* in the *File* menu.
A3 Click on the larger "A" icon in the menu.
A4 Click on *New code cell* in *Edit* menu.

15. *How can a new descriptive cell be added?*

A1 Click on the smaller "A" icon in the menu.
A2 Click on *New descriptive cell* in the *Edit* menu.
A3 Click on *New* in the *File* menu.
A4 Via a button located under any code cell or descriptive cell.

16. *What is the correct way to remove a code, output, or descriptive cell?*

A1 Cells, once created, cannot be removed.
A2 Via a button located under the cell.
A3 Click on *Remove active cell* in the *Edit* menu.
A4 Click on *Close* in the *File* menu.

17. *Which of the following are scientific libraries for Python?*

A1 GNU Scientific Library
A2 Numpy
A3 Scipy
A4 Sympy

18. *What is the correct way to print the text string "I like Python" ?*

A1 `print "I like Python"`
A2 `write "I like Python"`
A3 `write 'I like Python'`
A4 `write »I like Python«`

2 Using Python as a Calculator

1. *What will be the result of* `5 * 4 - 3 * 2` *?*

A1 `34`
A2 `10`
A3 `0`
A4 `14`

2. *What will be the result of* `11/4` *?*

A1 `2.75`
A2 `2`
A3 `3`
A4 Error message.

3. *Type* 3^2 *using Python syntax!*

A1 `3^2`
A2 `3^^2`
A3 `3*^2`
A4 `3**2`

4. *What will be the result of* `(-2)**3` *?*

   A1 `2`
   A2 `8`
   A3 `-8`
   A4 Error message

5. *What will be the result of* `(-2)**(3.5)` *?*

   A1 `-8`
   A2 `11.313708498984761`
   A3 `-11.313708498984761`
   A4 Error message

6. *What does the* modulo *operation do ?*

   A1 Rounding up
   A2 Remainder after integer division
   A3 Integer division
   A4 Rounding down

7. *Type* 5 *modulo* 2 *using Python syntax!*

   A1 `5 / 2`
   A2 `5 % 2`
   A3 `5 & 2`
   A4 `5 && 2`

8. *What will be the result of* `1**4*2` *?*

   A1 `0`
   A2 `1`
   A3 `2`
   A4 `4`

9. *What will be the result of* `6 / 3*2` *?*

A1 `1`
A2 `0`
A3 `4`
A4 `1`

10. *What will be the result of* `12 / 4 / 3` *?*

    A1 `1`
    A2 `12`
    A3 `4`
    A4 `3`

11. *What is the correct way to import all functionality from the Numpy library ?*

    A1 `from numpy import all`
    A2 `from numpy import everything`
    A3 `from numpy import *`
    A4 `from numpy import functions`

12. *Which code will calculate sin($\pi/4$)?*

    A1
    ```
    from numpy import sin
    sin(pi/4)
    ```
    A2
    ```
    from numpy import sin, pi
    sin(pi/4)
    ```
    A3
    ```
    from trigonometry import sin
    sin(pi/4)
    ```
    A4
    ```
    from trigonometry import sin, pi
    sin(pi/4)
    ```

13. *What is the correct way to use the* `fractions` *library to add the fractions* $5/27 + 5/9 + 21/81$*?*

    A1
    ```
    from fractions import Fraction
    Fraction(5/27) + Fraction(5/9) + Fraction(21/81)
    ```
    A2
    ```
    from fractions import Fraction
    Fraction(5/27 + 5/9 + 21/81)
    ```

A3
```
from fractions import Fraction
Fraction(5/27. + 5/9. + 21/81.)
```
A4
```
from fractions import Fraction
Fraction(5, 27) + Fraction(5, 9) + Fraction(21, 81)
```

14. *Which of the following codes will calculate the greatest common divisor of the numbers* 1377 *and* 4131 *?*

A1
```
from fractions import gcd
gcd(1377, 4131)
```
A2
```
from fractions import Fraction, gcd
gcd(1377, 4131)
```
A3
```
from fractions import *
gcd(1377, 4131)
```
A4
```
from fractions import *
gcd(4131, 1377)
```

15. *What will be the result of the following code?*

```
from fractions import gcd
gcd(20, 25, 55)
```

A1 `5`
A2 `20`
A3 `25`
A4 Error message

16. *Which code will generate a random real number between 0 and 1 ?*

A1
```
from random import random
random(0, 1)
```
A2
```
from random import random
random()
```
A3
```
from random import random
random(1.0)
```
A4
```
from random import rand
rand()
```

17. *With the* `random()` *function imported, which code will generate a random integer number between 5 and 10 ?*

    A1 `5 + int(5 * random())`
    A2 `5 + 5 * int(random())`
    A3 `int(5 + 5 * random())`
    A4 `int(10 * random()) - 5`

18. *Which of the following is / are correct way(s) to define a complex number?*

    A1 `2 + 3j`
    A2 `2 + 3J`
    A3 `2 + 3*j`
    A4 `2 + 3*J`

3 Functions

1. *Why do we define custom functions in programming?*

    A1 To isolate self-contained functionality and make it easily reusable.
    A2 To make computer programs run faster.
    A3 To split long programs into multiple segments that have fewer lines.
    A4 We should not use functions, it is not a good programming practice.

2. *When does a Python function have to return a value?*

    A1 When it accepts arguments.
    A2 When it accepts default arguments.
    A3 When it is used together with a print statement.
    A4 Never.

3. *Which of the following codes are syntactically correct?*

    A1
    ```
    def subtract(a, b)
        return a - b
    ```

A2
```
def subtract(a, b):
    return a - b
```
A3
```
def subtract[a, b]
    return a - b
```
A4
```
def subtract(a, b = 5):
    return a - b
```

4. *When will the function* `add(a, b)` *defined below accept text strings as arguments without throwing an error?*

```
def add(a, b):
    return a + b
```

A1 Only when the two text strings have the same length.
A2 Always.
A3 Only when the two text strings have different lengths.
A4 Never.

5. *When do we have to specify argument types in Python functions?*

A1 Never.
A2 Always.
A3 Always except for numbers and text strings.
A4 Only when default arguments are used.

6. *We need a function* `f(X)` *that accepts a number* `X` *and returns three values:* `3*X`*,* `5*X`*,* `7*X`*. Which of the following functions will do that?*

A1
```
def f(X):
    return 3*X
    return 5*X
    return 7*X
```
A2
```
def f(X):
    return 3*X, 5*X, 7*X
```
A3
```
def f(X):
    return 3*X + 5*X + 7*X
```

A4
```
def f(X):
    3*X, 5*X, 7*X
    return
```

7. *Which of the following four function definitions are syntactically correct?*

A1
```
from numpy import sqrt
def hypotenuse(a, b):
    return sqrt(a**2 + b**2)
```
A2
```
from numpy import sqrt
def hypotenuse(a=5, b):
    return sqrt(a**2 + b**2)
```
A3
```
from numpy import sqrt
def hypotenuse(a, b=5):
    return sqrt(a**2 + b**2)
```
A4
```
from numpy import sqrt
def hypotenuse(a=5, b=5):
    return sqrt(a**2 + b**2)
```

8. *For the function* `compose` *defined below, what function call(s) will not cause an error message?*

```
def compose(a, x = 1, y = 2, z = 3):
    return a + x + y + z
```

A1 `print compose(1)`
A2 `print compose(1, y = 10)`
A3 `print compose(1, x = 20, z = 5)`
A4 `print compose(1, 10, x = 20, z = 5)`

4 Colors and plotting

1. *Which of the following RGB values define(s) a green color?*

A1 [1, 0, 0]
A2 [1, 0.5, 0]

A3 [0, 0.5, 0]
A4 [0, 0.5, 1]

2. *Which of the following RGB values define(s) a shade of grey?*

A1 [0.2, 0.2, 0.2]
A2 [0.5, 0.6, 0.7]
A3 [1, 0.5, 0]
A4 [1, 0.5, 1]

3. *Which of the following RGB values define(s) color that is closest to purple?*

A1 [1, 0.8, 0]
A2 [0.5, 0.6, 0.1]
A3 [0.5, 0, 0.5]
A4 [1, 0.5, 1]

4. *Which of the following codes will draw a square with vertices [0, 0], [1, 0], [1, 1], [0, 1]?*

A1
```
from pylab import *
x = [0.0, 1.0, 1.0, 0.0]
y = [0.0, 0.0, 1.0, 1.0]
clf()
plot(x, y)
lab.show()
```
A2
```
from pylab import *
x = [0.0, 0.0, 1.0, 1.0]
y = [0.0, 1.0, 1.0, 0.0]
clf()
plot(x, y)
lab.show()
```
A3
```
from pylab import *
x = [0.0, 1.0, 1.0, 0.0, 0.0]
y = [0.0, 0.0, 1.0, 1.0, 0.0]
clf()
plot(x, y)
lab.show()
```

A4
```
from pylab import *
x = [0.0, 0.0, 1.0, 1.0, 0.0]
y = [0.0, 1.0, 1.0, 0.0, 1.0]
clf()
plot(x, y)
lab.show()
```

5. *Which of the following codes will define an array* `X` *of 20 equidistant points covering the interval [0, 10]?*

A1
```
from numpy import linspace
X = linspace(0, 10, 20)
```
A2
```
from numpy import linspace
X = linspace(0, 10, 0.5)
```
A3
```
from numpy import linspace
X = linspace(0 : 10 : 20)
```
A4
```
from numpy import linspace
X = linspace(0 : 10 : 0.5)
```

6. *What will be the output of the following code, each value rounded to an integer value?*

```
from numpy import linspace, sin, pi
X = linspace(0, 3*pi, 4)
Y = sin(X)
print Y
```

A1 `[0 0 0]`
A2 `[0 0 0 0]`
A3 `[0 1 0 -1 0]`
A4 `[0 0 0 0 0]`

7. *What is the meaning of the Pylab* `clf()` *function?*

A1 Cleans the canvas.
A2 Shows the legend.
A3 Makes both axes to be equally scaled.
A4 Suppresses warnings.

8. *What Pylab function is used to enforce equal scaling of the horizontal and vertical axes?*

   A1 `axes = "equal"`
   A2 `equal('axes')`
   A3 `axes('equal')`
   A4 `axis('equal')`

9. *What Pylab function is used to show the legend? Should it be used before or after the* `plot()` *function?*

   A1 `showlegend()`, to be used before the `plot()` function.
   A2 `showlegend()`, to be used after the `plot()` function.
   A3 `legend()`, to be used before the `plot()` function.
   A4 `legend()`, to be used after the `plot()` function.

10. *Which of the following codes will plot a cosine function in the interval $(0, \pi)$ using a dashed green line?*

    A1
    ```
    from numpy import *
    from pylab import *
    x = linspace(0, pi, 100)
    y = cos(x)
    axis=("equal")
    clf()
    plot(x, y, 'g-', label="cos(x)")
    legend()
    lab.show()
    ```
    A2
    ```
    from numpy import *
    from pylab import *
    x = linspace(0, pi, 100)
    y = cos(x)
    clf()
    plot(x, y, 'g--')
    lab.show()
    ```
    A3
    ```
    from numpy import *
    from pylab import *
    x = linspace(0, pi/2, 100)
    y = cos(x)
    ```

```
axis=("equal")
clf()
plot(x, y, 'g--', label="cos(x)")
legend()
lab.show()
```

A4
```
from numpy import *
from pylab import *
x = linspace(0, pi, 100)
y = cos(x)
axis=("equal")
clf()
plot(x, y, 'b--', label="cos(x)")
legend()
lab.show()
```

11. *We have two arrays* `x` *and* `y` *created via the* `linspace` *function. What Numpy function will create a 2D Cartesian product grid of them?*

A1 `X, Y = meshgrid(x, y)`
A2 `X, Y = cartesiangrid(x, y)`
A3 `X, Y = productgrid(x, y)`
A4 `X, Y = product(x, y)`

12. *The arrays* `X` *and* `Y` *represent a 2D Cartesian product grid. How can we assign values of the function* $1/(1 + x^2 + y^2)$ *to the grid points?*

A1 `Z = 1. / (1 + exp(X) + exp(Y))`
A2 `Z = 1. / (1 + X*X + Y*Y)`
A3 `Z = 1. / (1 + X^2 + Y^2)`
A4 `Z = 1. / (1 + X**2 + Y**2)`

13. *Only one of the following codes will display a simple pie chart. Which one is it?*

A1
```
from pylab import *
data = [20 %, 40 %, 40 %]
labels = ['bronze', 'silver', 'gold']
clf()
axis('equal')
```

```
    pie(data, labels=labels)
    lab.show()
A2 from pylab import *
    data = ['bronze', 'silver', 'gold']
    labels = [20, 40, 40]
    clf()
    axis('equal')
    pie(data, labels=labels)
    lab.show()
A3 from pylab import *
    data = [20, 40, 40]
    labels = ['bronze', 'silver', 'gold']
    clf()
    axis('equal')
    pie(data, labels=labels)
    lab.show()
A4 from pylab import *
    data = [20, 40, 40]
    labels = ['bronze', 'silver', 'gold']
    clf()
    axis('equal')
    pie()
    lab.show()
```

14. *Which of the following codes will display a simple bar chart with the values 5, 7, 3, 9 with bars starting at 0, 2, 4, and 6 on the horizontal axis?*

```
A1 from pylab import *
    data = [5, 7, 3, 9]
    clf()
    bar([0, 2, 4, 6], data)
    lab.show()
A2 from pylab import *
    data = [5, 7, 3, 9]
    clf()
    bar_chart([0, 2, 4, 6], data)
    lab.show()
A3 from pylab import *
    data = [0, 2, 4, 6]
```

```
   clf()
   bar([5, 7, 3, 9], data)
   lab.show()
```
A4
```
   from pylab import *
   data = [0, 2, 4, 6]
   clf()
   bar_chart([5, 7, 3, 9], data)
   lab.show()
```

5 Variables

1. *What are correct ways to assign the value* `2.0` *to a variable* `var`*?*

   A1 `val := 2.0`
   A2 `val == 2.0`
   A3 `val(2.0)`
   A4 `val = 2.0`

2. *What of the following are correct ways to store the text string "*`Pontiac`*" in a variable* `car`*?*

   A1 `car = Pontiac`
   A2 `car = "Pontiac"`
   A3 `car = 'Pontiac'`
   A4 `car := "Pontiac"`

3. *What is the correct way to create a Boolean variable* `answer` *whose value is* `True`*?*

   A1 `answer = True`
   A2 `answer = 'True'`
   A3 `answer = 1`
   A4 `answer = "True"`

4. *None of the variables* `a`*,* `b` *was declared before. Which of the following codes are correct?*

   A1 `a = b = 1`

A2 `a = 1 = b`
A3 `a = a`
A4 `a + b = 5`

5. *What of the following is the correct way to increase the value of an existing integer variable* `val` *by 10 and print the result?*

A1 `val + 10`
A2 `print val + 10`
A3 `print val += 10`
A4
```
val += 10
print val
```

6. *When can a given variable have different types in various parts of a Python program?*

A1 Any time.
A2 Never.
A3 Only when variable shadowing takes place.
A4 Only when the types are real number and integer number.

7. *There are two variables* `a` *and* `b` *whose values need to be swapped. Which of the following codes will do it?*

A1 `a = b = a`
A2
```
a = b
b = a
```
A3
```
a = b
c = a
b = a
```
A4
```
c = a
a = b
b = c
```

8. *Should we preferably use local variables, or global variables, and why?*

A1 Global variables because they are easily accessible from any part of the code.
A2 Global variables because they make our program faster.

A3 Local variables because then we can use shadowing.
A4 Local variables because our code is less prone to mistakes.

9. *Should we use global variables in functions and why?*

A1 Yes, the function definition is simpler.
A2 Yes, the program is less prone to mistakes.
A3 No, it makes the code less transparent and more prone to mistakes.
A4 No, Python does not allow it.

10. *What is "shadowing of variables"?*

A1 There are two or more functions that all use a local variable of the same name.
A2 The type of a global variable is changed by assigning value of a different type to it.
A3 The type of a local variable is changed by assigning value of a different type to it.
A4 There is a local variable whose name matches the name of a global one.

11. *What will be the output of the following code?*

```
val = 5.0
def power(x, p):
    val = x**p
    return val
result = power(3, 2)
print val
```

A1 `9.0`
A2 `5.0`
A3 Error message
A4 `9.0`
`5.0`

12. *Identify the output of the following code!*

```
from numpy import e
def add(c, d):
    return c + d
e = add(10, 20)
print e
```

A1 `30`
A2 `2.718281828459045`
A3 `32.718281828459045`
A4 Error message

6 Logic and Probability

1. *What is the output of the following code?*

```
a = True
b = False
c = a and b
print c
```

A1 `False`
A2 Undefined
A3 `True`
A4 Error message

2. *What is the output of this code?*

```
a = True
b = False
c = a or b
print c
```

A1 `True`
A2 Error message
A3 `False`

A4 Undefined

3. *What value will the variable* `d` *have after the following code is run?*

```
d = True
if d != False:
    d = not(d)
```

A1 `True`
A2 `False`
A3 `True` in Python 2.7 and `False` in Python 3.0
A4 Undefined

4. *What will the following code print?*

```
val = 2 + 3
if val = 6:
    print "Correct result."
else:
    print "Wrong result."
```

A1 `Correct result.`
A2 `Wrong result.`
A3 `5`
A4 Error message

5. *What will be the value of the variable* `var` *after the following code is run?*

```
n = 10 + 5
m = 4 * 5
var = n == m
```

A1 `15`
A2 `20`
A3 `False`
A4 `True`

6. *Let* `a` *and* `b` *be Boolean variables. What is the output of the following code?*

```
c = (a or b) or not (a or b)
print c
```

A1 `False`
A2 Undefined.
A3 `False` or `True`, depending on the values of `a` and `b`.
A4 `True`

7. *What are Monte Carlo methods in scientific computing?*

A1 Methods that use large numbers of random values.
A2 Methods that succeed or fail randomly.
A3 Methods whose outcome is either `True` or `False`
A4 Methods for efficient evaluation of Boolean expressions..

8. *Which of the four values below will be closest to the output of this program?*

```
from random import random
n = 1000000
m = 0
# Repeat n times:
for i in range(n):
    # Generate random real number between -2 and 2:
    x = 4 * random() - 2
    if x**2 - 1 < 0:
        m += 1
# Print the ratio of m and n:
print float(m) / n
```

A1 `0.0`
A2 `4.0`
A3 `0.5`
A4 `1.0`

7 Conditional Loop

1. *When should the `elif` statement be used?*

   A1 It should not be used, it is a bad programming practice.
   A2 When there are more than two cases in the `if - else` statement.
   A3 When the logical expression in the `if` branch is not likely to be True.
   A4 When the logical expression in the `if` branch is likely to be True.

2. *What will be the output of the following program?*

```
day = 11
if day == 1:
    print "Monday"
elif day == 2:
    print "Tuesday"
elif day == 3:
    print "Wednesday"
elif day == 4:
    print "Thursday"
elif day == 5:
    print "Friday"
elif day == 6:
    print "Saturday"
else:
    print "Sunday"
```

   A1 There is no output.
   A2
```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```
   A3 `Sunday`
   A4 Error message

3. *When should the* `while` *loop be used?*

   A1 When we cannot use the `if - else` condition.
   A2 When we know exactly how many repetitions will be done.
   A3 When the loop contains a variable that decreases to zero.
   A4 When the number of repetitions is not know a priori.

4. *What will be the output of the following program?*

```
n = 1
while n < 100:
    n *= 2
print n
```

   A1 `64`
   A2 `128`
   A3 `1`
   A4 This is an infinite loop, there is no output.

5. *What is the purpose of the* `break` *statement?*

   A1 Stop the program inside of a loop.
   A2 Exit a loop. If multiple loops are embedded, exit all of them.
   A3 Exit the body of an `if` or `else` statement.
   A4 Exit a loop. If multiple loops are embedded, exit just the closest one.

6. *What is the purpose of the* `continue` *statement?*

   A1 Continue repeating the body of the loop after the loop has finished.
   A2 Skip the rest of the loop's body and continue with next cycle.
   A3 Continue to the next command.
   A4 Continue to the first line after the loop's body.

7. *What will be the output of the following program?*

```
a = 0
while True:
    a += 1
    if a < 8:
        continue
    print a
    break
```

A1
```
0
1
2
3
4
5
6
7
8
```

A2
```
1
2
3
4
5
6
7
8
```

A3
```
1
2
3
4
5
6
7
```

A4
```
8
```

8. *What is the Newton's method in scientific computing?*

A1 Method to determine force from mass and gravity.
A2 Method to approximate solutions to nonlinear equations.
A3 Method to calculate Newton integrals.
A4 Method to determine duration of free fall of an apple.

8 Strings

1. *What of the following are correct ways to include quotes in a string?*

   A1 `"I say "goodbye", you say "hello""`
   A2 `"I say /"goodbye/", you say /"hello/""`
   A3 `"I say \"goodbye\", you say \"hello\""`
   A4 `"I say \'goodbye\', you say \'hello\'"`

2. *What of the following are correct ways to define a multiline string?*

   A1
   ```
   /*
   I say "High", you say "Low".
   You say "Why?" And I say "I don't know".
   */
   ```
   A2
   ```
   """\
   I say "High", you say "Low".
   You say "Why?" And I say "I don't know".\
   """
   ```
   A3
   ```
   I say "High", you say "Low". \
   You say "Why?" And I say "I don't know".
   ```
   A4
   ```
   "I say "High", you say "Low". \
   You say "Why?" And I say "I don't know"."
   ```

3. *What output will be produced by the following code?*

   ```
   s1 = "Thank you"
   s2 = "very"
   s3 = "much!"
   print s1 + 5*s2 + s3
   ```

   A1 `Thank you very very very very very much!`
   A2 `Thank youveryveryveryveryverymuch!`
   A3 `Thankyouveryveryveryveryverymuch!`
   A4 None - an error will be thrown.

4. *What output will be produced by the following code?*

```
s1 = "intermediate"
s2 = s1[7] + s1[4] + s1[3] + s1[6] + s1[5]
print s2
```

A1 `dreem`
A2 `dream`
A3 `dieta`
A4 `tamer`

5. *What output will be produced by the following code?*

```
s1 = "intermediate"
s2 = s1[:5]
print s2
s3 = s1[5:10]
print s3
s4 = s1[-2] + s1[-1]
print s4
```

A1 `intermediate`

A2
```
inter
media
et
```

A3
```
inter
media
te
```

A4 None - an error will be thrown.

6. *What is the correct way to measure and print the length of a string* `str`*?*

A1 `print length(str)`
A2 `print len(str)`
A3 `print abs(str)`
A4 `print str[0]`

7. *What will be the output of the following code?*

```
print range(2, 5)
```

A1 `[2, 3, 4, 5]`
A2 `[2, 3, 4, 5, 6]`
A3 `[2, 3, 4]`
A4 `[5, 6]`

8. *What output corresponds to the following code?*

```
word = "breakfast"
for m in range(5, 9):
    print word[m]
```

A1 `fast`
A2 `kfas`
A3
```
f
a
s
t
```
A4 `kfas`

9 Tuples, Lists, and Dictionaries

1. *What is the variable* `var`*?*

```
var = (1, 2, 3, 'A', 'B', 'C', "alpha", "beta", "gamma")
```

A1 List.
A2 Tuple.
A3 Dictionary.
A4 None of the above.

2. *What will be the output of the following program?*

```
var = (1, 2, 3, 'A', 'B', 'C', "alpha", "beta", "gamma")
print var[5]
print var[:3]
print var[6:8]
```

A1 `B`
`(1, 2, 3)`
`('alpha', 'beta', 'gamma')`
A2 `C`
`(1, 2, 3)`
`('alpha', 'beta')`
A3 `B`
`(1, 2, 3)`
`('C', 'alpha', 'beta', 'gamma')`
A4 `B`
`(1, 2, 3)`
`('alpha', 'beta')`

3. *Can new items be added to a tuple?*

A1 Yes but only to an empty tuple.
A2 Yes but only if all items are of the same type.
A3 No.
A4 Yes but only if not all items are of the same type.

4. *What is the correct way to determine the length of a tuple* `T`*?*

A1 `length(T)`
A2 `tlength(T)`
A3 `len(T)`
A4 `tlen(T)`

5. *What is the variable* `names`*?*

```
names = ["John", "Jake", "Josh"]
```

A1 List.

A2 Tuple.
A3 Dictionary.
A4 None of the above.

6. *Identify the output of the following code!*

```
names = ["John", "Jake", "Josh"]
name = names.del[1]
print name
```

A1 `John`
A2 `Jake`
A3 `Josh`
A4 Error message.

7. *Identify the output of the following code!*

```
names = ["John", "Jake", "Josh"]
names.append("Jerry")
print names
```

A1 `['John', 'Jake', 'Josh', 'Jerry']`
A2 `('John', 'Jake', 'Josh', 'Jerry')`
A3 `['Jerry', 'John', 'Jake', 'Josh']`
A4 Error message.

8. *Identify the output of the following code!*

```
names = ["John", "Jake", "Josh"]
names.pop(0)
print names
```

A1 `['Jake', 'Josh']`
A2 `('John', 'Jake', 'Josh')`
A3 `John`
A4 Error message.

9. *Identify the output of the following code!*

```
names = ["John", "Jake", "Josh"]
names.insert(1, "Jenny")
print names
```

A1 `['Jenny', 'John', 'Jake', 'Josh']`
A2 `['John', 'Jenny', 'Jake', 'Josh']`
A3 `['Jenny', 'Jake', 'Josh']`
A4 Error message.

10. *Identify the output of the following code!*

```
names = ["John", "Jake", "Josh"]
names.reverse()
print names
```

A1 `['John', 'Josh', 'Jake']`
A2 `['Josh', 'John', 'Jake']`
A3 `['Josh', 'Jake', 'John']`
A4 Error message.

11. *Identify the output of the following code!*

```
names = ["John", "Jerry", "Jake", "Josh", "Jerry"]
names.sort()
print names
```

A1 `['Jake', 'Jerry', 'John', 'Josh']`
A2 `['Jake', 'Jerry', 'Jerry', 'John', 'Josh']`
A3 `['Josh', 'John', 'Jerry', 'Jerry', 'Jake']`
A4 Error message.

12. *Identify the output of the following code!*

```
names = ["John", "Jerry", "Jake", "Josh", "Jerry"]
print names.count("Jerry")
```

A1 `1`
A2 `2`
A3 `(1, 4)`
A4 `(2, 5)`

13. *Identify the output of the following code!*

```
names = ["John", "Jerry", "Jake", "Josh", "Jerry"]
print names.index("Jerry")
```

A1 `1`
A2 `(1, -1)`
A3 `(1, 4)`
A4 `(2, 5)`

14. *Identify the output of the following code!*

```
A = ["John", "Jerry", "Jed"]
B = ['1', '2', '3']
print zip(A, B)
```

A1 `['John', 'Jerry', 'Jed', '1', '2', '3']`
A2 `[('John', 'Jerry', 'Jed'), ('1', '2', '3')]`
A3 `[('1', 'John'), ('2', 'Jerry'), ('3', 'Jed')]`
A4 `[('John', '1'), ('Jerry', '2'), ('Jed', '3')]`

15. *What is the correct way to define a dictionary* `D` *containing the English words "city", "fire", "sun" and their Spanish translations "ciudad", "fuego", and "sol" ?*

A1 `D = {'city': 'ciudad', 'fire': 'fuego', 'sun': 'sol'}`
A2 `D = {'fire': 'fuego', 'sun': 'sol', 'city': 'ciudad'}`
A3 `D = {'sun': 'sol', 'fire': 'fuego', 'city': 'ciudad'}`
A4 `D = {'fire': 'fuego', 'city': 'ciudad', 'sun': 'sol'}`

16. *What is the correct way to add to a dictionary* `D` *new key "school" whose value is "escuela"?*

A1 `D.append('school': 'escuela')`
A2 `D.append_key('school')`
   `D.append_value('escuela')`
A3 `D['school'] = 'escuela'`
A4 `D.add('school': 'escuela')`

17. *What is the correct way to print the value for the key "city" in the dictionary* `D`*?*

```
D = {'fire': 'fuego', 'city': 'ciudad', 'sun': 'sol'}
```

A1 `D.print('city')`
A2 `print D['city']`
A3 `print D('city')`
A4 `print D.get_key('city')`

18. *What is the correct way to ascertain whether or not the key "sun" is present in the dictionary* `D`*?*

```
D = {'fire': 'fuego', 'city': 'ciudad', 'sun': 'sol'}
```

A1 `D.contains('sun')`
A2 `D.key('sun')`
A3 `D.try('sun')`
A4 `D.has_key('sun')`

19. *What is the correct way to print all keys present in the dictionary* `D`*?*

```
D = {'fire': 'fuego', 'city': 'ciudad', 'sun': 'sol'}
```

A1 `print D.get_keys()`
A2 `print D.get_all()`
A3 `print get_keys(D)`
A4 `print D.keys()`

20. *What is the correct way to print all values present in the dictionary* `D`*?*

```
D = {'fire': 'fuego', 'city': 'ciudad', 'sun': 'sol'}
```

A1 `print D.get_values()`
A2 `print D.get_all()`
A3 `print get_values(D)`
A4 `print D.values()`

10 More on Counting Loop

1. *What language element in Karel the Robot is most similar to the* `for` *loop in Python?*

A1 The `while` loop.
A2 The `if - else` statement.
A3 The `repeat` loop.
A4 Recursion.

2. *When should the* `for` *loop be used?*

A1 When we need to go through a list or tuple.
A2 When the number of repetitions is known in advance.
A3 When the number of repetitions is not known in advance.
A4 To replace an infinite `while` loop.

3. *Which of the following codes will print numbers 4, 5, 6, 7, 8, 9?*

A1
```
for i in range(4:9):
    print i
```
A2
```
L = range(4, 10)
for number in L:
    print number
```
A3
```
for val in range(4:10):
    print val
```
A4
```
for i in range(4:10):
print i
```

11 Exceptions

1. *What is an* exception *in programming?*

A1 Sequence of commands that is executed when an `if` condition is not satisfied.
A2 Exceptional situation in the code leading to a crash when not handled.
A3 Sequence of commands that is executed after an error is thrown.
A4 Sequence of commands that is executed when an `if` condition is not satisfied.

2. *Which of the following are exceptions in Python.*

A1 `IndentationError`
A2 `UnboundLocalError`
A3 `TimeoutError`
A4 `OverflowError`

3. *What does* `assert(x != 0)` *do if* `x` *is zero?*

A1 Stops the program.
A2 Raises the `ZeroDivisionError` exception
A3 Raises the `AssertionError` exception.
A4 Prints a warning saying that `x` is zero.

4. *What is the correct way to raise a* `ValueError` *exception when* `x` *is greater than five?*

A1
```
if x > 5:
    ValueError("x should be <= five!")
```
A2
```
if x > 5:
    raise ValueError("x should be <= five!")
```
A3
```
if x > 5:
    exception ValueError("x should be <= five!")
```
A4
```
if x > 5:
    raise_exception ValueError("x should be <= five!")
```

12 Object-Oriented Programming

1. *The philosophy of object-oriented programming is:*

   A1 Operate with geometrical objects.
   A2 Use local variables in functions.
   A3 Use entities that combine functionality and data.
   A4 Use functions that are local in other functions.

2. *What is the relation between* class *and* object *in object-oriented programming?*

   A1 By an object we mean all the data defined in a class.
   A2 By an object we mean all the functions defined in a class.
   A3 Class is an instance (concrete realization) of an object.
   A4 Object is an instance (concrete realization) of a class.

3. *What are* methods *of a class?*

   A1 Specific ways the class is defined.
   A2 Special functions defined in the class that operate with data not owned by the class.
   A3 Specific way variables are arranged in a class.
   A4 Functions that are part of the class definition.

4. *Where are methods defined and where are they used?*

   A1 They are defined in an object and used by a class.
   A2 They are defined outside of a class and used inside of the class.
   A3 They are defined in a class and used by instances of the class.
   A4 They are defined in instances of a class.

5. *Can methods of a class operate with data not owned by the class and when?*

   A1 Yes, always.
   A2 Yes, but only if they also operate with data owned by the class.
   A3 Yes, but only if they do not operate with data owned by the class.
   A4 No.

6. *What is a* constructor*?*

A1 Method of a class that is used to initialize newly created instances.
A2 Special function that is defined in the object, not in the class.
A3 Function that turns a class into an object.
A4 Function that converts an object into a class.

7. *What is the correct way to define a constructor that initializes a variable* `A` *in a class with a value* `a`*?*

A1
```
def __init__(a):
    A = a
```
A2
```
def __init__(self, a):
    A = a
```
A3
```
def __init__(self, a):
    self.A = a
```
A4
```
def __init__(self, self.a):
    self.A = self.a
```

8. *A class contains a variable* `A`*. What is the correct way to define a method* `printdata` *of this class that prints the value of the variable?*

A1
```
def printdata():
    print "A =", A
```
A2
```
def printdata():
    print "A =", self.A
```
A3
```
def printdata(self):
    print "A =", A
```
A4
```
def printdata(self):
    print "A =", self.A
```

9. *Given a text string* `S`*, what is the correct way to count and print the number of appearances of another string* `word` *in* `S`*?*

A1 `print S.count("word")`
A2 `print S.find("word")`
A3 `print S.count(word)`

A4 `print S.parse_string("word")`

10. *What is the way to check whether a string* `S` *is a number?*

A1 `S.isdigit()`
A2 `S.isnumber()`
A3 `isdigit(S)`
A4 `isnumber(S)`

11. *What is the way to replace in a text string* `S` *a string* `s1` *with another string* `s2`*?*

A1 `find_and_replace(S, s1, s2)`
A2 `S.find_and_replace(s1, s2)`
A3 `S.replace("s1", "s2")`
A4 `S.replace(s1, s2)`

12. *How can be in Python a string* `S` *converted to uppercase?*

A1 `S.uppercase()`
A2 `S.raise()`
A3 `S.upper()`
A4 `S.capitalize()`

13 Class Inheritance

1. *When should inheritance be used in object-oriented programming?*

A1 Always because it makes definitions of descendant classes shorter.
A2 When some functionality is common to multiple classes.
A3 When we work with geometrical objects.
A4 We should avoid it as it makes code more complicated.

2. *What is the correct way to define a new class* `B` *which is a descendant of class* `A`*?*

A1 `class B(A):`

A2 `class A(B):`
A3 `class B: public A:`
A4 `class B = descendant(A)`

3. *What is the correct way to call from a descendant class* `B` *the constructor of its parent class* `A`*?*

A1
```
def __init__(self):
      A.__init__(self)
```
A2
```
def __init__(self):
      __init__(A)
```
A3
```
def __init__(self):
      __init__(self, A)
```
A4
```
def __init__(A):
      self.__init__(A)
```