Learning Python for Numerical Analysis

Kenny Erleben

2017

Contents

Chapter 1

Variables, Expressions and Types

1.1 Learning Objectives

In this part of our Python programming learning we will focus on understanding variables, expressions and types.

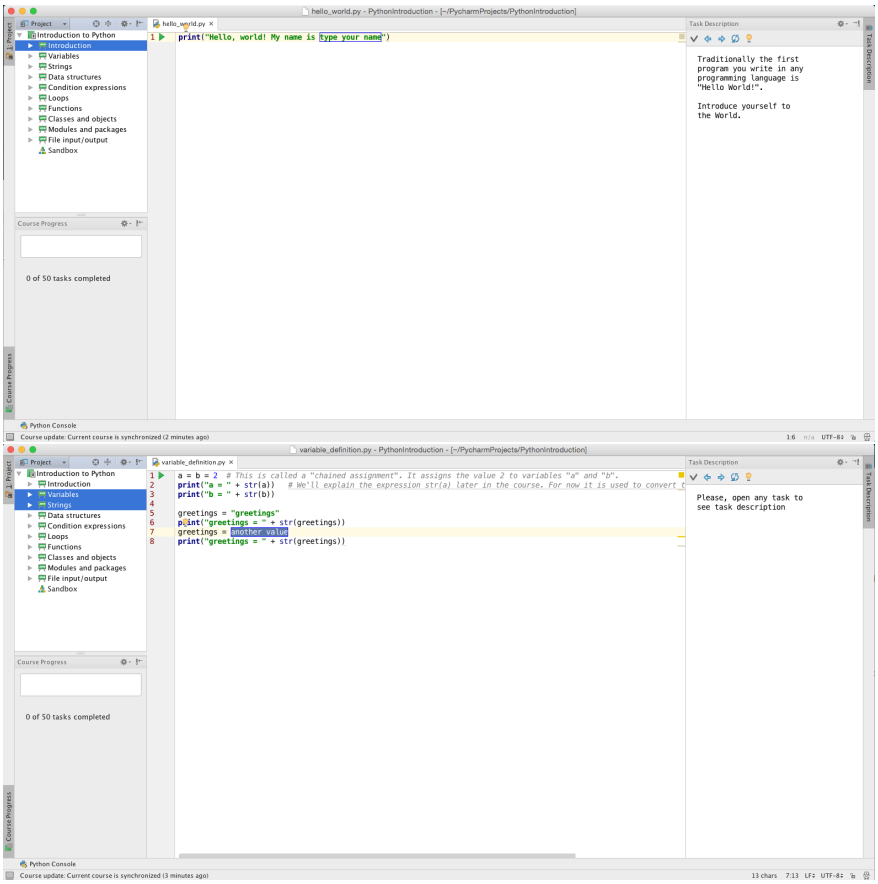
We will start on learning some simple control flow statements and simple usage of functions and modules. Later in next chapter we will go deeper into control flow statements and functions. Here we focus more on getting an introduction to these topics, but not a full coverage.

This part of our learning progress will put emphasis on learning how to debug. You will later in the course experience that this is a skill that needs to be trained through exercise and a task that you as a programmer will often encounter. It is a critical skill to master to ensure that one can pass the programming tests robustly. In any case, for your future career as a "programmer" (at least in this course) you are likely to spend 80% of your time doing debugging.

1.2 PyCharm Tasks

- Do all the tasks in the "Introduction"

- Do all the tasks in "Variables and Strings"


hello_world.py - PythonIntroduction - [~/PycharmProjects/PythonIntroduction]
Introduction to Python
Introduction
Variables
Strings
Data structures
Condition expressions
Loops
Functions
Classes and objects
Modules and packages
File input/output
Sandbox
print("Hello, world! My name is type your name")
Traditionally the first program you write in any programming language is "Hello World!".
Introduce yourself to the World.
0 of 50 tasks completed
Python Console
variable_definition.py - PythonIntroduction - [~/PycharmProjects/PythonIntroduction]
a = b = 2 # This is called a "chained assignment". It assigns the value 2 to variables "a" and "b".
print("a = " + str(a)) # We'll explain the expression str(a) later in the course. For now it is used to convert
print("b = " + str(b))
greetings = "greetings"
print("greetings = " + str(greetings))
greetings = another value
print("greetings = " + str(greetings))
Please, open any task to see task description
0 of 50 tasks completed
Python Console

Chapter 2

Control Flow, Functions, Data Structures and Scope

2.1 Learning Objectives

We will now complete our study of control flow in Python and get a deeper understanding of how to to write functions and what functions should be used for. Higher order functions are a concept that will be particularly important for us when implementing numerical methods. We will use this to pass a function to another function as an argument. This enable us to write more generic code that can solve different problems with the same single code.

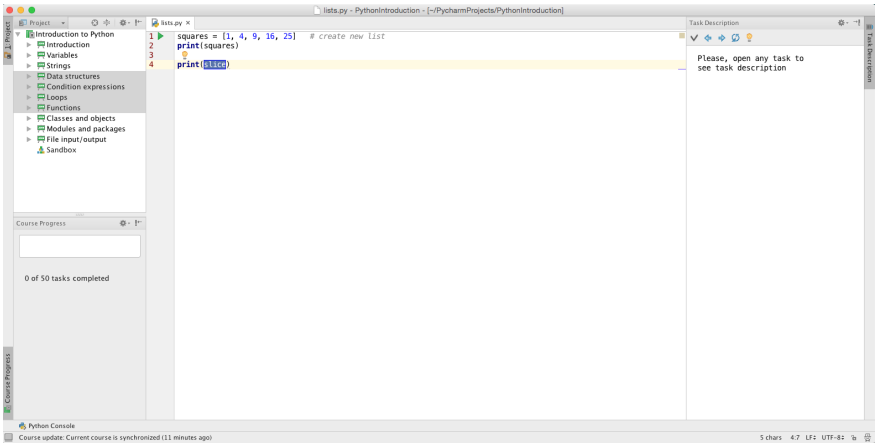
We recommend that students focus on having a deep understanding of the concept of scoping and how this relates to local and global variables and stack frames. In this course on numerical analysis we will hardly ever need to write functions that use anything else than ordinary function arguments. Hence, keyword and default arguments are less important and can be skipped by students not caring to learn Python in depth.

In this part of our Python learning experience we will go deeper into some data structures. In particular it is important to understand what "mutable" and "non-mutable" mean and what a "list" is.

We cover just the basics of what a module is. In this course you will hardly ever notice the difference as most of your hand-ins will be specified in the form of a module that you write. So without knowing it you have already know seen and used modules. Now in this part of the Python programming experience you will obtain the full understanding of what a module is. Modules are very convenient for organizing your code into different files that can be "imported" and used by other Python files (or scripts).

2.2 PyCharm Tasks

- Do all exercises in Data Structures
- Do all exercises in Condition expressions
- Do all exercises in Loops
- Do all exercises in Functions



Chapter 3

How Do I Learn to Create My First Program?

3.1 Introduction

Creating code from math equations can be very difficult and not very easy to approach. Particularly if one is completely new to programming. We will now go over an example of how one can go about this. The idea is to allow students to learn by example during which we will explain a few techniques and common gotchas.

After having read and done the exercises on this page a student should be able to

- Explain that programming is a craft that involves making design decisions
- Describe the idea and process behind prototyping
- Write Python code for computing values of any given number sequence
- Make plots of sequences of numbers
- Format printing of floating point numbers

3.2 Python Language Prerequisite for this Chapter

In this chapter we assume that you:

- Can use the Python commandline shell, or run Python code from your IDE
- Can write for-loops in Python using the range builtin function

- Can write simple floating point expressions
- Can store a floating point value in a variable
- Can create a list and add elements (values) to the list

This is covered in Chapter 1 and 2.

3.3 The Problem We Have to Solve with Programming

Say our task at hand is to create a computer program that can compute and help us analyze the sequence of numbers given by the math equation

$$x_n = \frac{n+1}{n^2}, \quad \forall n \geq 1 \tag{3.1}$$

This is the 'piece' of math equation that we somehow must transform into an algorithm that we can implement using the Python programming language.

This is a different assignment from simply learning how different Python statements work or learning how to use the integrated development environment (IDE) or how to debug one's own code. We need to learn how to create or design programs from scratch. This is like writing a crime book for teenagers, or any other kind of book for any other kind of specific target audience. The basic nature of the problem is the same regardless of which kind of book you would want to write. To begin with you have a rough idea about what you have to do. However, the details are not as of yet clear, you've got no idea of how the story should flow, there might be a lot of research you have to do to first, like learning about guns and poison. At this stage of the process you've only got an empty page, and of course you know your basic tools like how to spell and write proper sentences. The crime book is the analogy to the computer program (encoding the math equation above) which we now want to write.

3.4 Prototyping as a Book Writing Technique

There are several approaches for accomplishing the task of making a program that can help us analyze number sequences and there could even be many different Python programs that could get the job done. Their style might be very different. Even the simple style preferences of the author who wrote the Python program could give the solution a unique aesthetic look and feel. Things like variable names, documentation style, preference of loop statements, or the programming experience/level of the programmer. For instance, a more experience programmer might prefer List comprehensions in Python (Links to an external site.)Links to an external site. over simple loops. This is not unlike the case with books. There are many different books for you to enjoy, but you may prefer the distinctive writing style of say Terry Pratchett (Links to an external site.)Links

to an external site. over other authors. There might be authors whose style and command of language makes no sense to you. Reading program code can be like this too.

To state it shortly we are not guaranteed that there will be a single unique 'right' answer to an assignment such as the example we gave above. This fact can make it hard to learn how to program a computer because there are no single right or wrong answers. It is more like having a lot of different shades to pick from. To make this tangible and feasible to get started we will here present one structured approach that one can use to get started on writing one's own programs. This is our own cook-up of an approach, other teachers may have different ideas or preferences than us. The key to the whole thing is to be structured and methodological in what one does.

As we described previously the first natural step to take when dealing with numerical methods is to get some kind of algorithmic understanding of the math equations. We think of this layer of creating the algorithm as a kind of meta-layer between having the math equations and having a running program written in Python. One has to acquire the competence and skills to create this meta-layer description in order to be able to implement numerical methods. Depending on how familiar the task is to us, and on how many hints and clues we get for creating an algorithm, we can choose different approaches to solving our problem.

If the numerical analysis textbook has already created a kind of meta-level algorithm like a pseudocode description, then that pseudocode could be our starting point for making a Python implementation. If this is not the case then we may have to create our own meta-level representation of an algorithm based on the actual math equations. Fortunately our example is pretty simple and the meta-understanding of how an algorithm should work is intuitively visible already in the way the math is written up. This means we can allow ourselves to skip the step of developing an explicit meta-level description of how an algorithm should work.

Later in the course we will see examples where this meta-level understanding is critical to getting the right solution of the programming problem. We will later learn that when doing this, many design choices are involved, such as decisions about what kind of data structures to use. Whether we prefer while-loops or for-loops, iterative versus recursive solutions etc. The possible combinations of decisions and style flavors can be quite large in number.

Once we have a fair meta-level understanding of the algorithm then we can start the development of an actual prototype of our Python solution. Prototyping a solution in computer science terminology means we will solve the problem by implementing an "answer" for how the Python program should look like. Building a prototype can be a great way to better understand the challenges/issues we need to overcome to be able to solve the problem at hand. The prototype can help us make it more clear what is important, it can help us learn and focus on what is important to solve the task. Prototyping is an iterative process of creating a solution, then evaluating the partial solution, then improving or refine the partial solution, re-evaluating and so and so on. This means

the first prototype program will likely not be the full solution, however, we will modify the program iteratively until we get to a solution.

Step 1: Create an initial prototype - a partial solution to the problem at hand

Step 2: Evaluate if current prototype works as you want

Step 3: If current prototype is not working then improve/refine/fix it to get a new prototype and then go to Step 2 again

Step 4: Stop, your current prototype is working

Prototyping can be very powerful and can allow one to work rapidly. Particular if one has a fair understanding on a coarse level about what should be done. In certain cases one may even "cheat" and take some existing program as the starting point if the task at hand looks very similar to something previously solved. Programmers do not call this "cheating" rather it is often named as using "boiler-plate code" or "templates" for making a prototype.

Back to our illustrating example, the math equation:

$$x_n = \frac{n+1}{n^2}, \quad \forall n \geq 1 \tag{3.2}$$

This equation suggest to us that we will need some kind of for-loop running in the integer nnover the values of interest, and inside the body of this for-loop we should compute the value of x_n for each value that n takes.

This means that moving through values of the integer index n translates into a Python code equivalent piece looking something like this

```
for n in range(infinity):
```

You could not run this code in Python even if you wanted: If you did trytyping it into your Python shell then you would get an error similar to this

```
... Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'infinity' is not defined
```

This is because Python does not recognize the phrase "infinity". Furthermore, on a computer it is impractical to run a for-loop infinitely many times as this will take quite some time to compute. Because infinity is kind of a large number of iterations to make in a computer.

To be practical about it we instead pick a sufficiently large number, whatever value that might be, so the motion through values of the integernn(which we will input into the math equation in a moment) would translate into something like this

```
N = 10
for n in range(N):
```

For purpose of demonstration we picked the value 10 as a sufficiently large number to illustrate what happens when something goes to infinity.

AN IMPORTANT PIECE OF ADVICE Obviously you may rightfully argue that 10 is not a large number to study what happens when a sequence go to infinity. However, it is in our opinion actual very clever to start out using a small number when we are taking this trial and error approach to create our first prototype. Imagine that we have picked the value 10000000 and later made a implementation that would results in a wrong computation inside the for-loop body. Then it will take us considerable time before we might discover that we did something wrong. Hence, it is more practical to start with low numbers, get the workings of the code right, and when the code is working as we want, only then will we increase the value to a larger value as intended.

One must be careful as the code here generates values of n starting at the value zero: $n = 0$. Recall that the sequence we defined was

$$x_n = \frac{n+1}{n^2}, \quad \forall n \geq 1 \tag{3.3}$$

It is clear that the equation above is not meaningful for $n = 0$, converting the equation straightforwardly and naively into Python code yields

```
N = 10
for n in range(N):
    x_n =  (n+1)/n**2
```

If one tries to run this code with $n = 0$ then one gets the run-time error

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ZeroDivisionError: division by zero
```

Go ahead and try it out.

The error we made was that we should be more explicit about the starting index value of the range function, changing the implementation to being explicit about the starting value of n makes the division by zero problem go away.

Based on evaluating this first rough prototype and analyzing the error we can now refine our prototype. Hence the improved prototype code now reads

```
N = 10
for n in range(1,N):
    x_n =  (n+1)/n**2
```

If we execute this code in a Python shell (try doing this yourself) we get the following result

```
>>> N = 10
>>> for n in range(1,N):
...     x_n =  (n+1)/n**2
...
>>>
```

At this point we are very satisfied with ourselves. It seems that we have now created a first working prototype. At this stage we have completed a first whole round of the prototyping idea. Hence it may be fruitful to reflect on what we have learned so far.

What we have learned so far is that a number sequence (indexed by integer values) transforms naturally into a for-loop in a programming language. We have also learned that we must be very careful about the starting end ending values of the running index in the loop, in order to avoid division by zero errors or running "infinitely" many iterations. Observe how we have approached the solution of making a Python prototype. We have iteratively improved and refined our rough starting point. This was done by reevaluating and exploring possible errors and failures in the code and then fixing or patching up all the mistakes or oversights that we made from the beginning. This is essentially the core of the work cycle, build it, run it, evaluate it, fix it and try again if not working. We will learn more techniques for how to evaluate if programs are working. For now our example is sufficiently simple that it suffices with common sense and thinking about run-time error messages from the Python interpreter.

Now that we have what appears to be a working prototype it is about time to evaluate if we like it or not. Looking at the output of the prototype then it is clear to us that this is not very user friendly to us as we can not tell from this output what is going on or if the code even was running. One could try to increase the value of N in the code to see if it would take a longer time to compute before we get a prompt again (try it out on your own). Then we could at least be sure that something was computing. However, it would still be difficult for us to evaluate what goes on. Hence, when evaluating our working prototype it is clear that the program does not help us analyze what goes on with the sequence of numbers.

To get a better output from the Python code we could try printing out the numbers we are computing. That way we can better verify that things look alright and maybe even see if the "pattern" of the numbers converge to some specific value. Adding a print statement to our prototype example it now looks like this

```
N = 10
for n in range(1,N):
    x_n =  (n+1)/n**2
    print(x_n)
```

Running this prototype version creates an output such as this:

```
>>> N = 10
>>> for n in range(1,N):
...     x_n = (n+1)/n**2
...     print(x_n)
...
2.0
0.75
```

```
0.4444444444444444
0.3125
0.24
0.19444444444444445
0.16326530612244897
0.140625
0.12345679012345678
>>>
```

Now we can better observe that a computation of a sequence of numbers is going on and that numbers appear to be getting smaller and smaller. The next problem is that reading the output of the print statement is not very human-readable. At least not in a very human-friendly way. One improvement could be to change the output formatting by controlling how floating point numbers are printed. Try replacing the print statement with something like

```
    print('Iteration ',n, ' x_n = ', format(x_n,'20.15f')
        )
```

Running the example of the modified prototype generates the output

```
>>> N = 10
>>> for n in range(1,N):
...     x_n = (n+1)/n**2
...     print('Iteration ',n, ' x_n = ', format(x_n
   ,'20.15f'))
...
Iteration  1  x_n =     2.000000000000000
Iteration  2  x_n =     0.750000000000000
Iteration  3  x_n =     0.444444444444444
Iteration  4  x_n =     0.312500000000000
Iteration  5  x_n =     0.240000000000000
Iteration  6  x_n =     0.194444444444444
Iteration  7  x_n =     0.163265306122449
Iteration  8  x_n =     0.140625000000000
Iteration  9  x_n =     0.123456790123457
>>>
```

This is much more convenient for quickly observing a pattern in a value as we iterate. However, if we need N to be much larger to make our observation, then the print-approach becomes too cumbersome to work with. Try running the example using a very large N value and then think about if you consider this a practical approach for analyzing the sequence?

Another approach could be to plot the numbers for, as the saying goes, one picture is worth a thousand words. We will use a python package that will easily help us draw things. One has too add the "import" statement at the beginning of the program, as follows:

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

This will give us the drawing capabilities that we will be using. Next we must change our example program because we need to remember all the numbers we compute, not just the last computed x_n value. For this purpose we will use a list data structure. We create an empty list by writing

```
values = []
```

To add an element to the list we write

```
values.append( x_n)
```

We can now write up the modified code example of our prototype for computing the values

```
import matplotlib.pyplot as plt


N = 10
values = []
for n in range(1,N):
    values.append(   (n+1)/n**2   )
```

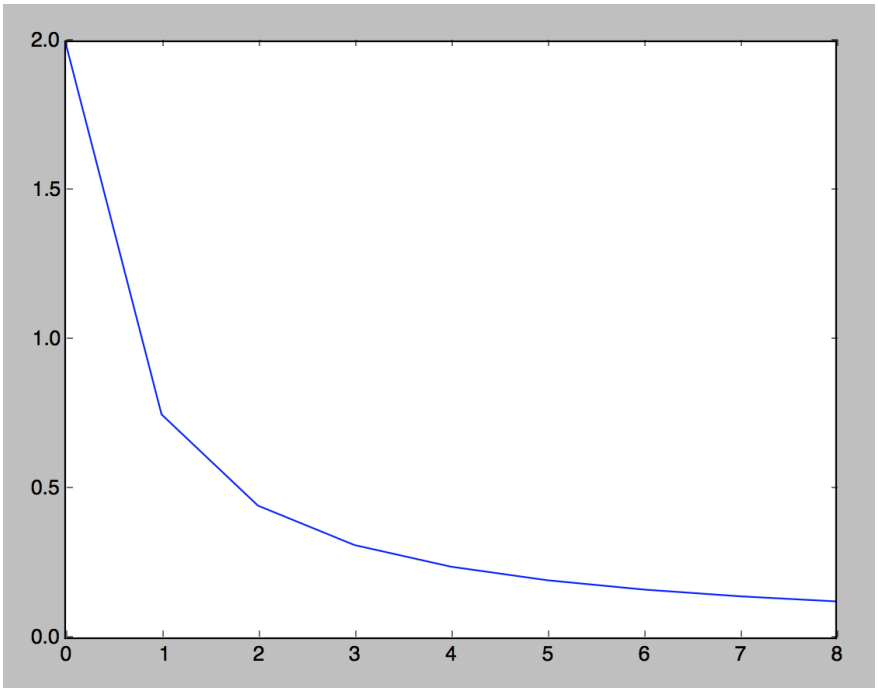
Next we need to add more code that will create the actual picture. This is done like this

```
plt.plot(values)
plt.show()
```

The complete code now reads

```
import matplotlib.pyplot as plt


N = 10
values = []
for n in range(1,N):
    values.append(   (n+1)/n**2   )
plt.plot(values)
plt.show()
```

Try running this code. You should get a plot shown that looks a little like the one shown below

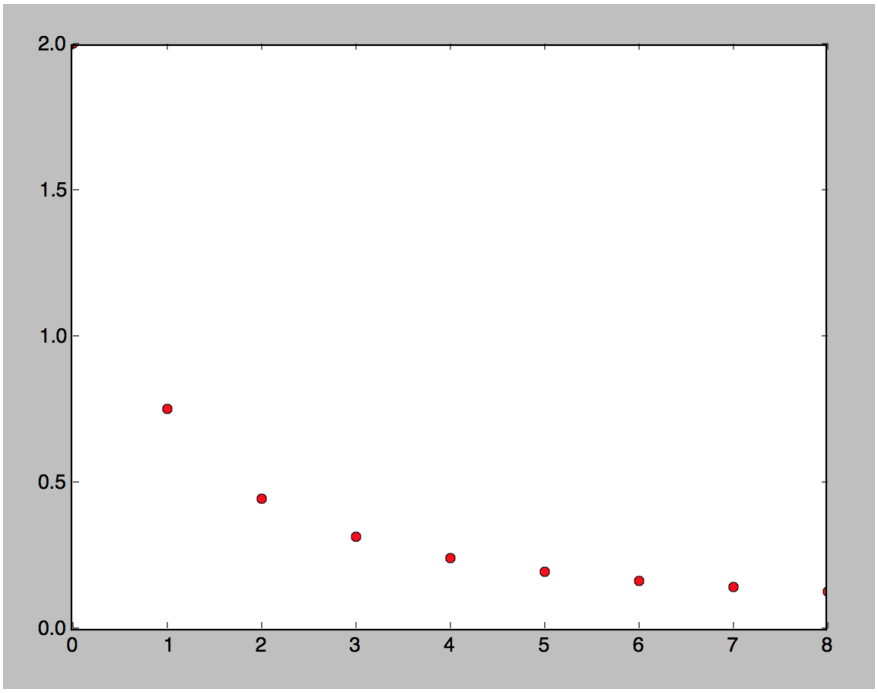


Now one can more easily choose a large number and simply by visual inspection get an idea of whether the sequence of numbers are converging toward something or not.

We may tune the layout of the plot to be slightly "correct", that is to say more representative of what is going on: The current plot shows the values as data points connected by straight blue lines. However, we clearly do not have any values between our data points, so the straight lines are in fact a little misleading. Instead we would rather just draw our values as points and maybe we like the color red better than blue. Let us add a formating string to the plot-statement, like this

```
plt.plot(values,'ro')
```

When rerunning the prototype we now obtain the plot shown below



There are many options to explore for formatting ones plot. However, for now we only need very simple functionality. Try out other options such as

```
plt.plot(values,'b-')
plt.plot(values,'gx')
plt.plot(values,'m-')
plt.plot(values,'r-.x')
```

You can play with more options if you wish or try and look up the documentation for the format string on the internet. Go to this home page

`http://matplotlib.org/api/index.html`

Here one can find details on all the functionality in the Matplotlib package for Python. Now try locating the pyplot.plot part of the documentation:

`http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot`

Here you can read details about string formatting for your output.

ANOTHER IMPORTANT PIECE OF ADVICE The internet is a valuable place to find documentation about how different functions or arguments should be used. Sometimes one may also find information about how to fix certain types of errors. It is not cheating to search the web for this purpose. In fact sometimes it is the fastest way to figure out what one is doing wrong, and one can learn a lot from it.

For now our prototype Python program for analyzing the convergence of a sequence of numbers is given by

```
import matplotlib.pyplot as plt


N = 100
values = []
for n in range(1,N):
    values.append(   (n+1)/n**2   )
plt.plot(values,'ro')
plt.show()
```

Let us now reflect on what we have learned from this example. Through very rapid prototyping we improved our solution, by using "print" for outputting values to us, then we improved the formatting of the numbers we printed, then we re-evaluated, we shifted to use plotting, and finally we tuned the plotting layout to be more consistent with looking at sequences of numbers.

In summary we have now seen and experienced the benefit of using rapid prototyping to create a solution for our problem.

3.5 Where to Go Next?

At this stage it is a good idea to practice the idea of prototyping a bit more before moving on. For this we created some exercises for you to have fun with.

- Try to use prototyping for improving our example prototype above even more, by adding a title and axes to the plot, or other features that makes the plot look even better.

- The purpose of this exercise was to illustrate that different design choices lead to different programs. Use prototyping for changing the prototype above such that it prints out the last 10 numbers of the list before showing the plot. Be careful to analyze issues (use your common sense) that you may encounter and think about how to catch and fix them. Here are some hints:

  - What if the list does not contain 10 elements?
  - Should one first compute all the numbers in one for-loop and then use another for-loop to print out the numbers? (Hint: write two sequential for-loops)
  - Should one print out the numbers inside the for-loop while computing them? (Hint: think of using an if-statement).

- How would you optimize your code to use list comprehensions instead of for-loops?

- Try writing a Python program to analyze the sequence equation

$$x_n = \left(1 + \frac{1}{n}\right)^n, \quad \forall n \geq 1 \tag{3.4}$$

- Try writing a Python program to analyze the sequence defined by

$$x_0 = 0.9 \tag{3.5}$$

$$x_n = x_{n-1}^2, \forall n \geq 1. \tag{3.6}$$

Chapter 4

Matrices and Recursion

4.1 Learning Objectives

By this time we master the basic subset of the Python programming language and we are familiar with our integrated development environment (IDE) and the interactive Python shell. Hence, we can move forward and polish our understanding of recursive functions, learn to improve and control the input and output of our own programs, and study more advanced data structures such as matrices and advanced plotting examples.

Matrices are essential when we study numerical methods for solving linear systems and a good knowledge of how to manipulate numbers stored in matrices is very important to solve these exercises. Recursive functions allow us to write very short, concise and readable code with an uncanny similarity to some of the mathematical counterparts of stating the algorithms which we will implement. Mastering the formatting of input and output is vital to making it easy to read and interpret the numerical solutions that your code is computing.

4.2 PyCharm Tasks

Feel free to play with remaining PyCharm tasks as much as you like. Many of them go beyond the learning objectives, but these are all good exercises for getting to know Python better.

Chapter 5

How Can I Be Clever About Making My Second Program?

5.1 Introduction

We just learned about prototyping in Chapter 3. We used this technique to create our very first Python program. This final program could help us quickly analyze a sequence of numbers by visually inspecting a plot of a finite range of the values. In this subsequent chapter we will demonstrate how to quickly adapt existing code when doing prototyping. As we did previously in the last chapter we will study this work process by walking through an example and prototype develop a Python program as we go along.

When having read this chapter a student should be able to

- Use boiler-plate code for prototyping simple problems such as analyzing a sequence of numbers
- Describe some of the advantages and pitfalls of using boiler-plate code
- Identify indexing errors caused by mismatch in math notation and data structure conventions
- Explain and apply approaches to fixing indexing errors
- Use prototyping for learning about orders of magnitude definitions
-

5.2 Python Language Prerequisites

In this chapter we assume that you

- Can use the Python (command line) shell, or know how to run Python code from your IDE

- Can write for-loops in Python using the built-in "range" function
- Can write simple floating point expressions
- Can store a floating point value in a variable
- Can create a list and add elements (values) to the list
- Can start the visual debugger in your IDE

This is covered in Chapter 1 and 2.

5.3 The Second Task

This time our assignment is to create a program for analyzing the sequence of numbers given by the equation

$$x_1 = 2 \tag{5.1}$$

$$x_{n+1} = \frac{1}{2}x_n + \frac{1}{x_n}, \quad \forall n \geq 1\,. \tag{5.2}$$

We have already seen sequence definitions before, and in the last chapter we learned how to systematically write code for computing sequence values. Hence it appears that our road to success is rather straightforward.

5.4 Using Boiler-Plate Code

As true academics, and being smart about it, it admittedly seems wasteful of time to be too straightforward about prototyping a solution for our second task using the exact same starting point as before. The task at hand now seems so similar to what we solved previously, so could we not just "borrow" from what we already have done? Thereby saving some time developing everything from scratch. The answer is that yes we can. Hence, we will take our last most recent prototype as our starting point. That is

```
import matplotlib.pyplot as plt


N = 100
values = []
for n in range(1,N):
    values.append(   (n+1)/n**2  )
plt.plot(values,'ro')
plt.show()
```

Clearly the code does not compute the sequence given above. However, we do have the basic functionality of setting up a for-loop for computing and then

storing the values in a list, and also for plotting the numbers we computed. Starting with this code it appears that all we need to do is to change the code in line 7

```
    values.append(    (n+1)/n**2   )
```

to reflect the math equation defining the new sequence that we wish to analyze. Looking at the equation

$$x_1 = 2 \tag{5.3}$$

$$x_{n+1} = \frac{1}{2}x_n + \frac{1}{x_n}, \quad \forall n \geq 1\,. \tag{5.4}$$

We immediately get inspired to make the following changes to the code

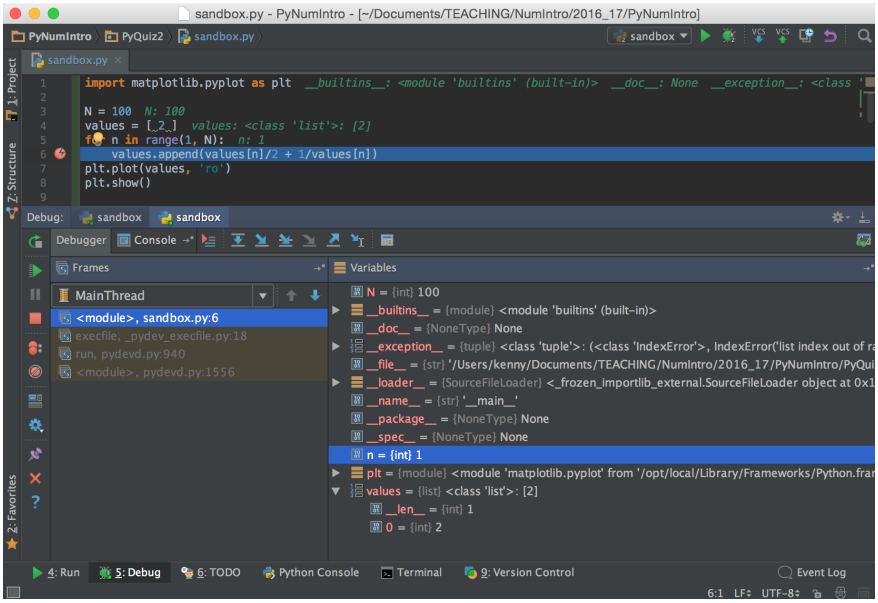
```
import matplotlib.pyplot as plt


N = 100
values = [ 2 ]
for n in range(1,N):
    values.append( values[n]/2 + 1/values[n]  )
plt.plot(values,'ro')
plt.show()
```

Observe carefully that here we have initialized the empty list to contain the value 2 and replaced the floating point expression with a new expression corresponding to the new sequence equation.

Now it is tempting to run the code to see if our new prototype works. Running it we get the result

```
  Traceback (most recent call last):
 File "...", line 7, in <module>
 values.append( values[n]/2 + 1/values[n] )
IndexError: list index out of range
```

This was quite unexpected. The code does not work while it does appear that we have implemented the correct recursive relation. What is going on? Looking carefully we notice that Python tells us that we have an index error when looking up the values stored in the list at index n. How can this be? To analyze further what went wrong we need to debug. So let us debug our current prototype. The built-in tool called the debugger will halt execution of the program for us when the error is encountered and we can then inspect the values of all the variables. Here is a screen dump of how my PyCharm debugger looks like when the error occurs



There is a lot of extra information here, but the central points are that to get to this screen:

- One simply wrote the code into a Python file, in the example this was named “sandbox.py”, but it could be a scratch file or any other name. It is not important.
- After this one opens the file in PyCharm, if it is not already open
- Then one right-clicks on the code window and chooses the option "Debug “sandbox”
- After a split second the debugger will halt showing a window similar to the one shown above.

We observe that when the crash occurred, our index n had the value $n = 1$ and the list named “values” stored one element with value 2. We also notice that the index of the element in the list is 0. From this we have enough clues to deduce what went wrong in this version of the prototype. The mathematical equation defining the sequence

$$x_1 = 2 \tag{5.5}$$

$$x_{n+1} = \frac{1}{2}x_n + \frac{1}{x_n}, \quad \forall n \geq 1\,, \tag{5.6}$$

starts with indices running from one, but the list data structure in Python uses indices starting at value zero(!).

ALTERNATIVE TO THE VISUAL DEBUGGER If one does not have a fancy IDE with a visual debugger, such as in the above example, then it

can be more difficult to analyze what went wrong. In our example we could have added more print statements to the prototype to obtain more output that could have helped us. For instance we could have added a print statement before the program line where we encountered the index error.

```
import matplotlib.pyplot as plt

N = 100
values = [ 2 ]
for n in range(1, N):
  print(n, values)
  values.append(values[n]/2 + 1/values[n])
plt.plot(values, 'ro')
plt.show()
```

The output from this prototype with debug-printing would look like this

```
Traceback (most recent call last):
1 [2]
 File "/Users/kenny/Documents/TEACHING/NumIntro/2016_17/
    PyNumIntro/PyQuiz2/sandbox.py", line 7, in <module>
 values.append(values[n]/2 + 1/values[n])
IndexError: list index out of range
```

From which we can conclude that the list named "values" contains one element with value 2, and that n has the value 1. It is however not as straightforward to see that the indexing of the list starts with the value 0 and not 1.

One way out of our misery is to rewrite or re-derive the math equations to use indexing consistent with our data structures. The recursive sequence equation is easily rewritten to start with the index 0 instead:

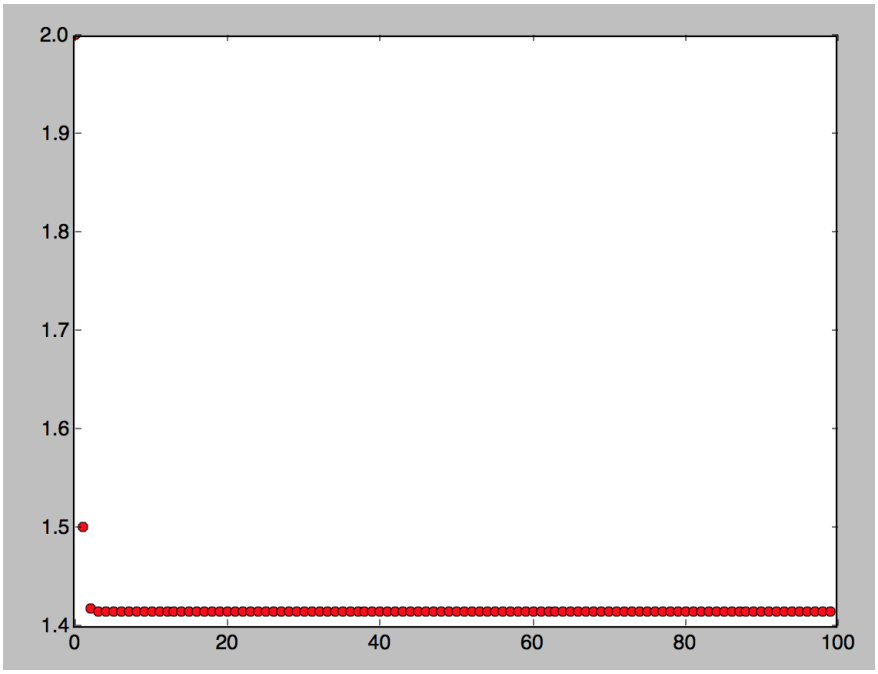
$$x_0 = 2 \tag{5.7}$$

$$x_n = \frac{1}{2}x_{n-1} + \frac{1}{x_{n-1}}, \quad \forall n \geq 1\,. \tag{5.8}$$

From this we may now modify the prototype code to yield

```
import matplotlib.pyplot as plt

N = 100
values = [ 2 ]
for n in range(1, N):
    values.append(values[n-1]/2 + 1/values[n-1])
plt.plot(values, 'ro')
plt.show()
```

When running the prototype we obtain the output plot shown below.



This plot shows a sequence that seems to rapidly converge to a value slightly larger than 1.4, as we would expect. Having obtained a successful prototype now is a good time to reflect on what we have learned.

- Jump-starting prototyping with boiler-plate code allows for more rapid development. In the example only two lines of code was needed to be modified.

- Index errors quickly arise if the math equations one builds upon does not adhere to the same type of indexing convention used by one's programming language.

Remarks on boiler-plate code We have presented boiler-plate coding as a technique for getting a rapid process and fast working prototype. There is more to be said to this. The boiler-plate code could be more complex than the simple example we presented here. It could involve many tests or much more complicated data structures. Surely one could imagine the headache from having to write a lot of code from scratch and debug it to get it to work. Boiler-plating can hence save debugging time too. Unfortunately, code is no better than the person who wrote it, so one can also inherit a lot of mistakes or bugs with boiler-plate code. Basically, it boils down to that it is not going to save you time and resources if you get faulty code to start with.

Remarks on index errors Let us briefly return to discuss index errors some more. In our experience such errors are very often encountered when implementing numerical methods. Particularly given that authors from different

textbooks have different preferences whether to use the index value 0 or 1 as their starting index values. Many programming languages tend to use 0 at the starting index, however other conventions are seen too. In our example the math was sufficiently simple to allow for an easy re-write hence we opted for going back to the actual math equation to fix the problem. This approach for getting out of the misery is not always viable. Perhaps the math derivation is too lengthy or time-consuming to do, or maybe too complex to follow, or maybe some steps are not sufficiently elaborated, we may even only have a pseudo-code description of the method, but not the actual math equations for deriving the method. In such cases one can get around fixing the indexing in the actual code using straightforward index conversion on the fly. For our simple example this approach will clutter the code and likely not be very efficient compared to what we did. However, the simplicity of the example allow us to capture and show the idea. Here we present a different prototype code that solves the indexing problem in code

```
import matplotlib.pyplot as plt

N = 100
values = [ 2 ]
for n1base in range(1, N):
    n0base = n1base - 1
    values.append(values[n0base]/2 + 1/values[n0base])
plt.plot(values, 'ro')
plt.show()
```

Notice that we simply created an auxiliary variable `n0base` that converted the one-based index value `n1base` to the zero-based index value that we needed for our list data structure (the structure named `values`). The example illustrates that we can implement the math with whatever indexing style and then add "index" conversions on the fly.

On the fly index conversions can be very dangerous if one is not very careful to make sure to have them in all the right places. For large complex numerical code it can be a headache to add index-conversions and one can easily make a mistake when writing the code. A mistake that can be quite hard to track down with a debugger.

5.5 What's Up Next, Doc?

Now is a good time to practice the boiler-plate coding "cheat" technique some more

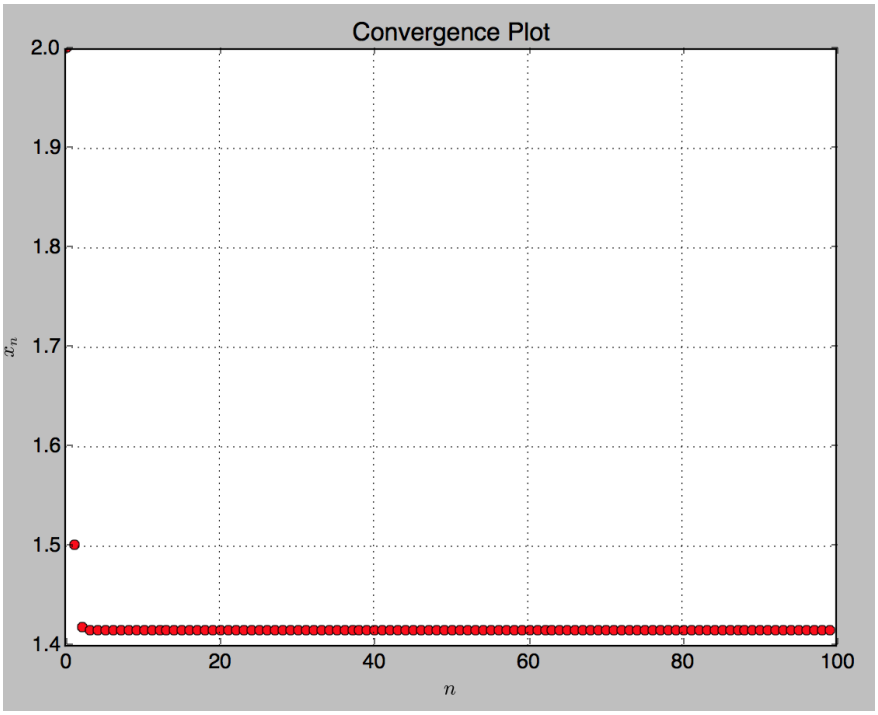
- Analyze if the sequence $x_n = \frac{1}{n \ln(n)}$ converges
- Analyze if the sequence $x_n = \frac{5}{n} + e^{-n}$ converges

5.6 Improving the Prototype Further

Having used the prototype now for several examples we have gained the insight that we want to have titles, axes and grid lines for making it easier to read the plots and to later remember what was what. Hence we now add these extra features to the current prototype

```
import matplotlib.pyplot as plt

N = 100
values = [ 2 ]
for n in range(1, N):
    values.append(values[n-1]/2 + 1/values[n-1])
plt.figure(1)
plt.plot(values, 'ro')
plt.title('Convergence Plot')
plt.xlabel('$n$')
plt.ylabel('$x_n$')
plt.grid(True)
plt.show()
```

Now the plotting looks more nice as shown below



Reflecting more on this plot it becomes clear to us that the rapid drop-off in values makes it difficult for us to visually inspect whether the sequence numbers

are converging to a number close to 1.4 or merely oscillating around this value? Changing the scaling of the y-axis can help us better visualize small changes in values. We can change the scaling of the plotting by replacing our current plotting with the statement

```
plt.semilogy(values, 'ro')
```

Or

```
plt.loglog(values, 'ro')
```

Try out how this works for the different sequences you have analyzed this far.

5.7 Using Programming for Learning about Orders of Magnitude

Armed with our new knowledge of plotting features in Python, it may be interesting to study the convergence behavior of the actual definitions of "orders of convergence". We will start by recapitulate the mathematically rigorous definitions of the most common convergence rates.

5.7.1 Linear Convergence Rate

This means that there exists a positive integer N such that when n is larger than N we have the relation

$$|x_n - x^*| \leq c\,|x_{n-1} - x^*|\,, \quad 0 < c < 1 \wedge \forall n > N \tag{5.9}$$

The constant c is termed the constant of convergence and x^* is the limit point of the sequence $\{x_n\}$. One may define the absolute error as $\varepsilon_n = |x_n - x^*|$. Using this notation the governing equation of the definition can be written compactly as

$$\varepsilon_n \leq c\,\varepsilon_{n-1} \tag{5.10}$$

5.7.2 Superlinear Convergence Rate

This definition looks very similar to the linear convergence. The only exception is that the constant of convergence is replaced by the sequence $\{c_n\}$ where $c_n \to 0$ for $n \to \infty$. Hence there exist N such that when $n > N$ we have

$$\varepsilon_n \leq c_n\,\varepsilon_{n-1} \tag{5.11}$$

5.7.3 Quadratic Convergence Rate

This can be summarized as

$$\varepsilon_n \leq C\,\varepsilon_{n-1}^2 \tag{5.12}$$

where the constant of convergence can be any positive number $C > 0$. Notice the power 2 is the reason for the name quadratic convergence, one may generalize the concept to higher orders

$$\varepsilon_n \leq C\,\varepsilon_{n-1}^{p} \tag{5.13}$$

Where p is any positive integer such that $p > 1$. The specific value of $p = 3$ we name cubic convergence rate.

5.8 The Programming Task at Hand

We will use the definitions of rate of convergence to define exemplary sequences. We do this by replacing the the less-than equal relations with equality, then we choose an initial error value etc. Mathematically we have the sequences

$$\varepsilon_n = c\,\varepsilon_{n-1}\,, \quad c = \tfrac{1}{2} \wedge \varepsilon_0 = \tfrac{9}{10} \wedge n > 0\,. \tag{5.14}$$

$$\varepsilon_n = c_n\,\varepsilon_{n-1}, \quad c_0 = 2 \quad \wedge \quad c_n = \tfrac{9}{10}\,c_{n-1} \quad \wedge \quad \varepsilon_0 = \tfrac{9}{10} \quad \wedge \quad n > 0\,. \tag{5.15}$$

$$\varepsilon_n = C\,\varepsilon_{n-1}^{2}\,, \quad C = 2 \quad \wedge \quad \varepsilon_0 = \tfrac{4}{10} \quad \wedge \quad n > 0\,. \tag{5.16}$$

$$\varepsilon_n = C\,\varepsilon_{n-1}^{3}\,, \quad C = 2 \quad \wedge \quad \varepsilon_0 = \tfrac{4}{10} \quad \wedge \quad n > 0\,. \tag{5.17}$$

Combined these four sequences represent the archetype behavior of linear, super linear, quadratic and cubic convergence behavior. It can be instructive and help provide us intuition about these rates of convergence if we analyze them using programming. Hence your task now is to take the current prototype, use this as a boiler plate code for creating one program that

- Can compute the numbers of all sequences simultaneously (Meaning: You click "run" once).
- Make one figure containing all four plots simultaneously
- Make a second figure showing the same plots using a log-scale
- Make a third figure showing the same plots using a log-log-scale

Here follow some hints that you may choose to use for creating your prototype

- Create four lists to contain the values of the sequences
- Create variables for all the constants of convergence
- When adding multiple plots to one figure then one needs a few more statements to control the drawing, study the "pyplot" API (application program interface) functions "figure", "hold", and "legend"

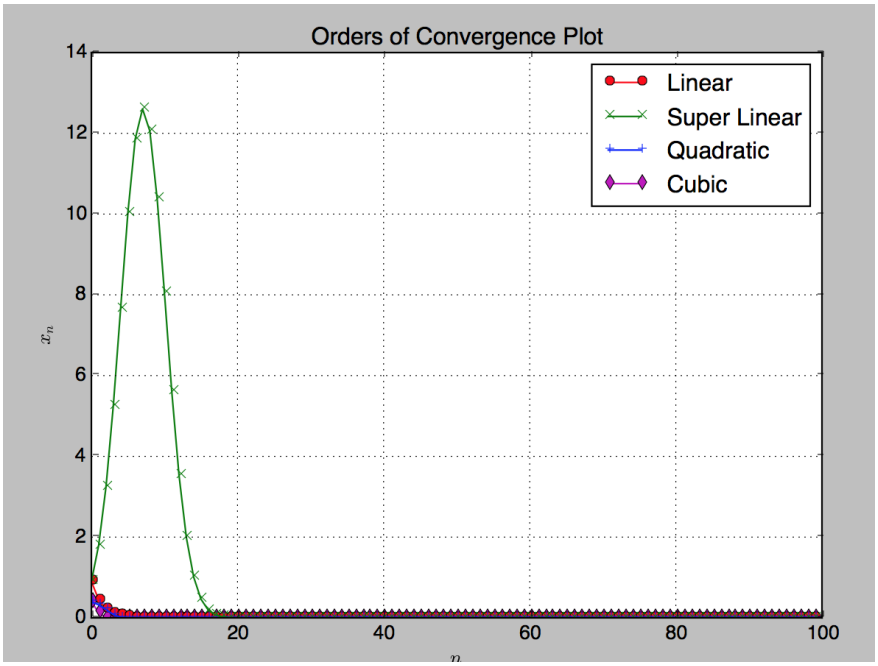
Here is some boiler-plate code for creating a figure with multiple plots in it

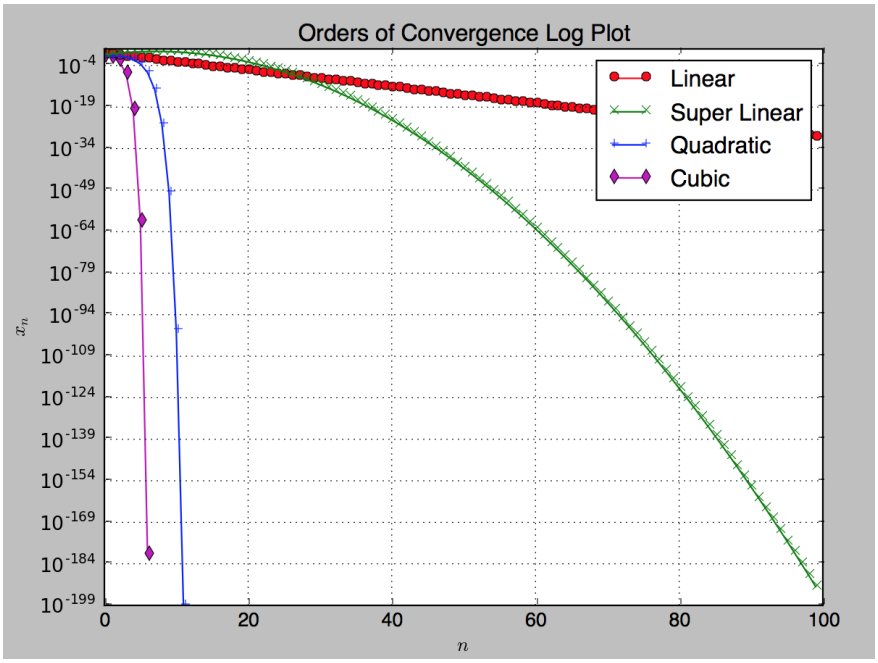
```
plt.figure(1)
p0, = plt.plot(values1, format1,label = 'Plot 1')  #
    values1 is some list, format is some string
plt.hold(True)
p1, = plt.plot(values2, format2, label='Plot 2')
# Add more plots here as you desire
pn, = plt.plot(valuesn, formatn,label='Plot 3')
plt.legend(handles = [p0, p1 , ... , pn])  # This is not
    value syntax '...' must be replaced
plt.hold(False)
plt.show()
```

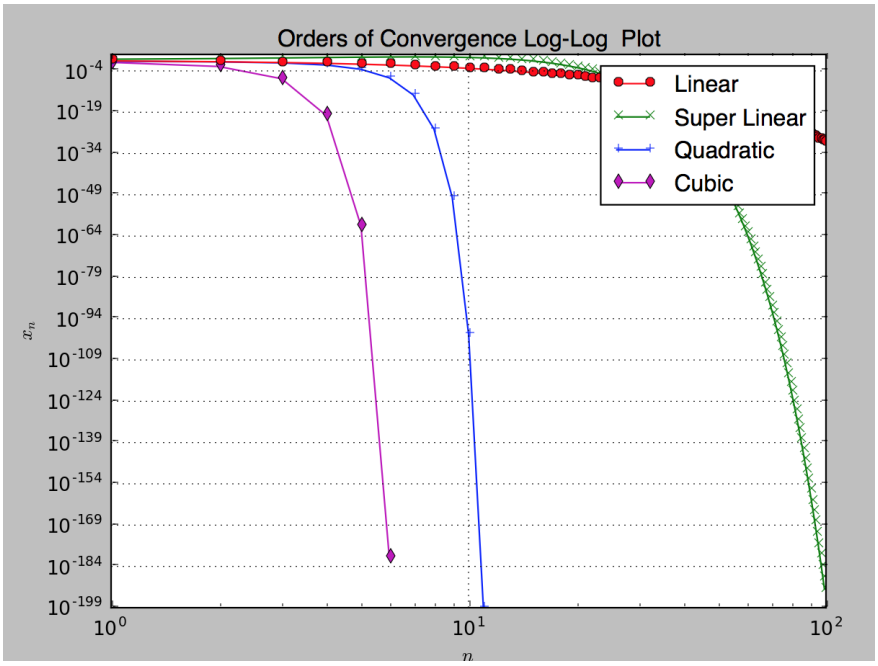
The figure command opens a window with the given figure number. The first hold statement must be done immediately after the very first plot and have the argument True. The second hold statement must be right before the show statement and have argument value False. The legend statements take a keyword parameter that is a list of all the plot handles (i.e. the return value from the plot functions).

5.9 The results

If you have solved the programming task correctly then you should see three output plots as shown below







Observe how the log and log-log scale plots are more easy to analyze. Particularly observe that a linear convergence rate in the log-plot shows up as a straight line, whereas in the first non-log scale plot it is impossible to clearly

see the difference between the plots, particularly when n gets larger than 20. The log-scale plots can help quickly get a hint about superlinear and p-order convergence rates. Notice how the spacings of the data points are different for the different types of convergence order.

Having intuition about how the plot shape of order of convergence definitions allows one to quickly experimentally verify if convergence theorems for various numerical methods are correct for one's own implementations. This is a fast and quick way to help verify if one's implementation works as expected.

5.10 Teacher's Code Solution

Below follows the prototype solution made by the teachers. You can use this for comparing with your own solution or if you do not care about learning to program you can cheat and copy-paste this code and play with it.

```
import matplotlib.pyplot as plt

N = 100
values0 = [ 0.9 ]  # Linear Convergence
values1 = [ 0.9 ]  # Super linear Convergence
values2 = [ 0.4 ]  # Quadratic Convergence
values3 = [ 0.4 ]  # Cubic Convergence
c = 0.5
c_n = 2.0
C = 2.0
for n in range(1, N):
    values0.append(c * values0[n-1])
    values1.append(c_n * values1[n - 1])
    c_n *= 0.9
    values2.append(C * (values2[n - 1]**2))
    values3.append(C * (values3[n - 1]**3))

plt.figure(1)
p3, = plt.plot(values3, 'm-d',label = 'Cubic')
plt.hold(True)
p0, = plt.plot(values0, 'r-o', label='Linear')
p2, = plt.plot(values2, 'b-+',label='Quadratic')
p1, = plt.plot(values1, 'g-x',label='Super Linear')
plt.legend(handles = [p0, p1 , p2 , p3])
plt.title('Orders of Convergence Plot')
plt.xlabel('$n$')
plt.ylabel('$x_n$')
plt.grid(True)
plt.hold(False)
plt.show()
```

```
plt.figure(2)
p3, = plt.semilogy(values3, 'm-d',label = 'Cubic')
plt.hold(True)
p0, = plt.semilogy(values0, 'r-o', label='Linear')
p2, = plt.semilogy(values2, 'b-+',label='Quadratic')
p1, = plt.semilogy(values1, 'g-x',label='Super Linear')
plt.legend(handles = [p0, p1 , p2 , p3])
plt.title('Orders of Convergence Log Plot')
plt.xlabel('$n$')
plt.ylabel('$x_n$')
plt.grid(True)
plt.hold(False)
plt.show()

plt.figure(3)
p3, = plt.loglog(values3, 'm-d',label = 'Cubic')
plt.hold(True)
p2, = plt.loglog(values2, 'b-+',label='Quadratic')
p1, = plt.loglog(values1, 'g-x',label='Super Linear')
p0, = plt.loglog(values0, 'r-o', label='Linear')
plt.legend(handles = [p0, p1 , p2 , p3])
plt.title('Orders of Convergence Log-Log  Plot')
plt.xlabel('$n$')
plt.ylabel('$x_n$')
plt.grid(True)
plt.hold(False)
plt.show()
```

If you feel like a daring person then try playing around with changing the values of the constants in the code and the initial values for the sequences. One fun experiment is to change the initial value of quadratic and cubic rate of convergence to a larger number such as the value 0.9.

Chapter 6

Visual Debugger Training

6.1 Introduction

In this chapter we have designed a series of small simple Python code examples. All code examples suffer from some kind of wrong doing or peculiarity. It is up to the student to determine which is which and possibly improve or fix the examples. To be specific the code examples may contain syntax errors so that they will not compile properly, or they suffer from run-time errors, or do not quite do the right thing even though they run just fine, or the examples try to illustrate some peculiarity of using Python for programming. It is up to the student to analyze the code using the visual debugger built into the integrated development environment (IDE) and determine and discuss his/her findings in class with the other students and teaching assistants. The expected learning outcome is that a student

- is sufficiently proficient in using the visual debugger to analyze possibly faulty code

6.2 Python Prerequisites

A student should be able to

- Use variables and expressions
- Use simple for-loops
- Use print-statement
- Describe the limitations of floating point representation
- Open IDE, write code samples and start visual debugger

6.3 Example 1 - Even Numbers

```
'''

Problem statement: Given a postive integer x list all
even numbers from zero and up to and including x

'''
x = 10
for i in range(x):
    if (i%2) == 0:
        print( "found even number = ", i)
```

6.4 Example 2 - Expression

```
'''

 Example of assigning an integer expression to a variable

'''

myVar = 5
b = (((myvar*2)+3)*2)/3
```

6.5 Example 3 - Float Division

```
'''

Example of using float values. A float number is not the
    same as a real number. A float has finite precision.
This means that a real number often will be 'truncated'
    or 'rounded' when used in computations or displayed.

This example helps illustrate some of the quirks when
    working with floating point values.


If we take a small number and continously try to make it
    even smaller then we will get to zero

'''
a = 1.97626258336e-323
```

```
print( a )
for x in range(10):
    a = a / 2
    print( a )
```

6.6 Example 4 - Float Multiplication

```
'''

  Example of using float values. A float number is not
      the same as a
  real number. A float only has finite precision.
  This means that a real number will often be 'rounded'
      when used
  in computations or displayed.

  '''
a = 1.97626258336e+307
print( a )
for x in range(10):
  a = a * 2
  print( a )
```

6.7 Example 5 - Machine Precision

```
'''

Example of using float values. A float number is not the
    same as a real number. A float has finite precision.
This means that a real number often will be 'truncated'
    or 'rounded' when used in computations or displayed.

This example helps illustrate some of the quirks when
    working with floating point values.


How small a floating point number can we add to the value
     1.0 and still get the value 1.0?

Example is taken from https://en.wikipedia.org/wiki/
    Machine_epsilon
```

```
'''

epsilon = 1.0

while (1.0 + 0.5 * epsilon) != 1.0:
    epsilon = 0.5 * epsilon

print("Approximated machine epsilon =", epsilon)
```

6.8 Example 6 - Rounding

```
'''

Example of using float values. A float number is not the
    same as a real number. A float has finite precision.
This means that a real number often will be 'truncated'
    or 'rounded' when used in computations or displayed.

This example helps illustrate some of the quirks when
    working with floating point values.

'''

x = 0.0
for i in range(10):
    x = x + 0.1
if x == 1.0:
    print( x, ' = 1.0')
else:
    print( x, ' <> 1.0')
```

6.9 Example 7 - Integer Root Version 1

```
'''

 Problem statement: given positive integer x find largest
     positive integer z such that z*z <= x.
 We call z the "integer" root of x.

'''
x = 39
for z in range(x):
```

```
    y = z*z
    if y > x:
        print ("Integer root of", x, "is", z-1)
```

6.10 Example 8 - Integer Root Version 2

```
'''

 Problem statement: given positive integer x find largest
     positive integer z such that z*z <= x.
 We call z the "integer" root of x.

'''
x = 39
for z in range(x):
    y = z*z
    if y > x:
        print ("Integer root of", x, "is", z-1)
    break
```

6.11 Example 9 - Square by Adding Version 1

```
'''

 Problem statement: given positive integer a compute the
     value of the square of a using only the addition
     operator

'''


a = 5
b = 1
for i in range(a):
    b = b + a
print ("The square of", a, "is", b)
```

6.12 Example 10 - Square by Addition Version 2

```
'''
```

```
 Problem statement: given positive integer a compute the
     value of the square of a using only the addition
     operator

'''

a = 5
b = 0
for i in range(a)
    b = b + a
print("The square of", a, "is", b)
```

6.13 Example 11 - Square by Adding Version 3

```
'''

 Problem statement: given positive integer a compute the
     value of the square of a using only the addition
     operator

'''

a = 5
b = 0
for i in range(a):
    b = b + a
print ("The square of, a, "is", b)
```

Chapter 7

Using Pseudocode

7.1 Introduction

In past chapters we developed prototyping as a technique for rapidly developing a Python program. We refined our work technique by showing how to jump-start prototyping using boiler-plate code, and we learned through examples how to use a visual debugger to improve on our prototype. Mastering this work process is a very powerful tool to rapidly develop a program. However, so far we have used simple and similar problems for our learning, namely problems of analyzing sequences of numbers. A common trait we can observe by now is that all these sequences are easily "converted" directly from a defining math equation into a corresponding Python code.

There are other examples of such math symbols that one may easily deal with without having to create a complex meta-level description of an algorithm before implementing a solution. In this chapter we will survey some often and common encountered mathematical constructs: summation, products, and piecewise functions. We will show how to transform these into Python code directly. In the study of numerical methods the textbooks quite often contains pseudocode as a kind of meta-level description of how an algorithm should work and subsequently be implemented. The textbook for this course is no exception and hence we will also treat the subject of how to convert pseudocode into Python code. Armed with this knowledge a student will be quite well-prepared for implementing many of the numerical methods we will encounter in the textbook of this course.

After having worked through this chapter the student should be able to

- Account for how to create Python code corresponding to summations, products and piecewise functions as given in mathematical equations
- Explain what pseudocode is and describe common issues encountered when writing Python code with outset in pseudocode

7.2 Python Prerequisites

A student should at this stage be able to

- Open and use his/her integrated development environment (IDE) and visual debugger
- Master variables, algebraic expressions, floating point types
- Be able to use printing for generating output from Python programs
- Master control flow statements such as if-statements and for-loops
- Master list data structures

One can learn the needed Python background by studying Chapter 1 and 2.

7.3 Other Math Equations That Are Easily Converted into Code

So far we have studied sequences of numbers. These we can characterize by having a so-called running index, the n-integer value that is used when setting up the for-loops we have been writing up till now. We will now instead study a math equation involving a summation

$$S = \sum_{n=1}^{N} x_n \tag{7.1}$$

where in this example we assume that we are given some list or sequence of known values $x_n \in \mathcal{R}$, $\forall n$. This is just the most simple generic exemplary form of a summation we could come up with and will serve as a good example to illustrate how to make corresponding Python code for computing the value of S.

When we study the mathematical construction then we here too observe the notion of a running index. To make the analogy stronger to our previous sequences example problems we used nn in the equation above. However, as this is a running index we could replace this symbol with any symbol. It is not important if one uses n, i or j for that matter. Namely, we have

$$S = \sum_{n=1}^{N} x_n = \sum_{i=1}^{N} x_i = \sum_{j=1}^{N} x_j = \sum_{\alpha=1}^{N} x_\alpha \tag{7.2}$$

For sake of making our example more explicit and concrete then let us for now assume that

$$x_n = \frac{1}{n} \tag{7.3}$$

Now we can prototype a Python program that can compute the summation. The idea is to use a for-loop that iteratively accumulates the sum by adding one value at the time. Here is an example of how this will look in Python

```
N = 10
S = 0
for n in range(1,N+1):
    S = S + 1/n

print('The total sum is =', S)
```

Notice that we simply had to make some choice for the value of N. Nothing in the problem description told us what value to use. Hence we are following our past advice of picking some small value while we are prototyping to get our code to work right. Observe how we carefully wrote the for-loop range such that n starts at value $n = 1$ and stops the loop right after having done $n = N$.

Try running the code for yourself. You should obtain an output like this:

```
The total sum is = 2.9289682539682538
```

As the example shows there is a clear connection between a summation symbol in a mathematical equation and a for-loop in Python. The observant student may recall our past issues in dealing with zero-based indexing versus one-based indexing. In the example we took great care to make the running index take on the exact same values of the defining summation used. However, imagine that we received the numbers to add up as a list and not as the closed form equation given above. That is we would be given a list like this

```
x = [ 1., 0.5, 0.33, 0.25, 0.2]
```

How can we change the prototype to use the list of given data? Obviously Python allows for some very nice syntax where one can simply write

```
x = [ 1., 0.5, 0.33, 0.25, 0.2]
S = 0
for x_n in x:
    S = S + x_n

print('The total sum is =', S)
```

This is convenient, but being persistent and sticking with the running index solution, how should the code then look like? Recall that the list data structure uses zero-based indexing. Hence n must run from 0 to $N - 1$ to work.

$$S = \sum_{n=0}^{N-1} x_n \tag{7.4}$$

The Python code will be

```
x = [ 1., 0.5, 0.33, 0.25, 0.2]
N = len(x)
S = 0
for n in range(0,N):
    S = S + x[n]

print('The total sum is =', S)
```

Clearly, the code is a bit more bloated than the syntax-nice solution. However, this prototype demonstrates that indexing issues can be handled in much that same way as we did when working with sequences. Above we redefined the summation range in the actual defining mathematical equation. We could have used on-the-fly index conversion instead.

Summations are not the last mathematical construct that translates naturally into a for-loop. Products are another example. Let us study something specifically, like this equation

$$P(x) = \Pi_{n=1}^{N}\,(x - r_n) \quad \wedge \quad r_n \in \mathcal{R},\, N \geq 1\,. \tag{7.5}$$

Here we assume that the r-values are given to us as a list of values. The key to get the link to a for loop is to notice that the multiplications of the terms $(x-r_n)$ can be performed one by one. Hence a prototype for this problem of computing the value of $P(x)$ for a specific given value of $x = \frac{1}{2}$ and $r_1 = -2$, $r_2 = 2$, $r_3 = 0$ gives the Python code

```
r = [ -2., 2., 0.]
N = len(r)
x = 1/2
P = 0
for n in range(0,N):
    P = P * (x-r[n])

print('The total product is =', P)
```

Without stepping too much into it we simply add the note here that this construction we showed above too suffer from possible zero and one-base indexing headaches if one is not careful about it.

It is worthwhile to now reflect on what we have learned on a higher abstraction level

- The values of sequences, summations and products are naturally computing using for-loops. Common to all of them is the issue with zero and one-based indexing. The good news is that the issue can be addressed in the same way for all of these.

Seems all we ever need for converting simple math equations into Python code counterparts is a for-loop. Not quite, there are some cases that naturally calls

for if-statements. For instance, all mathematical equations that make use of piecewise definitions are natural candidates for using if-statements. Let us study a very simple example of this here. Consider the function defined by

$$f(x,y) = \begin{cases} x; & \text{if } x \geq y \\ y; & \text{otherwise} \end{cases} \tag{7.6}$$

This is a very often used function. Maybe you can identify it by observing that it always picks the largest value of the two given numbers?

The Python code analog to this function is this code

```
x = 2
y = -3
f = None
if x >= y:
    f = x
else:
    f = y

print('f(', x, ',', y, ') = ', f)
```

The Python analog for this example is perhaps too obvious, but consider a slightly more complicated mathematical expression like this

$$S = \sum_{n=1}^{N} \max(x_n, 0) \tag{7.7}$$

Assuming that N is given and that all the numbers x_n are given as inputs. A headache may appear quickly unless one notices that the maximum function,max, is the same as the f-function in the previous example. Hence, a possible solution may be the code

```
x = [ 1., 0.5, 0.33, 0.25, 0.2]
N = len(x)
S = 0
for n in range(0,N):
    if x[n] > 0:
        S = S + x[n]

print('The sum of max-terms is =', S)
```

Observe that in this solution we "optimized" the if-statement to take into account that $y = 0$ always and further we took zero-based indexing into account too.

Let us study another common mathematical expression that calls naturally for if-statements. The example is a quite famous recurrence relation which we

exemplify here by the equations

$$x_0 = 1\,, \tag{7.8}$$
$$x_1 = 1\,, \tag{7.9}$$
$$x_n = x_{n-1} + x_{n-2}, \forall n \geq 2\,. \tag{7.10}$$

Out task now is that given some value $N \geq 0$ write Python code for computing all values of x_n for $0 \leq n \leq N$. We immediately observe that we are to create a sequence of numbers, so we decide to store these into a list as we compute them. Hence a prototype solution may be

```
N = 10
for n in range(0,N+1):
    if n <= 1:
        x.append(1)
    else:
        x.append( x[-1] + x[-2] )

print('The sequence is =', x)
```

Again we tried to be a little clever about how the code works when running. We notice that for `n==0` and `n==1` the same thing should happen in the code. Hence, rather than writing the "naive" solution

```
if n == 0:
    x.append(1)
elif n == 1:
    x.append(1)
else:
    x.append( x[-1] + x[-2] )
```

We notice the two identical blocks and collected these into one single if-condition. This reduced the amount of code we needed to write. The observant student should notice that such clever rewrites is another reason why there could be extremely many programs slightly different in implementation but would solve the problem at hand.

The output from the code above should be

```
The sequence is = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

The example shows that recurrence relations are naturally interpreted as if-statements in Python code. Hence, we may now conclude on our learning, namely

- Piecewise function definitions transform into if-statements in Python
- Recurrence relations transform into if-statements in Python

REMARK It may appear to students that we can generate quite a lot of Python code if we have to write out for-loops and if-statement completely every time we meet these kind of simple mathematical equations and expressions. Hence, we feel that a remark on this observation is in place. The examples we use are contrived and designed so that they serve as a perfect setting for learning and training basic concepts of programming. As the programming concepts grow more familiar and as one gains more experience it is natural that one would transition to using built-in functions and possibly lambda expressions for solving such examples. In particular the sum and map functions would be useful for generating sequences or computing summations. The filter and zip functions are also good to have in one's arsenal too. For instance a list of a sequence of numbers can be computed simply by writing one line using the map-function and a lambda expression

```
values = list(map(lambda n: 1/n, range(1,11)))
```

Using list comprehensions this can be written even shorter

```
values = [1/n for n in range(1,11)]
```

Conceptually, these one-liners produce the same output results as using the for-loops and appending results to a list. To be perfectly honest: As a rule of thumb it is generally a better choice to use built-in functions or list comprehensions when programming. The reasons for this is that the code footprint (ie how much space the code takes up) is far smaller, hence the risk of making a typo or mistake is reduced. Furthermore, the built-in functions may already have been optimized by other programmers to take advantage of special hardware or parallel-computing capabilities. This way one gets better performance for free. Lastly, a one-liner may be much simpler to read and hence easier to work with when having to read old code sometime in the future.

7.4 Making Code from Pseudocode

Pseudocode can look deceivingly close to a real programming language. The first example of this can be seen in the textbook when it presents a method for evaluating the value of a n'th degree polynomial at a given x-value. Here we reproduce the pseudocode from the textbook

$p \leftarrow a_n$
for $k = n - 1$ to 0 step -1 do
  $p \leftarrow xp + a_k$
end do

This looks a lot like code, but it is not. It won't even compile. Clearly this is the case for us, as it is not correct Python language syntax. It is quite common to mix in math symbols and notation when writing pseudocode. This is very unlike real programming languages.

The purpose of pseudocode is not to make an implementation of a program. The intention is to give the human reader a better understanding of the algorithmic flow. That is to help a human understand the order of computations and the kind of sub-problems being solved, but abstracting away tedious details that an actual f.ex. Python implementation would need to consider. Generally speaking, syntax and semantics of pseudocode are often not rigorously used nor defined when people write pseudocode. Instead authors rely on a common implicit understanding of notation and general knowledge of typical programming language concepts, like for-loops, if-statements etc.. For instance, in the textbook pseudocode example listed above the left arrows are used to specify assignments of values to variables. In Python terminology that would be a binding of a name to an object. The for-loop syntax illustrates starting, stop and step values and subscripts on variable names are like indexing into a data structure, similarly to indexing into a list in Python. Indentation is not used to specify scope (blocking). Instead this is done with a "do" to "end do" notation.

It may appear to a student at this stage that we can make a rule book of "translations" like shown in the above, and although it seems to work for this simple numerical method, many other pseudocode examples exist where no obvious rules can be applied. Consider for instance the imaginary bogus example shown below

```
for all element pairs A and b do
  solve Ax = b for x
  if r = b − Ax small enough then
     use x as solution
  end if
next A and b pair
```

This example is not taken from the textbook. Your teachers designed it to make a point. Notice that even though some of this description clearly make sense to a human reader, one would need to make many decisions to make an actual working implementation run on a computer. Such as: How does one generate the pairs of "A" and "b"? What type of solver does one use for solving the linear systems? What does it mean to say that "r" is "small enough"? In fact it is not even explicitly clear what variable types, "A", "b", "r" and "x" really have in this example, or what data structures are used to store A's and b's. Blocking indicators are also a bit inconsistent and indentation as used in Python is not the style being used here. The point we wish to make here boils down to

If one does not have the same implicit understanding of a given pseudocode as the actual author of the pseudocode then there is a large risk of misunderstanding what goes on, or not having a clue about how to make an actual implementation. On the other hand, if one does have the same implicit understanding, then pseudocode can be a very nice, short and abstract way to communicate an algorithm. The first pseudocode example shown above illustrates the numerical method known as Horner's method or synthetic division as it is also called. The pseudocode summarizes the logic and order of operations that results from rewriting the definition of a polynomial using nested multi-

plications. This is one approach for developing such a meta-level description of an algorithm via the pseudocode. Let us briefly illustrate this approach. The nn'th degree polynomial is defined as

$$p\,(x) = \sum_{k=0}^{n} a_k\, x^k \tag{7.11}$$

However, one can rewrite this into the form

$$p(x) = a_0 + x(a_1 + x(a_2 + x(\cdots x(a_{n-1} + x(a_n))\cdots))) \tag{7.12}$$

This form is barely readable by most people, but reworking it piece by piece one can see the relation to the pseudocode. If we re-work the nested multiplication a little by introducing the auxiliary variables $p_0, p_1, \ldots, p_n$ then we find a recurrence relation

$$p_n = a_n \tag{7.13}$$
$$p_{n-1} = x\, p_n + a_{n-1} \qquad n > 0 \tag{7.14}$$
$$p(x) = p_0 \tag{7.15}$$
$$\tag{7.16}$$

This is actually the piece of "math" that the pseudocode shows how to compute. Hence, pseudocode is a more human friendly way to understand a numerical method.

In summary, for us, this pseudocode is a high-level description of a numerical method. The purpose is to give the human reader, namely us, a better understanding/intuition about how the algorithm works, such that it will be easier for a human to make an actual implementation in a specific chosen programming language. As such we consider pseudocode a kind of meta-level loose description of a numerical method (an algorithm).

Here we show the Python code for Horner's method, we used prototyping taking the pseudocode as the starting point for making a boiler-plate code

```
x = 0.5                              # Evaluation point
a = [ 2., 0.5, 3., 1., 0.1]          # Coefficient of Polynomial
N = len(a)                           # The degree+1 of the
    polynomial
p = a[N-1]
for k in range( N-2, -1, -1):
    p = p*x + a[k]

print('p(x) =', p)
```

As pseudocode is a bit loosely defined, we caution students to be very careful when using it. In fact the textbook we use on this course never explicitly defines the meaning of the pseudocode syntax that it uses.

Our examples show that pseudocode can be a quite powerful tool for understanding the algorithms and making implementations. However, as we illustrated above, the interpretation of the pseudocode is rather implicit and herein lies a huge danger. The freedom in notation and implicit semantics implies that misunderstandings occur or misinterpretations of algorithms happens. Such errors are very hard to detect and fix when trying to make implementations work.

7.5 Where to go Next?

Let us practice out new skills for transforming math equations into Python code

- Write a Python program that can analyze the sequence $\{S_n\}$ with $S_n = \sum_{i=0}^{n} \frac{1}{2}^{n+1}$ and determine if it converges or diverges. Try also to experimentally analyze the order of convergence. (Hint: A sequence of summations suggests perhaps a solution using a double for-loop, also called a nested for-loop).

- Implement the pseudocode shown below

```
input n,k
r ← 1
for i = 1 to k do
  r ← r(n + 1 − i)/i
end do
output r
```

- Implement the pseudocode shown below

```
input R
a_{1,1} ← 1
for r = 2 to R do
  for c = 1 to r do
    if c>1 and c < r then
      a_{r,c} ← a_{r−1,c−1} + a_{r−1,c}
    else
      a_{r,c} ← 1
    end if
    output a_r
 end do
end do
```

(Hint: This is in fact Pascal's Triangle).

Chapter 8

Making your Code more Generic

8.1 Introduction

The Python language we have used so far enables us to implement rather complex numerical methods. However, to be truthful, your teachers appear to be mad and may seem to ask you to do foolish things. Well at least it may appear this way if you reflect upon what exercises your teachers have asked you to do so far. On more than one occasion your task has been to take almost identical code and modify this code into a specific tailored new prototype for a slightly different math equation. The new prototype often looks quite similar to the previous one. The only major difference seems to have been to replace a few algebraic expressions. Attacking programming like this is bound to generate a lot of almost similar code and that seems wasteful for many reasons which we will discuss a little in this chapter.

Fortunately, your command of the Python language should now have been extended to understand how to write functions and modules. It turns out that these programming concepts are quite important and useful for designing your program when you make prototypes. If done correctly one can create lean and mean code that is easy to maintain and can be reused easily without having to make changes to the existing code. This is the subject of this chapter – How to make more beautiful and useful programs when we prototype our solutions.

After having completed this chapter the student should be able to

- Implement prototypes using functions to wrap similar functionality into logical units
- Implement prototypes by organizing code (i.e. functions) into modules

8.2 Python Prerequisites

Students should be able to

- Master PyCharm and the visual debugger
- Be fluent in using variables, expressions, types, control flow (if-statements and for-loops), lists
- Be fluent in using print and plotting to create and display visual output from programs
- Explain how to write a function in Python
- Describe how scoping works
- Explain how to write a module in Python
- Know how to write a proper "import" statement in Python

This Python knowledge and skills are covered by Chapter 1 and 2.

8.3 Functions Make Code More General Applicable

Let us revisit the tasks of analyzing sequences of numbers where each number is given by some closed-form equation.

Recall for instance the sequence given by

$$\{x_n\} \quad \wedge \quad x_n = \frac{1}{n} \quad \wedge \quad n > 0 \tag{8.1}$$

Or given by

$$\{x_n\} \quad \wedge \quad x_n = \frac{n+1}{n^2} \quad \wedge \quad n > 0 \tag{8.2}$$

From these two equations we already notice the algorithmic similarity between them. The only real difference appears to be how we compute the value of x_n. The closed-form assumption means that x_n only depends on the value of n. This suggests that we could replace the expressions by calling some function $f(n)$ that would compute the value x_n.

In past chapters we learned to be careful about the starting value of the running index. Clearly in the above example using $n = 0$ would not work with the closed-form definitions. However, for other closed-form equations this could be different and $n = 0$ is the desired starting value. For instance $x_n = r^n$ for some given constant r-value. Hence, as both behaviors are needed and depend on the particular closed-form solution the user wishes to analyze, logic dictates that the user must tell the analyzing tool what starting index value to use.

Thinking about making a generic design, meaning we only want to write the analyzing code once and then never have to modify this part of the code again, means we would like to feed a closed-form function, f, to the sequence analysis tool. This f-function can then be invoked by the "analysis" when in need of computing x_n. Let us try and sketch the idea in a prototype. We wish to pass a closed-form functionffas an argument to a function that can analyze a sequence. The "analyzer" should then invoke the given f-function when wanting to compute the value of x_n. This will look somewhat like this

```
import matplotlib.pyplot as plt


def analyze_sequence(f, n0, N):
    values = []
    for n in range(n0, n0+N+1):
        x_n = f(n)
        values.append(x_n)

    plt.figure(1)
    plt.plot(range(n0, n0+N+1), values, 'ro')
    plt.show()
```

To give users more control over the analyzer function the starting index value and the number of numbers wanted is specified. What we remain to do is to implement the closed-form functions. As these are simple algebraic expressions that is done more or less naively by converting the math equations directly into expression counterparts in Python code, like so:

```
def f1(n):
    return 1/n

def f2(n):
    return (n+1)/n**2
```

Finally, we have all the pieces in place to analyze the two given sequences by invoking the `analyze_sequence` function with proper arguments. That would be

```
analyze_sequence(f1, 1, 10)
analyze_sequence(f2, 1, 10)
```

Notice, we choose starting index values for both sequences to be 1 and that we followed our own advice and started prototyping using a small number of numbers, $N = 10$. The whole prototype code now reads

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

```
def analyze_sequence(f, n0, N):
    values = []
    for n in range(n0, n0+N+1):
        x_n = f(n)
        values.append(x_n)

    plt.figure(1)
    plt.plot(range(n0, n0+N+1), values, 'ro')
    plt.show()


def f1(n):
    return 1/n


def f2(n):
    return (n+1)/n**2


analyze_sequence(f1, 1, 10)
analyze_sequence(f2, 1, 10)
```

Imagine that your mad teacher comes along now and ask you to analyze yet another sequence, say

$$x_n = 2^{-n}, \quad n \geq 0 \tag{8.3}$$

You may feel the eye twitching when thinking about copying some boiler-plate code and starting to hack it and tailor it into a specific prototype for this new sequence. However, when you think twice about this and recall your generic analyzer tool then you create a closed form function and invoke your analyzer on this new function, like so

```
def f3(n):
  return (1/2)**n

analyze_sequence(f3, 0, 10)
```

Observe that by carefully redesigning your code into logical chunks of "similar" behavior you managed to solve the next task given by your teacher using only the few rather simple lines of code shown above. It is evident that you can save tremendous amount of time from here on when your teacher ask you to solve different sequences. We call this to reuse one's own code and it is like studying the art form to become good at the craft of designing good reusable code. In our experience a lot of practice seems essential to gain the needed experience.

A perhaps less obvious benefit from making reusable code is that once you have implemented and tested the code then you can trust the code not to have

bugs. Common sense dictates that by reusing bug-free code one reduces the risk of introducing bugs oneself.

Let us summarize our learning efforts so far

- Functions allow us to create reusable code and minimize risk of bugs in the long run
- We may use a function to parametrize different behaviors of similar tasks or problems. This is the `analyze_sequence` function. The different behaviors are the closed-form definition (f-function), starting value of running index (n_0 argument), and number of elements to compute (N argument).
- We used higher order functions to make it possible to call functionality (the f-function) defined elsewhere (`f1`, `f2` and `f3` functions. This programming pattern is termed a callback function. Python makes callbacks easy to implement using higher order functions as we showed in the example above.

8.4 Modules Make Code Easier to Organize

By example we will learn about the usefulness of modules. Going back to the analyzer example from past section it seems the code has become much more reusable. However, reflecting on how one would use the code it becomes clear that we are still unhappy with one issue. Namely when we add more sequences to analyze we would need to add more callback functions, like `f5`, `f6`, `f7` and so on, to the Python file that contains the `analyze_sequence` function. Furthermore, when running the analysis we must invoke the `analyze_sequence` function using the new call-back we added, all inside this single Python file that already contains all our older tasks that we have solved.

For now when our programming tasks are small and not too complex this may not seem like a big deal breaker. However, over time as mad teachers generate more tasks for you to solve your Python file will start get bloated with callback functions that will pollute the nice work-space where you implemented your nice re-usable function. Over time you may start to out-comment older "tasks" to avoid side-effects from this pollution. What we wish for is the capability to split the code into several Python files helping us to reorganize the code and avoid pollution.

We will keep one single Python file, which we here call "analyze.py". This file holds the nice re-usable function and nothing else. The idea behind splitting this function into its own file, is that we can keep avoid ever to have to write or make changes in the analyze.py file again. That is we have

Listing 8.1: analyze.py file

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

```
def analyze_sequence(f, n0, N):
    values = []
    for n in range(n0, n0+N+1):
        x_n = f(n)
        values.append(x_n)

    plt.figure(1)
    plt.plot(range(n0, n0+N+1), values, 'ro')
    plt.show()
```

Creating a py-file means we created what in Python is called a module.

One would also not wish to mix up the code from the different tasks that the mad teacher has asked one to solve. Hence, we wish to have one Python file for the first task, another file for the second task and so on. That would imply several Python files, that we here name task1.py and task2.py. We need to make sure that the task1.py and task2.py files can see the function in the analyze.py file. This is done using an import statement. Specifically we use the form `from XXX import YYY`. The resulting code look like shown below.

Listing 8.2: task1.py file

```
from analyze import analyze_sequence


def f1(n):
    return 1/n


analyze_sequence(f1, 1, 10)
```

Listing 8.3: task2.py file

```
from analyze import analyze_sequence


def f2(n):
    return (n+1)/n**2


analyze_sequence(f2, 1, 10)
```

We can now run either of the task files independently of each other. Hence, the two programming tasks will not cause side effects for each other. It is also easier for us to get an overview of what goes on inside the files as the code footprint is kept smaller by organizing the code in this way. Particularly we do not have to make changes to older tasks when confronted with solving new programming tasks.

If the teacher comes up with more tasks we can simply copy one of the task1.py files into a new file, say task3.py (and so on), and use this copied file as a boiler-plate code for writing the callback function for the next task and invoking the analyzer tool.

Let us reflect on what we learned about using modules

- Modules can be used to split code into individual files, The splitting allows us to reorganize code files to contain "code" that never needs to be written or changed again. Organizing code into logical units helps avoid side-effects from older code and helps provide better overview of the code by avoiding bloating and pollution of the code.

8.5 Where to Go Next?

Try to solve the tasks below to gain more practice in programming

- Create an optimized version of the `analyzer_sequence` function that uses list comprehension
- Inspired by the `analyzer_sequence` example, implement a function that uses a recurrence relation to compute the numbers in the sequence rather than an explicit closed-form solution. Assume the recurrence relation to be of the general linear homogeneous form

$$x_n = \sum_{i=1}^{K} c_i \, x_{n-i} \tag{8.4}$$

  Assume the coefficients c_i are given as input to you in a list as input and K can be found by your program from the length of this list. Design a function `analyzer_recurrence` which uses a callback function to compute x_n. You also need the first K values $x_0, x_1, \ldots, x_{K-1}$ as input to your function. Test your implementation using `c=[1,1]` and initial x-values `[1, 1]` (Hints: Use `analyzer_sequence` as boiler-plate code, write a generic callback function to implement the linear recurrence relation)

Chapter 9

Python Quirks, Testing and Commenting

9.1 Introduction

At this stage of our learning curve in programming with Python we have a good basic understanding of the core language and we have started being able to create our own programs from scratch. We are now in this transition from understanding the language to starting to use it for creating meaningful programs. During this process we are bound to stumble across some quirks of using Python. In our own experience two issues often emerge: performance considerations of recursive functions and side-effects from using mutable arguments. In this chapter we will study these two cases more in depth. However, we will also move a little bit beyond these and briefly have a look at testing and commenting your code. In the past we learned how to use our debugger to locate and fix bugs. However, we now need to create tests to have something to debug and we need to make comments in the code to help yourselves at a later stage - or others reading the code - to get a better idea of what various parts of the code might be doing.

After having completed this chapter the student should be able to

- Account for how recursive and iterative methods are two approaches for solving the same problem
- Account for possible performance side-effects of using recursive functions
- Explain by example how to use a cache of computed results to improve performance of recursive functions
- Account for how "name binding" can create side-effects for mutual data structures used as arguments for functions
- Explain by example how to protect arguments from "mutable side-effects"

- Create test cases for debugging
- Provide sufficient level of commenting in own code

9.2 Python Prerequisites

It is expected that a student is able to

- Write recursive and iterative functions
- Use control flow statements such as if- and for-loops
- Use print statements
- Use lists and dictionaries
- Know about local and global scope of variables

The language prerequisites are covered by Chapter 1, 2 and 4.

9.3 Recursive Versus Iterative Solutions for Programming

Recursive functions are a very powerful tool for solving certain problems. One of the archetypical examples of this is the computation of the factorial of an integer. We may define the factorial as follows

$$n! = \Pi_{i=1}^{n} i = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1; \quad n > 0 \tag{9.1}$$

$$n! = 0! = 1; \quad n = 0 \tag{9.2}$$

This way of presenting the definition leads to a natural iterative solution like this

```
def factorial(n):
    value = 1
    for i in range(1,n+1):
       value *= i
    return value
```

This is not new to us, we have previously seen how sufficiently simple math equations such as recurrence relations, summations and products have natural loop counterparts in Python. However, one may express the factorial definition more elegantly using a recursive relation

$$n! = n \cdot (n-1)! \qquad n > 0 \tag{9.3}$$

$$0! = 1 \tag{9.4}$$

This way of writing the math illustrates a different solution approach using a recursive function

```
def factorial(n):
    if n>0:
       return n * factorial(n-1)
    return 1
```

Clearly, the implementation reflects the math equation in an almost one-to-one way. In principle one can always rewrite a recursive function into an iterative and vice-versa. The recursive concept is quite powerful and many examples exist that are nicely written as recursive relations, Fibonacci numbers, greatest common divisor, modulus of a number, floating point reminder, Pascal's Triangle, Binomial coefficients, Bisection root search and many more.

Let us proceed to an example that helps illustrate some of the dangers that recursive functions can cause. For this we will study the Fibonacci numbers, which can be defined as

$$F_n = F_{n-1} + F_{n-2}, \quad F_1 = 1, \quad F_0 = 1 \tag{9.5}$$

We immediately see the recursive definition that gives us the Python code counterpart

```
def fibo(n):
    if n>1:
        return fibo(n-1) + fibo(n-2)
    return 1
```

Again we observe the uncanny resemblance to our mathematical understanding of the definition of Fibonacci numbers and our Python implementation for computing them.

The simplicity in this case can be illusive. Let us extend slightly our code example to give us some pointers about what goes on behind the scenes when running the code that we've made

```
K = 0

def fibo(n):
    global K
    K += 1
    print('fibo(', n, ')')
    if n>1:
        return fibo(n-1) + fibo(n-2)
    return 1

fibo(10)

print('fibo was called', K, 'times')
```

Now when computing the 10^{th} Fibonacci number we can see which calls are being made to the fibo-function and as we have used a global variable to count

the number of invocations we can get an idea of how much work is going into our computation. Try running the code. Observe that we get the output

```
...
fibo( 1 )
fibo( 4 )
fibo( 3 )
fibo( 2 )
fibo( 1 )
fibo( 0 )
fibo( 1 )
fibo( 2 )
fibo( 1 )
fibo( 0 )
fibo was called 177 times
```

Even for this small Fibonacci number we observe that many more calls are going on, and many of these calls being made are the same. This illustrates that although we have the mathematical simplicity and beauty of our implementation we are suffering from very poor run-time performance due to these many redundant calls.

Let us rewrite our Fibonacci number implementation to use an iterative approach instead. Note there that if we compute the numbers incrementally from $k = 0$ and up to $k = n$ and keep the last two computed values around then we can always compute the next Fibonacci number. That is

```
def fibo_ver2(n):
    if n < 2:
        return 1
    f0 = 1
    f1 = 1
    f2 = f1 + f0
    for k in range(2, n+1):
        f2 = f1 + f0
        f1, f0 = f2, f1
    return f2
```

Now let us compare the performance of our `fibo` and `fibo_ver2` implementations.

```
import time

start = time.clock()
fibo(30)
end = time.clock()
print('recursive solution took', end-start, 'seconds')

start = time.clock()
```

```
fibo_ver2(30)
end = time.clock()
print('iterative solution took', end-start, 'seconds')
```

When running the above comparison code one obtains an output looking something like this

```
recursive solution took 0.394713 seconds
iterative solution took 8.000000000008e-06 seconds
```

Hence the relative speed-up of the iterative method over the recursive method for this specific element of the Fibonacci sequence (and on your teacher's computer) is approximately by 50000 times.

The learning outcome we have emphasized here is that

- Using recursiveness blindly can have dramatic negative impact on performance.
- Rewriting recursive methods into iterative methods can help increase performance substantially.
-

How can one keep the recursive implementation without paying the huge performance penalty? We will here show a little neat trick of using a cache-technique for remembering previous evaluations of function calls. The idea is simply to use a Python dictionary to store the results of any previous invocations of the function. When a new call is made then one first checks in the cache to see if the cache already contains the results being asked for. If so the cached value is returned immediately - if it's not already in the cache then a new computation is done and the result is stored in the cache for later use. This is how the implementation looks like

```
def fibo_ver3(n, cache={}):
    if n in cache:
        return cache[n]
    if n > 1:
        value = fibo_ver3(n-1, cache) + fibo2_ver3(n-2,
            cache)
        cache[n] = value
        return value
    return 1
```

If we measure performance in a similar fashion as we did before

```
import time

start = time.clock()
fibo_ver3(30)
```

```
end = time.clock()
print('recursive solution w. cache took', end-start, '
    seconds')
```

Then we obtain the output

```
recursive solution w. cache took 1.8000000000018e-05
    seconds
```

Which suggests that now the relative speed-up of the iterative method compared to the cached recursive method is only by about a factor of 2. Thus, we still pay a small cost for using the cache, but the performance of the iterative and the cached-recursive approach is much more comparable now.

We reflect

- Using a cache of computed results can improve performance of recursive functions tremendously

At this point one may merely consider this to be a matter of performance. However, there can be an even more severe drawback of recursive functions. Try running this instructive example for computing the value 1.

```
def  rec(n):
    if n>1:
        return rec(n-1)
    return 1

rec(999)
```

On your teacher's machine `rec(998)` works, but running `rec(999)` causes an error like the one shown below

```
Traceback (most recent call last):
 File "/Users/kenny/play.py", line 6, in <module>
 rec(999)
...
...
...
RecursionError: maximum recursion depth exceeded in
    comparison
```

This is much more serious. Now we cannot even complete the computations. You should try running this simple example and keep increasing the number until you get the recursion error. This way you know the limitations better of your specific computer and system.

This example was quite different from the Fibonacci number example in that no redundant calls were made. Hence, a cache-solution will not work to save this simple example. The problem is caused by how the interpreter works when a function is called. When a call is made a "frame" is put onto a stack. The

"frame" contains all info about the function call and it stays on the stack as long as the function call has not yet terminated. In the above example any preceding function call stays alive with us. It is only when n becomes equal to one that the last function call will terminate, then the previous one and the one before that one and so on. If n is too big this means we run out of stack space to store frames onto before n gets equal to one. Python does not have tail recursion elimination as other languages provide. Hence the only viable solution is to rewrite the implementation into for instance an iterative method.

9.4 Side effects of Mutable Arguments

How do mutable data structures work when we change their values or bind names to them? Here is an example that illustrates the answer and the quirks that can arise if one is not very careful(!):

```
a = [9, 1, 2, 3]
b = a
print('a = ', a)
print('b = ', b)

a[1] = 100
print('a = ', a)
print('b = ', b)

c = b.copy()
b[3] = -100
print('a = ', a)
print('b = ', b)
print('c = ', c)
```

The output of this Python code is

```
a = [9, 1, 2, 3]
b = [9, 1, 2, 3]

a = [9, 100, 2, 3]
b = [9, 100, 2, 3]

a = [9, 100, 2, -100]
b = [9, 100, 2, -100]
c = [9, 100, 2, 3]
```

Let us explain the quirk in a little bit of detail. One could have been mislead to think that the second Python code line

```
b = a
```

Would create a new b-list. However, this is not true. As the output show, when changing values in the elements of a, we are also apparently changing the value of the b-elements. This is because there are not two lists in the code. At this stage we only have one list object and two names 'a' and 'b' that are bound to the same object.

As the example shows, one must explicitly use the copy( ) function to get a new list, like this line

```
c = b.copy()
```

As seen in the example, when making changes to the b-list elements there are no longer side-effects in the c-list. This is because the name c is bound to a new list object.

Python has the built in **id**-function that can help identify if two variables are bound to the same object. The **id**-function does as the name suggest provide us with a unique way to identify what objects we are are working with. Think of the **id**-function as giving you a kind of social security number of the object that a name is bound to. Here is some example code illustrating the idea

```
a = [1, 2, 3]
b = a
id(a)
id(b)
c = a.copy()
id(c)
if id(a) == id(b):
    print('We are the same object')
if id(a) =!= id(c):
    print('We are NOT the same object')
```

The output of the Python code is

```
4493690888
4493690888
4370839304
We are the same object
We are NOT the same object
```

As expected we get different social security numbers for different objects. The way of how the social security numbers work are the same if we worked with say int-types. Let us write a small example

```
a = 1
b = a
c = 2
id(a)
id(b)
id(c)
```

The output will be something like this

```
4297546560
4297546560
4297546592
```

As we learn from this we see that the names `a` and `b` both are bound to the `1` object, where as `c` is bound to the `2` object. To test your understanding imaging extended above code with the lines

```
c = 1
id(c)
```

What would you expect as output? (Try this for yourself and try explaining what happens).

Armed with our knowledge of how variable bindings can let to two names being bound to the same mutable object, we can analyze this illustrative code example

```
def moddy(x, y):
    while x>y:
        x -= y
    return x

a = 5.0
b = 3.0
R = moddy(a, b)
print(a,'mod',b,'=',R)
```

When running this code (try it yourself) we see the output

```
 5.0 mod 3.0 = 2.0
```

as we would expect. For the sake of making an educational point and an otherwise completely non-practical change to our Python code, let us try changing it into one that uses lists. Like so

```
def moddy(x, y):
    while x[0]>y[0]:
        x[0] -= y[0]
    return x

a = [5.0]
b = [3.0]
R = moddy(a, b)
print(a, 'mod', b, '=', R)
```

Running this code, now produces the output

```
  [2.0] mod [3.0] = [2.0]
```

Clearly a true statement, but we kind of expected the output to be

```
  [5.0] mod [3.0] = [2.0]
```

What is going on here?

What happens is that we see a side-effect of using a mutable data type as an argument. If one does use mutable data types as an argument then if one updates these arguments inside a function then one is actually making changes to arguments "outside" of the function. In the example this means that the x-list is really bound to the a-list object. Hence using 'x' to make changes to the object will also reflect changes on the object that 'a' is bound to.

Sometimes this behavior can be quite unexpected. A user of a library might expect that the library does not change the values of any given arguments. Clearly, such a user would be very unhappy with the moddy-function given above! Fortunately, one can make the moddy-function behave a little more as expected by "protecting" the arguments. This essentially means that one as a programmer must make explicit copies of any arguments that must not be "hurt" as a side-effect of the function one is about to program. Here is how the moddy-function could be changed to protect the input variables:

```
def moddy(x_in, y):
    x = x_in.copy()
    while x[0]>y[0]:
        x[0] -= y[0]
    return x

a = [5.0]
b = [3.0]
R = moddy(a, b)
print(a, 'mod', b, '=', R)
```

Now we obtain the expected output

```
[5.0] mod [3.0] = [2.0]
```

Let us summarize what we have learned

- When programming functions that update mutable arguments a caller might find the behavior strange.
- To protect mutable input arguments from being modified, one can make explicit working copies when implementing one's own functions.

9.5 Testing

In this course the teachers do testing of your code to essentially give you a grade. Your teachers wish to be certain that you know how to write small-sized good quality Python code. Here, good quality means from your teachers's point

of view that your code can run without syntax or run-time errors and that it computes the expected numerical results when given correct input values, and without causing strange side-effects or logical errors. However, when you as student do your code testing in this course it is mainly in order to debug your own code so that you know that it is working correctly before you hand it in to your teacher. Testing is therefore part of your work cycle to guarantee that you submit the "right" answer when you make your hand-ins.

In non-academic life, testing is also a very important part of making software and it is a natural part of programming. In fact test-driven development (Links to an external site.)Links to an external site. is one such example where making test cases before programming is used to prove that one's software delivers on any requirements that might have been made.

In this course we do not have the need for such a big machinery, as our code is not too long, nor too complex, and expected outcomes from given input are quite well-defined. Hence, we will settle for an approach which is a little more ad hoc when talking about testing. There are many kinds of different methods (Links to an external site.)Links to an external site. for doing the testing. In this course we are mainly going to make use of what is often called

- black box testing
- white box testing

For hand-ins and grading in this course we will make use of a lightweight framework to write small functional tests. This is termed "unit-tests".

Let us develop the needed concepts of testing by way of example. Imagine that you are given the task of implementing a function that can compute the Fibonacci numbers. Imagine that you never did this before, nor do you have any clue about how to go at it, so you might immediately start prototyping something like this

```
def fibo(n):
    ?
```

At the second line you might halt and reconsider. What is it that this function is supposed to do? Looking at the defining math equation you know the Fibonacci sequence to have the values (or at very least your teacher told you so):

```
1, 1, 2, 3, 5, 8, 13, 21, 44, 75,...
```

So we know that the function should return some new integer given an integer as input. However, we know even more. These first cases provide us with ground truth knowledge about how our implementation should work. Hence, at this point we could actually start adding test-cases to our prototype implementation. For sake of example let us do so

```
def fibo(n):
    ?
```

```
f0 = fibo(0)
if f0 == 1:
    print('success')
else:
    print('failure')

f1 = fibo(1)
if f1 == 1:
    print('success')
else:
    print('failure')

f2 = fibo(2)
if f2 == 2:
    print('success')
else:
    print('failure')

f3 = fibo(3)
if f3 == 5:
    print('success')
else:
    print('failure')

... as many more test cases you care to write will go
   here ...
```

This is an example of doing so-called "black box" testing by writing a "unit-tests". Above we did not know anything about how `fibo` would be implemented. That is we treated `fibo` as a "black box" that we could not see through or look inside in order to figure out how it works. We wrote test-cases using if-statements that test actual output from `fibo` against the expected output. A test-case written like this is like a hypothesis that some specific input (the value of the number n) will produce some expected specific output.

Writing test cases like this has the benefit that it allows us to start debugging and testing even before we think about writing the code of the fibo-function. It gives us the tools to help us debug the code and the tools to prove (to ourselves or others) that our code is working. The reflective student may now ask the question: How many test cases should we make? Or what test cases should we include?

The obvious and not very helpful answer is: enough (= sufficient) test cases such that we with (sufficient) certainty can claim that the code is robust and working for the general case. Imagine a very clever implementation of fibo looking like this

```
def fibo(n):
```

```
    magic = [1, 1, 2, 3, 5, 8, 13, 21, 44, 75]
    return magic[n]
```

Running with just the four test-cases given above we may be lead one to conclude that the fibo-function is working and immediately make our hand-in like so. However, if we added a test-case more like so

```
f30 = fibo(30)
if f30 == 1346269:
  print('success')
else:
  print('failure')
```

Then by adding this test-case we will observe output similar the one shown below

```
Traceback (most recent call last):
  File "...", line 2, in <module>
IndexError: list index out of range
```

Clearly having this blac -box test case of using large values of the input number n helps catch a bad solution. Imagine a new prototype being developed. Given our knowledge that an iterative method might be good for Fibonacci numbers we prototype now instead a solution like this

```
def fibo(n):
    if n < 2:
         return 1
    f0 = 1
    f1 = 1
    f2 = f1 + f0
    for k in range(2, n+1):
        f2 = f1 + f0
        f1, f0 = f2, f1
     return f2
```

Now all our test-cases passes and we conclude the code is working sufficiently correctly and we can make our hand-in based on it. Unfortunately, your sneaky, but high-integrity teacher was using another set of test-cases for your grading. These test cases were

```
fn1 = fibo(-1)
if fn1 == 0:
  print('success')
else:
  print('failure')

f9 = fibo(9)
```

```
if f9 == 44:
  print('success')
else:
  print('failure')

f30 = fibo(30)
if f30 == 1346269:
  print('success')
else:
  print('failure')
```

The teacher will give you a pass grade if more than 50% of the test cases passes. Running on the new prototype the teacher observe

```
failure
failure
success
```

and concludes the student failed the programming test. This contrived little example will help us illustrate one more important fact about writing test-cases, namely

- Test cases are no better than the person writing them

Looking closely at the teacher's first test-case, we observe that the teacher was thinking about how `fibo` should work on numbers which are in a sense "illegal". In this case the teacher made the assumption that the zero value was supposed to be the right value. Many other assumptions could have been used and the test may in a sense be unfair, if the student was not informed of this. Still testing if one's code is robust on "illegal" input makes very good sense. In some cases infinite loops or whole system crashes (and hence the risk of hacking of sensitive systems) can be caused by letting one's code accept and attempt to work with "illegal" input values.

We are very happy when looking at the last test-case made by the teacher. Clearly the iterative solution was a good idea for matching this test-case. However, when looking at the middle test-case made by the teacher it becomes strange again. All the student test cases passed, testing n = 0,1,2,3 and n = 30 worked too, so why did the case n = 9 fail? There are two points to be made here.

The student did not discover that there was an issue here, as the student test cases never went to larger numbers than n = 4. If the student included the test case for n = 9 the student would discover that the teacher did not know the Fibonacci numbers, and that fibo(9) should equal 55 and not 44! (Did you catch the mistake yourself before?) The teacher used wrong ground truth data for writing a black box test-case. Given the experience so far, we now formulate some general pieces of advice which one should keep in mind when thinking about "black box" testing in this class

- Think about test-cases before writing the code solution
- Verify the test-cases:
  - Are there enough and varied enough test-cases to "stress" the code? Do the input values fairly resemble all thinkable scenarios, or should one add more test cases to make certain that one's code is really working in all cases? If relevant, has invalid/illegal input values been considered?
  - Have extreme input values been considered? (Such as using a very large input number in `fibo`.)
  - Are the test-cases correct? Do they test if a wrong answer is given?

For designing test cases it can be helpful to think about grouping input values into logical sets that should have similar output behaviour. For the fibo-example this could be like

- The set of all illegal values $n < 0$
- The set of initial conditions $n = 0$ and $n = 1$
- The set of reasonable thinkable values, like $n = 4$, $n = 9$, $n = 10$, $n = ...$
- The set of extreme thinkable input values, like $n = 1000$

Clearly, generating test-cases from these four sets will help us verify that we can say that the `fibo` function can compute Fibonacci number in a general as well as robust manner.

We have cheated a little when generating the first and last logical sets of input values that we should consider testing. We actually knew from experience that Fibonacci numbers are likely to be computed using either an iterative or recursive method. Hence, those two sets would help test if such implementations have correctly considered extreme cases. For instance the negative inputs if used as stop-conditions in loops can cause infinite loops to occur. Extreme large values when used to determine if recursion should halt can stall a computer or cause a run-time recursion error, as we saw.

When starting to reason about the intrinsic details of how `fibo` is actually implemented, then we are moving away from "black box" testing towards the direction of "white box" testing. "White box" testing also works by writing test cases, but the test cases are generated based on intimate knowledge and know-how of the actual code that is being tested. As the name reflect, white is opposite from black, hence a "white box" is a box that one can see through.

Typically one would care about stop-conditions, input data types, range of input values, recursion etc. when generating white box test cases. For the case at hand, the current fibo-function is rather simple. The if-statement n $<$ 2 suggests that we at least should test for two numbers, one smaller than two and one larger than or equal two. The for-loops suggest testing for large n-values too. Negative n-values could be bad for this for-loop but that should be prevented

by the if-statement. Still, creating the test might be a good idea to ensure the code is working in all possible cases.

We have learned

- "White box" testing requires knowing the implementation details and generating tests from that implementation or knowledge hereof.
- "Black box" testing consider the implementation unknown. Test cases are generated from "hypotheses" about how input values map to output values only.
- Your teacher is only human, so do not trust his test cases - make sure you have enough of your own test cases

Up to this point our need for writing "unit-tests" could be handled with simple if-statements. However, we start doing computations with floating point numbers "equality" testing is no longer a great idea. Furthermore, if the output of a function is a 10 x 10 matrix it becomes painful to write unit-tests for each single entry in that matrix. To make life easier your teachers have designed a small primitive auto unit test frameworkPreview the documentView in a new window that allows one to quickly write unit-tests. We will now briefly introduce how this module works.

Firstly, when writing a test script one must import the unit-test module. This is done using an import statement

```
import auto_test_tools as att
```

At some later point in the script one may wish to start creating tests. Surrounding all the test cases one must explicitly tell the test tool where the testing starts and stop, like so

```
import auto_test_tools as att


# Here there will be some code you wrote that you wish to
    test

att.start()

# Rest of test code goes here in next steps

att.stop()
```

A test script is subdivided into test tasks. We call them this as students are often asked to solve a programming task, and hence there should be a corresponding set of test tasks for that programming task. A test task is created like so

```
import auto_test_tools as att
```

```
# Here there will be some code you wrote that you wish to
    test


att.start()

att.begin_task('Task 1')

# Here we will add test cases

att.end_task()

att.begin_task('Task 2')

# Here we will add test cases

att.end_task()

att.stop()
```

So far the example contains two test tasks. We may now generate test cases inside the test tasks. For this our `auto_test_tools` provides many functions that can be used. Below we show their signatures

```
def float_is_close(a: float, b: float, tolerance: float,
    line: int, msg: str = None) -> bool:


def is_equal(a, b, line: int, msg: str = None) -> bool:


def list_is_close(x: list, y: list, tolerance: float,
    line: int, msg: str = None) -> bool:


def vector_is_close(x: np.ndarray, y: np.ndarray,
    tolerance: float, line: int, msg: str = None) -> bool:


def matrix_is_close(A: np.ndarray, B: np.ndarray,
    tolerance: float, line: int, msg: str = None) -> bool:
```

Observe that the function names suggest what the function actually does. The first two arguments for all functions are the expected and actual values to be compared in the test. All floating point test cases the same third argument `tolerance`. This defines the acceptable relative error for two non-zero floating

point value, or the absolute error in case one of the floating point values is zero. All functions take a line argument next. This number is used to generate output error messages when errors occur. One should simply always just use

```
att.get_linenumber()
```

For this, the last argument is a user-specified string value. This can be anything. However, it might be smart to write something sensible that allows one to remember what the test was about. Let us complete the example. Imagine two tasks, write a function to multiply floating point numbers and integers. A simple solution with corresponding test script will look like this

```
import auto_test_tools as att


def mymuller(a,b):
    return a*b


att.start()

att.begin_task('Task 1')
a1 = 5.0
b1 = 5.0
c1 = 25.0
d1 = mymuller(a1,b1)
att.float_is_close(c1,d1,10e-7,att.get_linenumber(), '
    float multiplication failed')
att.end_task()

att.begin_task('Task 2')
a2 = 5
b2 = 5
c2 = 25
d2 = mymuller(a2,b2)
att.is_equal(c2,d2,att.get_linenumber(), ' integer
    multiplication failed')
att.end_task()

att.stop()
```

With this tool in the arsenal one can start writing test cases or extend given tests by teachers to make sure one's own code is working.

9.6 Where Did That Comment Go?

For every time a teacher gives a programming task there is likely going to be as many different solutions to the task as there are students solving that task. Clearly the sources to differences are large. Personal style for how to name variables, to whether one prefers recursive functions over iterative for loops or preferences for using lists or instances of `ndarray`. Hence, the teacher now has a hard time correcting the different answers. Particularly, if only raw code is provided. The teacher will likely appreciate the students who used code commenting to describe non-trivial parts of their implementation. This is one example where code comments come in handy. Helping others to understand what goes on. When one has been programming for years then a hard lesson to learn is that the person most in need of the code comments often turn out to be yourself! Hence, it makes for best practice to make sure that one can understand what one did in the past, in order not to waste time re-inventing the wheel.

It is thus not hard to see the value of code comments. However, how does one move from knowing the syntax of one-line and multi-line comments to figuring out where and how to use them?

The best advice we can give is to use comments, but don't waste them on commenting on something obvious. Like so

```
import fancy.pancy as fp   # Get fancy pancy stuff into
    short name fp for convenience


def f(a, b):
    """
    This is a function that takes two arguments as input.
    """
```

It is perhaps painstakingly obvious that the comments above do not add any real value to the information we already could have obtained from just reading the code in the first place. Comments should be used in such a way that they add important information not already trivially visible.

Let us continue the examples.

```
def hugo(A):
    # Fills in a matrix A such that A[i,j] = i*i/(j+1)
    for i in range(M):
        for j in range(N):
            A[i,j] = i*i/(j+1)
    return A
```

The above examples are arguably still simple. However, clearly the commenting starts to have more value. Here the one-liner elaborates on the purpose/intent of the double for-loop.

There are plenty of standards and style guides as to how one should write comments. It seems that every programmer has their own rule book for this. You may wish to read a little about the PEP 8 (Links to an external site.)Links to an external site. style guide. The PEP 257 (Links to an external site.)Links to an external site. Style guide might also be fun reading.

On a final note, code comments can actually be used to write documentation for one's code while one writes the code. For instance the use of documentation strings, also called docstrings (Links to an external site.)Links to an external site., is an example. In this course the teachers mostly use a style of documentation that resembles reStructuredText (Links to an external site.)Links to an external site. (reST) which the tool SPHINX (Links to an external site.)Links to an external site. can use. Both reST and docstrings are supported by most integrated development environment (IDEs) to provide context-sensitive or fly-over help for the end-users.

9.7 Where to Go Next

It is now time for some training of the concepts that we have been investigating

- Use black box testing for writing sufficient unit-tests to test if a greatest common divisor function is really working as it was supposed to. All you know about the greatest common divisor function is that it is defined as this

```
def gcd(a: int, b:int) -> int:
"""
Computes the positive integer number d defined by
    that
there exist n and m integers such that both a - n*d
    == 0
and b - m*d == 0. In other words d is the largest
    positive
integer that one can divide both a and b by and get
    the
remainder zero. Notice that m and n must be
    relatively prime.

:param a:       The first input value.
:param b:       The second input value.
:return:        The greatest common divisor of a and b
    .
"""
    ...
```

Now generate unit-tests using a black box testing approach. Reason about the test cases you design. Think about whether you can bundle the set of

input values into some logical subsets where you know that you expect the same behavior in the output values. You may use the `auto_test_tools` made by the teachers or write simple if-statements if you feel more at ease with this.

- Implement an iterative and a recursive solution for computing the greatest common divisor.

$$\gcd(a, 0) = a \tag{9.6}$$
$$\gcd(a, b) = \gcd(b, a) \tag{9.7}$$
$$\gcd(a, b) = \gcd(a - b, b); \quad a > b \tag{9.8}$$
$$\gcd(a, b) = \gcd(a, b - a); \quad a < b \tag{9.9}$$

  Consider if there are any that quirks you need to pay attention to or not. Once you have designed your solutions, use the test-script from previous exercise to verify that your implementation is working.

- Use the same approach as shown in this chapter for comparing performance between and iterative and recursive method on your gcd solutions. Which of your versions are the best and how much is the speed-up over the other version?

- (Trick Question) Given the Python code

```
def moddy(x, y):
    print('id of x = ', id(x))
    while x > y:
        x -= y
        print('id of x = ', id(x))
    return x

a = 5.0
b = 3.0
print('id of a = ', id(a))
R = moddy(a, b)
print('id of a = ', id(a))
```

  Explain why the output of this is

```
id of a = 4423799360
id of x = 4423799360
id of x = 4423799336
id of a = 4423799360
```

  Or phrased differently, why is the last line not `id of a = 4423799336`?

Appendix A

Pop Quiz: Variables, Expressions and Types

This pop quiz tries to test if students have a basic understanding of how variables, expressions, types and floating point representation work in the Python programming language.

The quiz is intended to be done without any means of help or aids. The idea is that the student will be presented with code samples from a Python shell and then the student has to pick the right answer for the output of the presented code based on hand-simulation of the presented code.

If students fail to pass the test then it can be very helpful to use ones Python shell or IDE to try out the code pieces and debug and analyse the code to understand why the answers are wrong.

Question 1: What is the output of running this Python code?

```
a = 1.5
b = a
c = b
a = 1
print(a,b,c)
```

Answer 1:

```
1 1 1
```

Answer 2:

```
1 1.5 1.5
```

Answer 3:

```
1.5 1.5 1
```

Question 2: What is the output of running this Python code?

```
a = 1.5
b = 2
c = 3
type(a+b/c)
```

Answer 1:

Answer 2:

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

Answer 3:

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

Answer 4:

```
<class 'str'>
```

Question 3: What is the output of running this Python code?

```
1/3
```

Answer 1:

```
0.3333333333333333
```

Answer 2:

```
0.0
```

Answer 3:

```
1
```

Answer 4:

```
0
```

Question 4: What is the output of running the Python code?

```
a = 2,5
b = 3
type(a/b)
```

Answer 1:

```
0.8333333333333334
```

Answer 2:

```
Traceback (most recent call last): File "<stdin>", line
    1, in <module> TypeError: unsupported operand type(
   s) for /: 'tuple' and 'int'
```

Answer 3:

```
0
```

Question 5: What is the output of running the Python code?

```
a = 2.5
b = 3
type(a / b)
```

Answer 1:

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

Answer 2:

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

Answer 3:

```
<class 'str'>
```

Question 6: What is the output of running the Python code?

```
(1.0).hex()
```

Answer 1:

```
'0x1.0000000000000p+0'
```

Answer 2:

```
0x1.0000000000000p+0
```

Answer 3:

```
'0x1.0'
```

Question 7: What is the output of running the Python code?

```
float.fromhex('0x1.5000000000000p+0')
```

Answer 1:

```
1.3125
```

Answer 2:

```
1.5
```

Question 8: What is the output of running the Python code?

```
float.fromhex('0x1.0000000000000p+10')
```

Answer 1:

```
1024.0
```

Answer 2:

```
a
```

Answer 3:

```
1.0e10
```

Answer 4:

```
'100000000000.0'
```

Question 9: What is the output of running the Python code?

```
print( int(3.14) )
```

Answer 1:

```
3.0
```

Answer 2:

```
'int(3.14)'
```

Answer 3:

```
3
```

Question 10: What is the output of running the Python code?

```
print( float(3) )
```

Answer 1:

```
3.0
```

Answer 2:

```
3
```

Answer 3:

```
'3.0'
```

Question 11: What is the output of running the Python code?

```
  int(3.75) - 3
```

Answer 1:

```
0
```

Answer 2:

```
0.75
```

Answer 3:

```
0.0
```

Question 12: What is the output of running the Python code?

```
  float(10)/int(4.99)
```

Answer 1:

```
2.5
```

Answer 2:

```
2
```

Answer 3:

```
2.0
```

Question 13: What happens if one tries to run the Python code?

```
A = 3
C = A + B
B = 2
print(C)
```

Answer 1:

```
Traceback (most recent call last): File "<stdin>", line
   1, in <module> NameError: name 'B' is not defined
```

Answer 2:

```
5
```

Question 14: What is the output form running the Python code?

```
  7.5 // 2.0 ** 2.0
```

Answer 1:

```
1.0
```

Answer 2:

```
3164.0625
```

Answer 3:

```
9.0
```

Question 15: What is the output of running the Python code?

```
  (7.5 // 2.0) ** 2.0
```

Answer 1:

```
1.0
```

Answer 2:

```
9.0
```

Answer 3:

```
3164.0625
```

Question 16: What is the output of running the Python code?

```
  50 - 4 * 6 / 4
```

Answer 1:

```
6.5
```

Answer 2:

```
44.0
```

Answer 3:

```
44
```

Question 17: What is the output of running the Python code?

```
  103 % 53
```

Answer 1:

```
50.0
```

Answer 2:

```
53
```

Answer 3:

```
50
```

Answer 4:

```
0.0
```

Question 18: What is the output of running the Python code?

```
a = 4
Print a
```

Answer 1:

```
4.0
```

Answer 2:

```
>>> File "<stdin>", line 1 Print a ^ SyntaxError: invalid
    syntax
```

Answer 3:

```
4
```

Question 19: What is the output from running the Python code?

```
hex(a) + '2'
```

Answer 1:

```
6
```

Answer 2:

```
'0x1.0000000000000p+22'
```

Answer 3:

```
'0x42'
```

Question 20: What is the output of running the Python code?

```
a = 4.0
a.Hex()
```

Answer 1:

```
Traceback (most recent call last): File "<stdin>", line
   1, in <module> TypeError: 'float' object cannot be
   interpreted as an integer
```

Answer 2:

```
Traceback (most recent call last): File "<stdin>", line
   1, in <module> AttributeError: 'float' object has no
   attribute 'Hex'
```

Answer 3:

```
'0x1.0000000000000p+2'
```

Appendix B

Pop Quiz: Control Flow, Functions, Data Structures and Scope

This pop quiz tries to test if students have a basic understanding of how control flow (mostly if and for loops) functions (simple functions recursive function and higher order functions) and data structures (mostly lists) work in the Python programming language. Understanding of scoping is also tested in particular globals.

The quiz is intended to be done without any means of help or aids. The idea is that the student will be presented with code samples from a Python shell and then the student has to pick the right answer for the output of the presented code based on hand-simulation of the presented code.

If students fail to pass the test then it can be very helpful to use ones Python shell or IDE to try out the code pieces and debug and analyse the code to understand why the answers are wrong.

Question 1: What is the output from running this Python code?

```
C = [1, 2, 3]

def mulifier( C ):
    A = list()
    for c in range(len(C)):
        A.append( C[c-1]*2 )
    return A

A = mulifier(C)
print(A)
```

Answer 1:

```
 [6, 2, 4]
```

Answer 2:

```
2, 4, 6
```

Answer 3:

```
[]
```

Question 2: What is the output of running this Python code?

```
C = [1, 2, 3]

def mulifier( C ):
    for c in range(len(C)):
        C[c] = C[c]*2
    return C

mulifier(C)
print(C)
```

Answer 1:

```
[2, 4, 6]
```

Answer 2:

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

Answer 3:

```
[0, 2, 4]
```

Question 3: What is the output of running the Python code?

```
C = 1, 2, 3

def mulifier( C ):
    for c in range(len(C)):
        C[c] = C[c]/2

mulifier(C)
print(C)
```

Answer 1:

```
Traceback (most recent call last): File "...", line 7, in
    <module> mulifier(C) File "...", line 5, in mulifier
   C[c] = C[c]/2 TypeError: 'tuple' object does not
   support item assignment
```

Answer 2:

```
[0.5, 1.0, 1.5]
```

Answer 3:

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

Question 4: What is the output from the Python code?

```
C = [1, 2, 3]

def mulifier( C ):
    for c in range(len(C))
        C[c] = C[c]/2

mulifier(C)
print(C)
```

Answer 1:

```
File "...", line 4 for c in range(len(C)) ^ SyntaxError:
   invalid syntax
```

Answer 2:

```
[0.5, 1.0, 1.5]
```

Answer 3:

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

Question 5: What is the output from the Python code?

```
def sgn(x):
  if x>0:
    return 1;
  elif x<0:
    return -1
  else:
    return 0;
```

```
print(type( sgn(2/3)))
```

Answer 1:

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

Answer 2:

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

Answer 3:

```
<class 'str'>
```

Question 6: What is the output of the Python code?

```
def foo( a, N):
  if N < 1:
    return 0
  n = 1
  x = 1.0
  while n<=N:
    n = n +1
    x = x*a
  return x

print( [foo(2.0, x) for x in range(4)  ] )
```

Answer 1:

```
[0, 2.0, 4.0, 8.0]
```

Answer 2:

```
0, 2.0, 4.0, 8.0
```

Answer 3:

```
[0, 1.0, 2.0, 3.0]
```

Question 7: What is the output of the Python code?

```
def R (A,D):
   if A <  D :
     return A
   return R(A-D,D)
```

```
print( [R(10, x) for x in range(1,10)  ] )
```

Answer 1:

```
 [0, 0, 1, 2, 0, 4, 3, 2, 1]
```

Answer 2:

```
[0.0, 0.0, 1.0, 2.0, 0.0, 4.0, 3.0, 2.0, 1.0]
```

Answer 3:

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

Question 8: What is the output of running this Python code?

```
def has_property(p, i):
    if p==1 or i==p:
        return True
    elif p % i == 0:
        return False
    return has_property(p, i+1)

def test_for_property(p):
    return has_property(p,2)

print(test_for_property(1),test_for_property(2),
   test_for_property(3),test_for_property(4))
```

Answer 1:

```
True True True False
```

Answer 2:

```
True True True True
```

Answer 3:

```
 True False True False
```

Question 9: What is the output of running this Python code?

```
a = 2.0
b = 3
a, b = b, a
print(a,b)
```

Answer 1:

```
 3 2.0
```

Answer 2:

```
2.0 3.0
```

Answer 3:

```
3, 2.0
```

Question 10: What is the output of the Python code?

```
def poly2(a):
    return a**2 - 2*a + 1

def apply(f, X):
    y = []
    for x in X:
        y.append( f(x) )
    return y

X = range(4)
Y = apply(poly2, X)
print(Y)
```

Answer 1:

```
[1, 0, 1, 4]
```

Answer 2:

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

Answer 3:

```
[1.0, 0.0, 1.0, 4.0]
```

Question 11: What is the output of the Python code?

```
def poly2(a):
    return a**2 - 2*a + 1

def apply(f, X):
    y = []
    for x in X:
```

```
        y.append( f(x) )
        return y

X = range(4)
Y = apply(poly2, X)
print(Y)
```

Answer 1:

```
[1]
```

Answer 2:

```
[]
```

Answer 3:

```
[1, 0, 1, 4]
```

Question 12: What is the output of the Python code?

```
def myfun(a):
    return a

def myfun(a,b):
    return a+b

myfun(2)+myfun(2,3)
```

Answer 1:

```
Traceback (most recent call last): File "...", line 7, in
    <module> myfun(2) TypeError: myfun() missing 1
   required positional argument: 'b'
```

Answer 2:

```
7
```

Question 13: What is the output of the Python code?

```
def funny(a):
  global b
  b = a*b
  return b
```

```
def happy(a):
  global b
  b = a+b
  return b

b = 1
funny(1)
funny(2)
funny(3)
happy(1)
happy(1)
print(b)
```

Answer 1:

```
8
```

Answer 2:

```
Traceback (most recent call last): File "", line 9, in <
    module> funny(1) File "", line 3, in funny b = a*b
    NameError: name 'b' is not defined
```

Question 14: What is the output of the Python code?

```
n = 0

def recursive(N):
  global n
  n = n+1
  if N>0 :
     return recursive(N-1) + recursive(N-2)
  return 1

recursive(2)
print(n)
```

Answer 1:

```
5
```

Answer 2:

```
3
```

Answer 3:

```
4
```

Question 15: What is the output of the Python code?

```
n = 1
for x in range(4):
    n = n*x
print(n)
```

Answer 1:

```
0
```

Answer 2:

```
24
```

Answer 3:

```
 6
```

Question 16: What is the output of the Python code?

```
n = 1
for x in range(4)
    n = n*x+1
print(n)
```

Answer 1:

```
File "...", line 3 for x in range(4) ^ SyntaxError:
   invalid syntax
```

Answer 2:

```
16
```

Answer 3:

```
24
```

Question 17: What is the output of the Python code?

```
n = 4
for x in range(n):
    n = x
print(n)
```

Answer 1:

```
3
```

Answer 2:

```
1
```

Answer 3:

```
4
```

Question 18: What is the output the Python code?

```
a = -1
b = 1
c = a*b
if a+b or c<0:
    print('monty')
else:
    print('ytnom')
```

Answer 1:

```
monty
```

Answer 2:

```
ytnom
```

Question 19: What is the output of the Python code?

```
print( 'This is my output ' + '\'a\'' + ' EOL')
```

Answer 1:

```
This is my output 'a' EOL
```

Answer 2:

```
This is my output a EOL
```

Answer 3:

```
This is my output \'a\' EOL
```

Question 20: What is the output of this Python code?

```
print ( format(1.0/3.0, '.3f') )
```

Answer 1:

```
0.333
```

Answer 2:

```
0.33
```

Answer 3:

```
0.3333333333
```

Appendix C

Pop Quiz: Matrices and Recursion

This pop quiz tries to test if students have a basic understanding of how matrices from NumPy and plot from Matplotlib works.

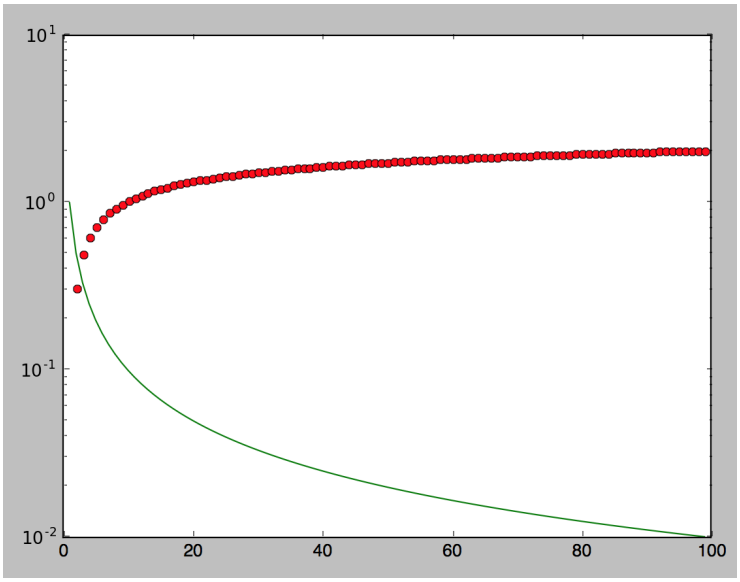
The quiz is intended to be done without any means of help or aids. The idea is that the student will be presented with code samples from a Python shell and then the student has to pick the right answer for the output of the presented code based on hand-simulation of the presented code.

If students fail to pass the test then it can be very helpful to use ones Python shell or IDE to try out the code pieces and debug and analyse the code to understand why the answers are wrong.

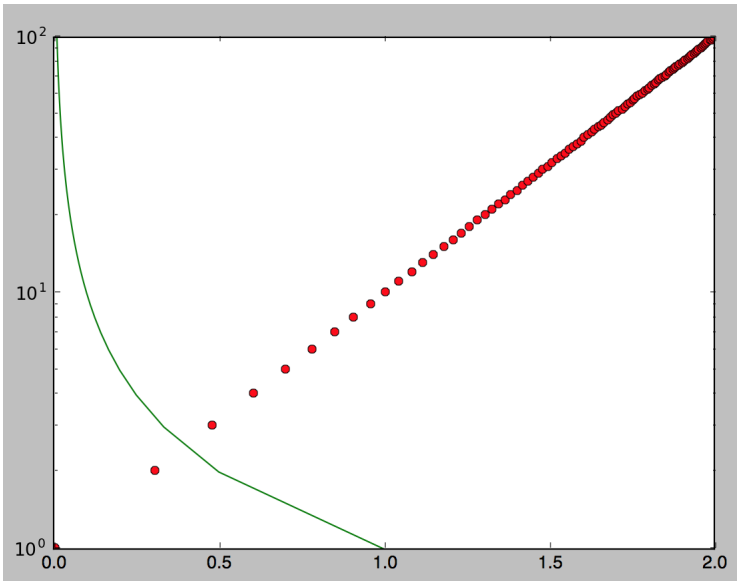
Question 1: What plot is generated with the Python code?

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

x = range(1,100)
y1 = [ math.log10(z) for z in x ]
y2 = [ 1.0/z for z in x ]
plt.semilogy(x,y1,'go')
plt.semilogy(x,y2,'r-')
plt.show()
```

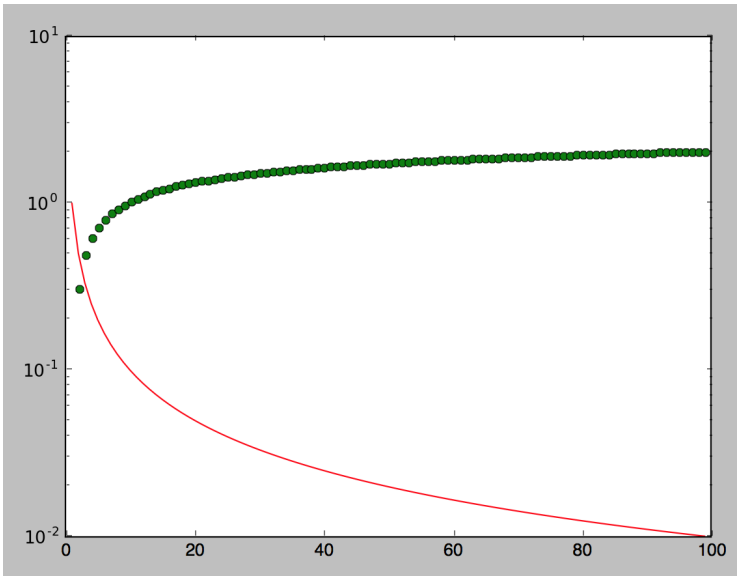
Answer 1:



Answer 2:



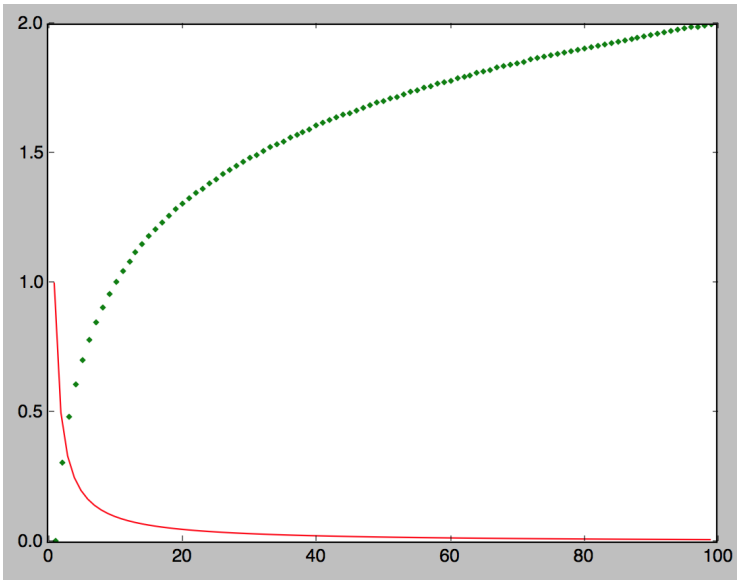
Answer 3:



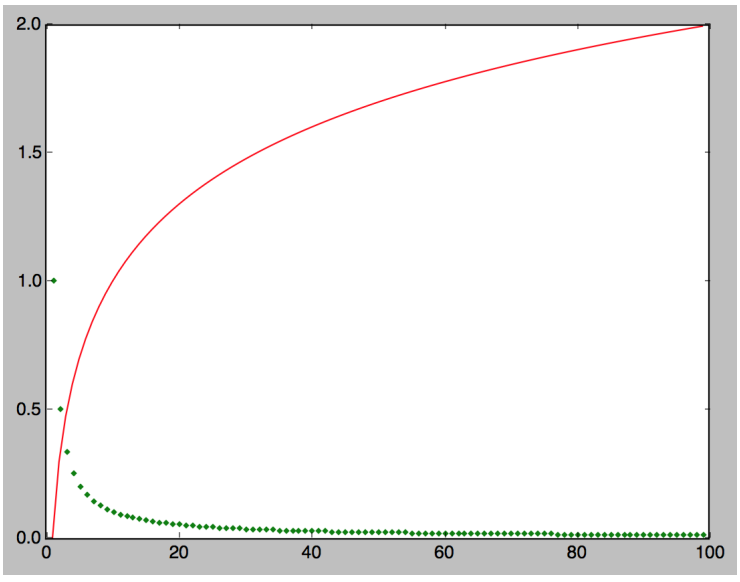
Question 2: What plot is generated from the Python code?

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

x = range(1,100)
y1 = [ math.log10(z) for z in x ]
y2 = [ 1.0/z for z in x ]
plt.plot(x,y1,'g.')
plt.plot(x,y2,'r-')
plt.show()
```

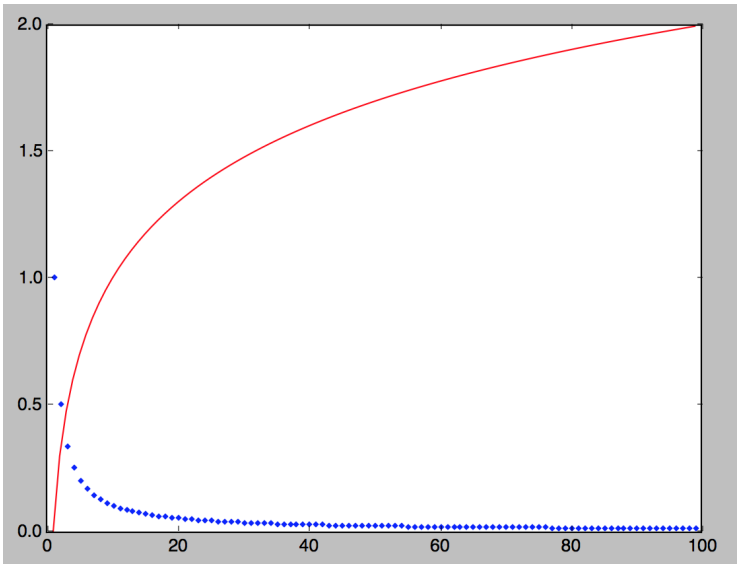
Answer 1:



Answer 2:



Answer 3:



Question 3: What is the output of the Python code?

```
import numpy as np

def make_it(n):
    A = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            A[i,j] = 1.0 / (i+j+1)
    return A

print(make_it(3))
```

Answer 1:

```
[[ 1.  0.5  0.33333333]
 [ 0.5  0.33333333  0.25 ]
 [ 0.33333333  0.25  0.2 ]]
```

Answer 2:

```
[[ 0  0  0]
 [ 0  0  0 ]
 [ 0  0  0 ]]
```

Question 4: What is the output of the Python code?

```
import numpy as np

def make_it(B : np.ndarray):
    (m,n) = B.shape
    N = max( B.shape )
    A = np.arange(m, None, None, np.float64)
    for i in range(N):
        A[i] = B[i,i]
    return A

B = np.random.rand(3,3)
print(make_it(B))
```

Answer 1:

```
[ 0.61877696 0.93906928 0.09262115]
```

Answer 2:

```
[[ 0.61877696 0.06447968 0.51443369]
 [ 0.32877109 0.93906928 0.60717936]
 [ 0.24531295 0.06240987 0.09262115]]
```

Answer 3:

```
[[ 0.61877696 0.0 0.0]
 [ 0.0 0.93906928 0.0]
 [ 0.0 0.0 0.09262115]]
```

Question 5: What is the output of the Python code?

```
A = [[0, 1, 0], [1, 0, 0], [0, 1, 0]]
print(A[0][1] + A[1][0] + A[2][1])
```

Answer 1:

```
  3
```

Answer 2:

```
  3.0
```

Answer 3:

```
  2.0
```

Answer 4:

```
  2
```

Question 6: What is the output of the Python program?

```
import numpy as np


A = [[0, 1, 0], [1, 0, 0], [0, 1, 0]]
B = np.array(A)
print(B[0, 1] + B[0, 1] + B[0, 1])
```

Answer 1:

```
  3
```

Answer 2:

```
  3.0
```

Answer 3:

```
  2.0
```

Answer 4:

```
  2
```

Question 7: What is the output of the Python code?

```
import numpy as np


A = [[0, 1, 0], [1, 0, 0], [0, 1, 0]]
print(A[0, 1] + A[0, 1] + A[0, 1])
```

Answer 1:

```
 Traceback (most recent call last): File "...", line 5,
    in <module> print(A[0, 1] + A[0, 1] + A[0, 1])
    TypeError: list indices must be integers or slices,
    not tuple
```

Answer 1:

```
3
```

Answer 2:

```
2
```

Answer 3:

```
3.0
```

Question 8: What is the output of the Python code?

```
import numpy as np


A = [[0, 1, 0], [1, 0, 0], [0, 1, 0]]
B = np.array(A)
print(B[0][1] + B[1][0] + B[2][1])
```

Answer 1:

```
3
```

Answer 2:

```
3.0
```

Answer 3:

```
2.0
```

Answer 4:

```
2
```

Question 9: What is the output of the Python code?

```
import numpy as np


A = [[0, 1, 0], [1, 0, 0], [0, 1, 0]]
B = np.array(A)
print(B[1:3, 1:3])
```

Answer 1:

```
[[0 0]
 [1 0]]
```

Answer 2:

```
[[0 1 0]
 [1 0 0]
 [0 1 0]]
```

Answer 3:

```
[[0  0]
 [0  0]]
```

Question 10: What is the output of the Python code?

```
import numpy as np


A = [[0, 1, 0], [1, 0, 0], [0, 1, 0]]
B = np.array(A)
print(B[1, :])
```

Answer 1:

```
 [1 0 0]
```

Answer 2:

```
 [0 1 0]
```

Answer 3:

```
 [1 0 1]
```

Answer 4:

```
 [0 1 0]
```

Question 11: What is the output of the Python code?

```
import numpy as np


A = [[0, 1, 0], [1, 0, 0], [0, 1, 0]]
B = np.array(A)
idx = [0, 2]
print(B[idx])
```

Answer 1:

```
[[0 1 0]
 [0 1 0]]
```

Answer 2:

```
[[0 1 0]
 [1 0 1]]
```

Answer 3:

```
[[0 1 0]
 [0 0 1]]
```

Question 12: What is the output of the Python code?

```
import numpy as np


A = [[0, 1, 0], [1, 0, 0], [0, 1, 0]]
B = np.array(A)
idx = [0, 2]
print(B[idx, 1])
```

Answer 1:

```
 [1 1]
```

Answer 2:

```
 [0 0]
```

Answer 3:

```
 [1 0]
```

Answer 4:

```
 [0 1]
```

Question 13: What is the output of the Python code?

```
import numpy as np


A = [[0, 1, 0], [1, 0, 0], [0, 1, 0]]
B = np.array(A)
print( B[1]*2 + B[0] )
```

Answer 1:

```
[2 1 0]
```

Answer 2:

```
[0 1 2]
```

Answer 3:

```
[1 0 2]
```

Question 14: What is the output of the Python code?

```
import numpy as np

A = np.array([[0, 1, 0], [1, 0, 0], [0, 0, 1]])
print( A[3] )
```

Answer 1:

```
Traceback (most recent call last): File "...", line 4, in
    <module> print( A[3] ) IndexError: index 3 is out of
   bounds for axis 0 with size 3
```

Answer 2:

```
[0 0 1]
```

Answer 3:

```
[0 1 0]
```

Question 15: What is the output of the Python code?

```
import numpy as np

A = np.array([[0.0, 1.0, 0.0], [1.0, 0.0, 0.0], [0.0,
    0.0, 1.0]])
B = np.array([[1.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0,
    1.0, 0.0]])
C = 2*A + B;
print(C)
```

Answer 1:

```
[[ 1. 2. 0.]
 [ 2. 0. 1.]
 [ 0. 1. 2.]]
```

Answer 2:

```
[[ 2. 1. 0.]
 [ 1. 0. 1.]
 [ 0. 2. 1.]]
```

Answer 3:

```
[[ 0. 2. 0.]
 [ 2. 0. 0.]
 [ 0. 0. 2.]]
```

Question 16: What is the output of the Python code?

```
import numpy as np


x = np.array([0.0, 1.0, 0.0])
A = np.array([[0.0, 1.0, 0.0], [1.0, 0.0, 0.0], [0.0,
    0.0, 1.0]])
print(A.dot(x))
```

Answer 1:

```
[ 1. 0. 0.]
```

Answer 2:

```
[[ 0. 1. 0.]
 [ 0. 0. 0.]
 [ 0. 0. 0.]]
```

Answer 3:

```
[[ 0. 1. 0.]
 [ 1. 0. 0.]
 [ 0. 0. 1.]]
```

Question 17: What is the output of the Python code?

```
import numpy as np
```

```
x = np.array([0.0, 1.0, 0.0])
A = np.array([[0.0, 1.0, 0.0], [1.0, 0.0, 0.0], [0.0,
    0.0, 1.0]])
print(A*x)
```

Answer 1:

```
[[ 0. 1. 0.]
 [ 0. 0. 0.]
 [ 0. 0. 0.]]
```

Answer 2:

```
[ 0. 1. 0.]
```

Answer 3:

```
[[ 0. 1. 0.]
 [ 1. 0. 0.]
 [ 0. 0. 1.]]
```

Question 18: What is the output of the Python code?

```
import numpy as np


x = np.arange(3.0,0.0,-1.0)
print(x)
```

Answer 1:

```
 [ 3. 2. 1.]
```

Answer 2:

```
 [ 3 2 1]
```

Answer 3:

```
 [ 1. 2. 3.]
```

Answer 4:

```
 [ 3. 2. 1. 0.]
```

Question 19: What is the output of the Python code?

```
import numpy as np


A = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]])
A[1,:] =  A[1,:] - A[0,:] * 4.0
print(A)
```

Answer 1:

```
[[ 1.  2.  3.]
 [ 0. -3. -6.]
 [ 7.  8.  9.]]
```

Answer 2:

```
[[ 1.  2.  3.]
 [ 0.  3.  6.]
 [ 7.  8.  9.]]
```

Answer 3:

```
[[ 1. -2.  3.]
 [ 4. -11.  6.]
 [ 7. -20.  9.]]
```

Question 20: What is the output of the Python code?

```
import numpy as np


A = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]])
print(A.transpose())
```

Answer 1:

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

Answer 2:

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