CCPS 109 Computer Science I

(**outline and CMF by Ilkka Kokkarinen, July 19, 2020**)

Outline

This course is an introduction to computer science and programming in Python 3 programming language. Its topics are broken down into twelve separate modules that divide this course into three separate parts of four modules each. Each part has its own distinct spirit and style of approach that distinguishes it from the other two parts.

Part One: Core Language

The first part introduces the core Python language and its central standard library modules, in principle sufficient to solve all problems in the graded labs. Interactive exercises amply available at the CodingBat Python training site drill in the syntax of Python until students reach the stage where the language works with them as aid for solving problems.

1. Python as a scriptable calculator
2. Functions that make decisions
3. Sequence types string, list and tuple
4. Sequence iteration

Part Two: Advanced Techniques

Having unleashed the students to work on the graded labs of the course until the end, the second part of the course introduces higher-level techniques to express computations, and showcases non-trivial examples of interesting problems where the power of Python shines.

5. General repetition
6. Educational example programs
7. Lazy sequences
8. Recursion

Part Three: Python In The Real World

At this point, the students should be hard at work completing their chosen lab problems to reach a higher course grade. This third part is therefore more relaxed and theoretical in nature. It introduces powerful mechanisms that make the language even more expressive, along with some important and universal ideas of computer science that are independent of any particular programming language. These ideas will then be expanded further in the course CCPS 209 Computer Science II and the courses beyond that.

9. String processing
10. Numerical computation with numpy
11. Classes and objects
12. Spillover of interesting examples that were skipped earlier

Materials

The public GitHub repository `ikokkari/PythonExamples` always contains the latest versions of lecture notes (as PDF files) and all example programs (as Python scripts) used in this course.

All software needed to do everything in this course is free to download and use in both Windows and Apple computers. Students can either (**1**) download and install the Anaconda package manager and use its Spyder environment to write, run and test their programs on their local computers, or (**2**) create themselves a free account to the cloud-based repl.it programming environment, and import the course repositories into their "repls" to work on. Especially students whose aging laptop and desktop computers lie in the lower end of the speed spectrum should find working on the cloud far less frustrating as modern high-performance cloud servers perform the grunt work of their actual computations.

Labs

The first four modules of this course use the problems offered in the free CodingBat Python online programming environment to drill in the basics of the core Python language. The simple yet excellent and educational problems on that site allow beginning students to quickly pinpoint what their functions are doing right and wrong, and what issues still remain to fix. When combined with another excellent online utility of Python Tutor that allows these functions to be animated and stepped through forwards and backwards, their educational powers mutually reinforce each other.

After acquiring proficiency with the language with these two free training sites so that the students can start applying this knowledge in solving problems with the language as their ally instead of the opponent, the lab problems in fifth module onwards have been chosen from the problem collection "109 Python Problems for CCPS 109". The public GitHub repository `ikokkari/PythonProblems` always contains the latest versions of the automated tester script and the necessary data files.

Grading

The course grade is based solely on how many lab problems from the collection "109 Python Problems for CCPS 109" the student successfully solves. Nothing else can affect the course grade in either direction. These labs will be submitted all at once, at the same time at the end of the course. The functions written in these labs should be written inside the Python source file `labs109.py` for the automated tester script `tester109.py` to find and evaluate.

The course grade granted at the end of the course is computed with the following formula. To earn the passing grade of 50% (D minus), the student must successfully complete ten labs. After that, every additional lab that they complete adds one more point to their course grade. Therefore to get the highest possible course grade of 90% (A plus), the student has to complete fifty lab problems out of the 109+ problems offered.

Students are expected to create the solutions to these problems as their own individual work. However, they are allowed to look for advice and guidance by searching any and all existing material in online articles and programming forums such as Stack Overflow. (Whatever problem you have in coding, thousands of others have had that exact same problem before, the more so the more universal and general that particular issue happens to be.) However, until the grades for this course have been tallied, students are not allowed to post any new questions concerning these lab problems anywhere.

As long as no code is being copied via direct copy-paste or going through the eyes, brains and fingers of the student so that this discussion stays at the level of ideas, students may freely discuss these problems with anybody both inside and outside this course, with no fear of any repercussions from such activity. After all, this is how all learning works. Except for the things that you happen to discover by yourself, all knowledge comes from somebody else. Whether that somebody is your formal instructor or some other person has no epistemological consequences on that.

Finally, should some students get stuck with some problem that they have given their honest effort to but somehow can't seem to set it right and eliminate the bugs (and this will eventually happen to everyone regardless of their skill level, since everyone has mental blind spots *somewhere*), it is acceptable to occasionally, but not as a general rule, to show the code to somebody for a second pair of eyeballs that will hopefully spot something that the original student missed.

That other person may also be the instructor of the course, as long as this does not become a regular habit and the student gives the impression of having first given the problem a good college try. The best way to do this is most likely by email, as old-fashioned as that form of media might seem to us now. In case of some difficult corner case bug that was not caught and revealed by the first 300 recorded test cases for that problem but shows up only as a checksum discrepancy, the instructor can put the `discrepancy` function inside the `tester109.py` script to good use and run both functions for all test cases, and find out exactly where the student function and the instructor's private model solution disagree. Such happenstances have historically revealed genuine bugs in the instructor's private model solutions, although these get ever more rare as the course material matures. (But some bugs will always be lurking, no matter how strenuously we try to stomp them down!)

Module 1: Python as a scriptable calculator

Introduction

Python 3 is a modern programming language whose core syntax is simple enough to learn in five weeks, and yet behind this simple syntax lies an entire universe of tremendous power. This module shows you how to set up this system to start your exploration of programming and computer science by first using Python as a scriptable calculator that can operate on not just numbers, but on characters and text.

Topics

1. Launching the Python interactive environment either locally or on the cloud.
2. Entering arithmetic expressions inside the Python REPL.
3. Defining names that are associated with values, and using these names inside later expressions.
4. Basic operations of integer arithmetic, with a suggestion to avoid floating point numbers in your work if possible.
5. Importing new functions from the standard library modules and using them in your computations.
6. Representing text as string literals in Python.
7. Composing complex text strings from simpler pieces with the f-string mechanism.

Learning objectives

After this module, students can launch the Python interactive environment either on their local computer (Anaconda/Spyder) or on the cloud (repl.it). They know how to evaluate arbitrary arithmetic expressions in the REPL window, and how integer arithmetic operators work in Python. They can create names that remember values that were assigned to them, and use these names in later expressions to refer to results that were calculated earlier. They can open, edit, save and execute scripts in some Python IDE, and can make these scripts print results for the user to see. They can define text literals as ordinary and f-strings in Python. They can import new functions and data types from the standard library into their scripts, and know where to go in the official Python documentation to learn how to use these functions in their code.

Readings

1. "Python Lecture Notes", sections "1.1. Expressions", "1.2. Scripts", "1.3. Naming things", "1.4. Assignment operator", "1.5. Types", "1.6 Importing functionality from modules", "1.7. Some useful standard library modules".

2. GitHub repository `ikokkari/PythonExamples`: `first.py`
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "Python 1-1: Overview", "Python 1-2: And God created integers", "Python 1-3: Give me a name instead of a number", "Python 1-4: Executable text".

Activities

The purpose of this first module is to get acquainted with the Python Interactive Development Environment (IDE) in which the Python programs are created and executed. Instead of an old-school installation straight from the source www.python.org, students who choose to work locally on their own computers can download and install the **Anaconda** package manager, and do their work inside the **Spyder** IDE there. Alternatively, students may choose to work on a cloud-based Python environment `repl.it` that can be accessed anywhere via a web browser. All examples, exercises and lab problems in this course can be successfully completed in either environment.

Option 1 (**Anaconda/Spyder locally**): Download and install Anaconda Individual Edition. While this behemoth is installing, you might as well use the waiting time to download the repositories ikokkari/PythonExamples and ikokkari/PythonProblems as zip archives (use the green Clone button of the recently updated GitHub interface), and unzip these archives into folders `PythonExamples` and `PythonProblems` into whatever place in your file system you normally store your data files. Start up Anaconda, and from there start Spyder. (In the first run, both of these might be a bit sluggish to start up while they initialize some stuff, but they will get there and work faster in the future.) Use `File->Open...` to open the script `first.py` into an editor window. Press the green play button to execute the script whose editor window is currently active. Observe the results in the console window. When this script asks you to enter your name and age, give it something so that the script can proceed.

Option 2 (**repl.it on the cloud**): Create yourself a free account on the site repl.it under which your files will be stored under. Click the black "**import repo**" button on the top right corner, enter the URL `https://github.com/ikokkari/PythonExamples` into the dialog box and click "**Import from GitHub**". You should now see a Python 3 "repl" named `PythonExamples` under your account that contains all the example programs from this course. Do the same for the URL `https://github.com/ikokkari/PythonProblems` to import the automated testing environment for the lab problems for this course. Then go to your `PythonExamples` repl and press the green "Run" button to execute the script `first.py`. When this first script asks you to enter your name and age, give it something so that the script can proceed. To make the "Run" button execute some other script, you have to edit the second line of the file `.replit` to read `run="python3 scriptname.py"` where you should replace `scriptname` with the actual name of the script that you want to run.

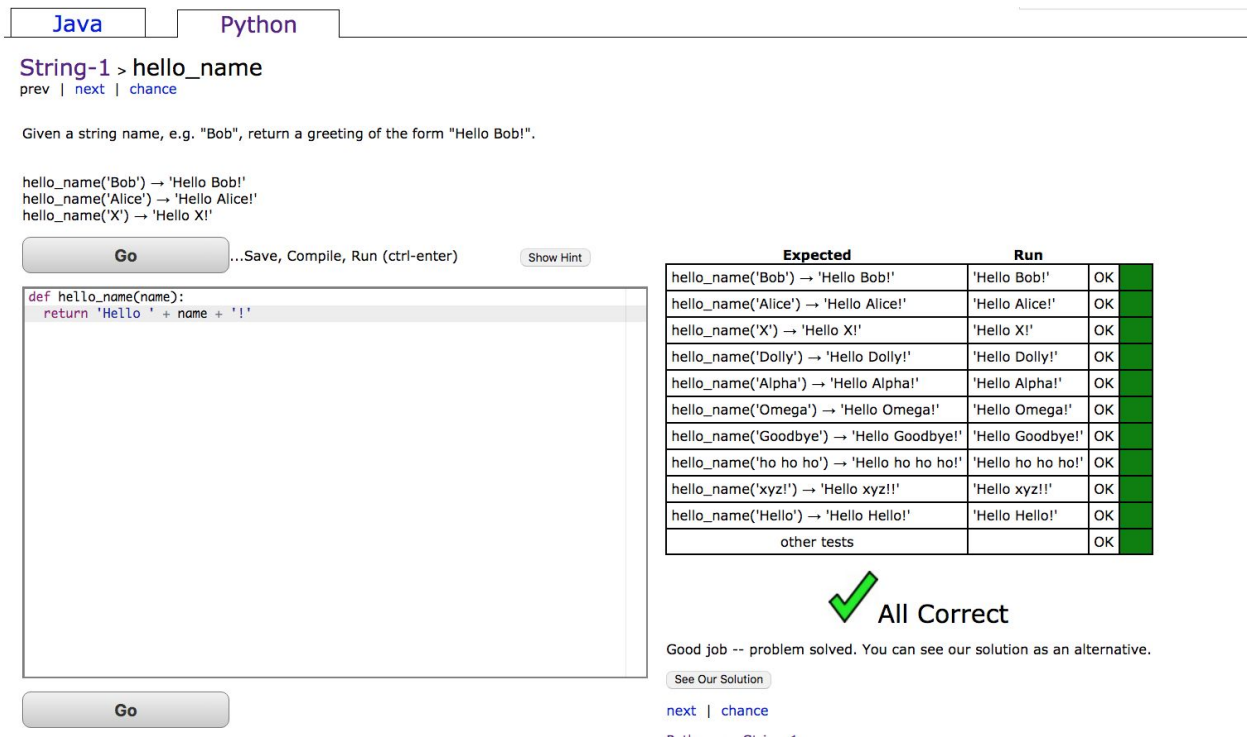
Lab problems

During the first five modules, this course uses the excellent CodingBat Python interactive training site. This site contains lots of small programming problems whose purpose is to drill in the core Python language that we learn during the first five weeks. From the sixth week onwards, we move on to apply this knowledge to solve problems from the collection "109 Python Problems for CCPS 109".

Every CodingBat problem is solved as a script that defines exactly one function. Since we will be writing functions next week onwards, this week is devoted to setting up the programming environment (either Spyder or repl.it) and learning to use the CodingBat interactive coding environment. Create yourself a free account at CodingBat so that it will remember your solved problems and the solutions that are works in progress. Then, in the collection "String-1", go to the problem `hello_name`. The editor pane already contains the signature of the function that you should always keep exactly as it was given, since otherwise the automated testers. Under the function signature, type one line of text (please don't just copy-paste from here)

```
return 'Hello' + name + '!'
```

indented two spaces to denote that this statement is inside the function `hello_name` that is being defined in this script. Once you have done this, press the "Go" for CodingBat to grade your function by giving it a bunch of test cases and verifying that your function returns the correct result for each. You should see something resembling the screenshot below:



Next to your editor pane, CodingBat shows a table with all the test cases in the "Expected" column, with the individual results from your function in the "Run" column. The green box means that your function returned what it was supposed to return for the test case in that row. If your function had returned anything other than the expected result, that row would then have a red box at the end to show you the existence of a bug in your code. (Don't worry, you will write your first bug soon enough to see plenty of red!)

One caveat of using CodingBat, at least the moment of writing this, is that the server uses Python 2 internally, so the nice things of Python 3 that we learn in this course do not work in these scripts. For example, trying to implement `hello_name` using an **f-string** will only result in a syntax error. Another caveat is that many problems have been adapted from the Java version of the site, and the terminology has not been edited to correspond to Python. For example, whenever you see something referred to as an "array", just mentally substitute the word "list" in its place.

Module 2: Functions that make decisions

Introduction

The simple two-way decision is the foundation of all computation from which all other computational operations are built from bottom up, all the way down to the logic gates of the computer processor up to the highest levels of abstraction inside Python. The existence of forks in the road that the execution of a program has to choose between makes it possible for the outcome to be different for different inputs given to that program, instead of every computation leading to Rome regardless of these inputs.

Topics

1. Defining your own functions that compute and return a result that depends on the function arguments.
2. Purpose of whitespace to indicate nesting of statements inside a Python script.
3. Two-way decisions to branch the execution of a script into two mutually exclusive branches based on the truth value of some condition.
4. Equality and order comparisons in Python.
5. Building up larger decision trees from two-way decisions via nesting and laddering.
6. Expressing more complex conditions with logical connectives.

Learning objectives

After this module, students are able to first define functions in their scripts, and then call these functions properly later in the code using different argument values. They understand the difference between the function printing the result versus returning it. They can explain the connection of the whitespace indentation that starts a Python statement and the nesting of that statement inside the Python script. They can make their functions perform simple two-way if-else statements, and combine these two-way decisions into more complex decision trees by nesting them and creating if-else ladders. They recognize the core logical connectives `and`, `or` and `not`, and can explain how the truth value of a complex condition built with these is determined by the truth values of its subconditions.

Readings

1. "Python Lecture Notes", sections "2.1. Defining your own functions", "2.2. Simple two-way decisions", "2.3. Outcomes of equality", "2.4. Complex conditions", "2.5. If-else ladders".
2. GitHub repository ikokkari/PythonExamples: `conditions.py` `timedemo.py`
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "Python 2-1: Verb every word to me", "Python 2-2: Forks on the road", "Python 2-3: The Galton board of reality", "Python 2-4: A Skinner box for humans, localized entirely within your computer", "Python 2-5: Once in a century".

Activities

Study the example script `timedemo` as an example of how to do computations with calendar dates and times with the `datetime` module in the standard library. Use this knowledge to adapt the script to find out what day in the future you will be exactly twice as old as you are today. (Compute how many days you have been alive so far, and then add that many days to the current date.) Then, find out which day of the week you were born in.

Lab problems

Since the humble two-way decision is the fundamental building block of all computations, having this operation at our disposal allows us, in principle if not always in practice, to solve any computational problem as a Python function... provided that the problem has an upper limit to how large its input can be, so that we can encode all possibilities into a decision tree implicitly executed by the statements of the function. This is especially the case when the arguments given to the function are truth values so that each argument is either `True` or `False`, with no third alternative.

The problems in the CodingBat Python sections Warmup-1, Logic-1 and Logic-2 will have you write functions that cover all possible situations of their argument values with a small handful of decisions properly nested and laddered. **Your task this week is to solve at least ten problems of your choice from these three sections**. This ought to drill the Python syntax (including that annoying redundant colon after every condition, and that `==` and `=` are different things, as are also `//` and `/`) into your fingertips, while at the same time your mind gets accustomed to the task of breaking down a complex decision into a decision tree made of simple two-way decisions.

When comparing your solutions to those of other students, remember that whenever you come up with one way to organize a decision tree, there always would have been many other result-equivalent ways to do so.

Module 3: Sequence types string, list and tuple

Introduction

There are only so many interesting questions that we can ask about a single integer or other scalar value. However, in the current era of terabytes, even everyday computers that not only the five richest kings of Europe but also us commoners and other normos can afford to own, can process sequences that consist of millions, even billions, of such scalar values of data. Sequences allow our Python programs aggregate scalar values together under a single name, and the powerful polymorphic functions and operators on sequences can handle all types of sequences in a uniform fashion.

Topics

1. Embedding special characters inside string literals with escape sequences.
2. The slicing operator for Python sequences, as demonstrated by slicing pieces from existing text strings.
3. Strings as immutable sequences of Unicode characters.
4. The structurally flexible and heterogeneous lists in Python.
5. Constructing new list objects from existing sequences with list comprehensions.

Learning objectives

After this module, students know how to make up text string literals that can contain arbitrary Unicode characters. They can slice and dice text strings forwards and backwards with the slicing operator `[]` of Python. Their ability to represent data has expanded from scalar values into sequences of arbitrary values as Python lists. Students can take an existing list and transform it into a new list with list comprehensions.

Readings

1. "Python Lecture Notes", sections "3.1. String literals from Unicode characters", "3.2. String literals from arbitrary pieces", "3.3. String slicing", "3.4. Lists made of things" and "3.5. List comprehensions". (Students looking for extra material can also study the section "3.6. Nested list comprehensions", but this topic is not needed in this course.)
2. GitHub repository ikokkari/PythonExamples: `listdemo.py` `stringdemo.py` `comprehensions.py` `tuples.py`
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "Python 3-1: It slices, it dices, and so much more!", "Python 3-2: Pretty objects all in a row", "Python 3-3: All sequences eager and lazy", "Python 3-4: Definition of terse"

Activities

When a function that is more complex than a couple of lines does not return the answers you expect, it is often convenient to be able to **step through the execution** to see exactly what is going on in each step. We shall later learn how to use such **debuggers** built in Spyder and repl.it, but until then, students should check out the excellent Python Tutor for this purpose. Copy-paste one of the functions that you have written earlier into the editor pane on that page, followed by a line in which you call that function for some argument values. To see a step-by-step animation of the function execution, press the button "Visualize execution" to take you to the screen where you can step through your execution both forward and backward. You can see exactly the values associated with each name at each step during this execution, which will be useful in finding out the exact spot where the computation does something unexpected for some failing test case.

Lab problems

To practice handling sequences of data and learn to understand how these sequences can be freely sliced into smaller pieces and concatenated into larger pieces, CodingBat sections Warmup-2, String-1 and List-1 contain a host of excellent problems ready to be solved with the Python language learned in this week's module. Your task, same as last week, again is to drill in the language syntax into your fingertips while your mind adapts to think about entire sequences of data of unlimited length instead of mere individual scalar values. Therefore, **this week you should again complete at least ten, but preferably more, problems chosen from these three sections.**

Module 4: Sequence iteration

Introduction

In computing, we are accustomed to the idea that duplication of bytes by the billions is essentially free. The same principle also applies inside the computer program. Once you have discovered a way to solve some problem for one element, you can place this solution inside some repetition control structure that then solves it for the entire sequence of such elements. This module introduces the basic for-loop control structure to do this very thing.

Many problems can be expressed in, or at least converted to the form of some small local operation being repeatedly performed for the elements of some cleverly chosen existing sequence. Especially when asked to do something five times, the computer program does the exact same thing as a human who keeps a running count going "one, two, three, four, five", thereby unknowingly iterating through a simple arithmetic progression that turns out to be isomorphic to the actual problem that exists outside the mind of that particular human. That, and the computer program usually starts counting up from zero because that turns out to be mathematically convenient. However, a computer can count up to a billion much faster than us humans, even though just like us, it does not need to remember all the numbers that it has previously processed, but only the number that it is currently at.

Module topics

1. Iterating through the elements of an arbitrary sequence with the same for-loop.
2. Lazy sequences that produce their elements one at the time on command.
3. The extremely memory-efficient `range` sequences for the commonly needed arithmetic progressions of integers.
4. Files as sequences of lines, each line a sequence of characters.
5. Tuples as compact immutable sequences of a handful of elements.
6. Sets and dictionaries to remember what the function has seen and done.

Module learning objectives

After this module, students can write for-loops to iterate through the elements of the given sequence, and have the body of the for-loop perform the same computation for each element. This computation can involve updating the local state of the function such as tallying up the sum or the maximum of the elements seen so far. They know how to represent arithmetic progressions as `range` objects, and understand why such a `range` requires far less memory to store compared to an eager `list` that contains its elements explicitly. They know the difference between mutable lists and immutable tuples, and which operations are possible based on that.

Students can create a fresh new `set` and use it to store the values encountered during the function execution, and can explain why asking a set whether it currently contains some element is much faster than asking the same question from a list.

Readings

1. "Python Lecture Notes", sections "4.1. Processing items individually with for-loops", "4.2. Iterator objects", "4.3. Integer sequences as range objects", "4.4. Files as sequences", "4.5. Tuples as immutable sequences", "4.7. Members only", "4.8. From keys to values". (Students looking for extra reading can also check out sections "4.6. Breaking tuples into components" and "4.9. Assorted list, set and dictionary goodies", but these topics are not required in this course.)
2. GitHub repository ikokkari/PythonExamples: `defdemo.py` `setsanddicts.py` `wordcount.py`
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "Python 4.1: For your elements only", "Python 4-2: Memory of a goldfish", "Python 4-3: Frank Benford, gumshoe who works scale", "Python 4-4: Those who can't remember the past...", "Python 4-5: Memories of lit"

Lab problems

The ability to make your function perform the same operation for every element of an arbitrary sequence turns out to be quite a lot more powerful than it might initially seem. In fact, we now have enough Python language backing us up that in principle, we could already express literally any computation that could be realized inside our entire physical universe! However, this previous sentence should be understood in the same sense as the sentence "Once you learn all the letters of the alphabet and the punctuation and other characters that can appear inside a manuscript, you can write any book whatsoever", which is true only in a very special theoretical sense, but not in any practical sense.

After such an energetic start, we can confidently face the problems in the CodingBat Python sections String-2 and List-2. These problems require the use of some for-loop to process the elements of the given string or a list. However, note that at least at the time of writing this document, many of these problems have obviously been copy-pasted from the older and larger CodingBat Java site in that they talk about "arrays", which Python does not have. For the purposes of every such question, whenever you see the word "array", simply mentally substitute the word "list" in its place.

Module 5: General repetition

Introduction

In many computational problems we have a straight and narrow road ahead of us, and the goal that we are trying to achieve is promised to be somewhere along that road. Often, we know how far we have to go the moment the moment we start the first step, as in iterating through sequences that have a known beginning and a known end. However, in general it is not possible to know how many steps we need to execute until the goal state is reached.

Instead of having to decide in the beginning how many times we are going to repeat the body of the loop, a while-loop breaks this potentially infinite journey into local two-way decisions of the form "Are we there yet?" While we have not reached the goal state, we execute the body of the while-loop once more. If the goal is somewhere along the execution path, this mechanism will eventually get us there. It is also important to realize that sometimes we are already standing on the goal to begin with, and taking even a step away from it would give the whole game away. Furthermore, you must keep going so that each step takes you forward somehow so that you are not just marching forever in place. Even so, there exist problems where the goal simply does not exist anywhere in the program execution path, thus making that journey to be doomed to go on forever in an infinite loop.

Topics

1. Using a while-loop to repeat the body of statements until the desired goal is reached.
2. Finite and infinite loops.
3. The optional else-block after a Python loop.
4. Binary search to narrow down even an astronomical number of possibilities to just one.
5. Breaking out of the execution of a loop, or just skipping directly to the next round.

Module learning objectives

After this module, students know the syntax and use of the general repetition structure of a while-loop in Python for situations where it is not known in the beginning how many times the loop body should be executed to achieve the goal of the problem. They know that an infinite loop is an actual possibility, just like doing nothing is another possibility that we don't often think about in real world systems biased towards action. They recognize the optional else-block after a Python loop and its behaviour in functions written by other people. They know the binary search algorithm when applied to searching inside a sorted random access sequence, and can use the implementation in the Python standard library. They know the effect of the `break` and `continue` statements inside the loop and can use them in their own functions, but with the

knowledge that doing so is never necessary and is in fact impure in the structured programming spirit.

Readings

1. "Python Lecture Notes", sections "5.1. Unlimited repetition", "5.2. While not there yet, take one more step towards it", "5.3. Binary search", "5.4. The power of repeated halving", "5.5. Nested loops".
2. GitHub repository `ikokkari/PythonExamples`: `mathproblems.py` `cardproblems.py` `hangman.py`
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "Python 5-1: Are we there yet?", "Python 5-2: Straight and narrow road", "Python 5-3: A handful of symbolism", "Python 5-4: Cold as the cards lie", "Python 5-5: A crooked little pair"

Activities

With all the language in our fingertips that we need to solve computational problems, the time has come to thank CodingBat for its service. This week, we move on to the graded labs of this course in the repository `ikokkari/PythonProblems` that we last saw during the first week of this course, either in Spyder or `repl.it` depending on your choice of working environment, and see how the actual labs of this course are executed and tested.

Run the automated tester script `tester109.py` to execute the tests for all the functions currently implemented in the `labs109.py` file. To ensure that the tester script is working, this file already contains a working implementation `ryerson_letter_grade` of the first problem "Ryerson Letter Grade" from the collection "109 Python Problems for CCPS 109". The tester should finish quickly and you should see the outcome in the repl window. (The first line and the rounded running time of the test may be slightly different.)

```
109 Python Problems tester, June 10, 2020, Ilkka Kokkarinen.
ryerson_letter_grade: Success in 0.001 seconds.
1 out of 1 functions (of 109 possible) work.
```

Every lab problem that you solve in this course consists of exactly one function. This function must have the exact signature required in the problem specification. All functions must be written in the file `labs109.py` for the automated tester script to be able to find them. Even if you keep these functions in some other files during your development, you need to copy those functions to `labs109.py` to be able to run the test. Most problems can be solved with just one function, but it is okay to write additional helper functions either inside the lab functions or on the top level. Some helper functions will even be useful in several lab problems, not just the one problem that they were originally developed for. The very best helper functions, of course, are so universally useful for everyone that they have long since become part of the Python standard

library. Instead of redundantly reinventing that same wheel once again, your functions are allowed to use anything in the Python 3.8 standard library modules.

Edit the function `ryerson_letter_grade` to return `"Fail"` or any other string instead of the only correct expected answer `"F"`, and run the test script again. You should now see an error message similar to following:

```
109 Python Problems tester, June 10, 2020, Ilkka Kokkarinen.
ryerson_letter_grade: DISCREPANCY AT TEST CASE #0:
TEST CASE: 0
EXPECTED: F
RETURNED: Fail
0 out of 1 functions (of 109 possible) work.
```

This error means that your function returned the wrong answer for at least one of the test cases that the automated tester gives to this function. Restore the `ryerson_letter_grade` function to its original form, and watch it cleanly pass the test again.

Since the individual lab problems are relatively small and there are so many of them, the grading for this course grants no partial marks for effort for functions that do not pass the tester script cleanly and promptly. For each function to be tested, the tester script generates a large number of **pseudorandom** test cases to your function, and computes a **checksum**, sort of a **digital signature**, of the answers returned by your function. (The term "pseudorandom" means that these test cases are effectively as good as genuinely random test cases, and yet will always be the same for everyone everywhere every time this test battery is executed.)

If this checksum is different from the checksum originally generated from the answers returned by the instructor's model solution, these two functions must have returned a different answer to at least one test case. However, **the checksum mismatch reveals only the existence of at least one such difference**, but cannot even in principle pinpoint which test case caused it. A checksum mismatch also does not reveal whether these functions disagreed on the answer for just one **edge case**, or for a thousand ordinary test cases.

To direct the debugging effort after such all-or-nothing checksum mismatch, the `record` file included in the repository contains the expected correct answers for the first 300 test cases for each function, as recorded from the results of the instructor's private model solution for those same pseudorandom test cases. The test script will compare the results produced by the student function to these recorded results, and stop at first discrepancy to show the test case, the expected answer, and the student function answer. Even if some functions will be tested with millions of test cases, the vast majority of bugs that exist in student functions will almost certainly be revealed by at least one test case among the first 300 test cases. The test case generators inside the `tester109.py` script have been generally designed to yield their test cases

in increasing order of length and complexity, so the first reported test case mismatch tends to be relatively short and simple among all such failing test cases.

The specification of each function in the problem specification document comes with a handful of representative test cases to immediately try out the function interactively. Once the function passes this small sample of representative test cases, you know that at least you have solved the correct problem. Sometimes, the test case listed in the last line of that table is meant to show how large arguments your function is expected to gracefully handle; some functions must not gulp even when given behemoth arguments of something like one googol, sometimes even far above that!

However, **your function returning the expected result for the test cases given in the specification does not entail that your function is equally correct for the infinite number of other possible test cases**. The automated tester throws a randomly chosen subset of these infinite possibilities as a "shotgun blast" towards your function, hoping for at least one of the individual test cases ("pellets") to hit something revealing.

Lab problems

Having gained the understanding of how the automated tester works and what its various messages mean for the functions being tested, students can start working with the graded labs. During the lab session, the instructor will answer the questions from students regarding the labs that they are stuck with, and also demonstrate the general process of solving these labs with some particular problems chosen for this purpose. Solutions to the problems covered in these weekly lab sessions can be included in the submission of labs at the end of the course.

The first three lab problems from the collection "109 Python Problems for CCPS 109" chosen for us to sharpen our teeth with this week are:

1. Multidimensional knight moves (`knight_jump`)
2. Tukey's ninther (`tukeys_ninthers`)
3. Interesting, intersecting (`square_intersect`)

Module 6: Some educational example scripts

Introduction

The material of this week's module does not introduce any new major language features, since the existing ones are already ample enough for all the problems we wish to solve. Instead, we go through a number of example programs that showcase the language structures used so far, and most importantly, how to put these structures together in different ways to create different

behaviour and outcomes. Python already comes with a module for importing data from JSON files and turning their contents into Python objects so that we can perform some poor man's data science on them.

With no new language to learn this week, the reading material instead contains three sections on software testing and how to come up with good test cases for the functions that you write both during this course and the rest of your career afterwards. All interesting functions contain bugs while they are being developed. The faster we can pinpoint the existence of these bugs so that the logic of the function can be reorganized to work correctly, the better.

Topics

1. Reading in data encoded in the JSON standard format for representing structured data, and processing it with Python.
2. Writing functions to solve problems in a problem domain of playing cards and some familiar games that are played with them.
3. The `global` keyword that allows your function to talk about data in the same module.
4. Having a while-loop run around a large problem space looking for a goal that satisfies our expectations for the given problem.
5. Using custom sorting criteria with the optional `key` function given to `sort` function.

Learning objectives

This module reviews and reinforces the core control structures that Python functions are made of, and how these structures can be put together in various combinations to solve interesting computational problems. This knowledge is necessary for advancing from the knowledge level of the Bloom taxonomy to the level of applying this knowledge to solve problems. Students know how to import data from JSON files to give our programs something interesting from the real world to deal with instead of just always doing everything with made-up numbers. They know the mechanism to make their functions remember things by naming these things outside the function in the same module so that they continue to exist even between calls. They can modify the standard `sort` function to use a custom sorting criteria by giving it the optional `key` function, and create such `key` functions as one-liners with `lambda` expressions.

Readings

1. "Python Lecture Notes", sections "6.1. Different levels of programming errors", "6.2. Software testing", "6.3. Designing good test cases", "6.4. JSON".
2. GitHub repository `ikokkari/PythonExamples`: `stringproblems.py` `countries.py` `mountains.py` `specialnumbers.py`
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "6-1: Some hazards we know", "6-2: Loop and a half men", "6-3: Powerful words", "6-4: On the primal side".

Lab problems

This week, another three graded labs were chosen from the collection "109 Python Problems for CCPS 109" to function as our training ground and a confidence course:

1. Three summers ago (`three_summers`)
2. Nearest smaller element (`nearest_smaller`)
3. What do you hear, what do you say? (`count_and_say`)

Module 7: Lazy sequences

Introduction

So far, our functions have been dealing with random-access sequences so that even though we usually process such sequences linearly from beginning to end, our functions could jump to any position in the sequence to access the data there in the same constant time. However, such eager sequences are fundamentally restricted by the amount of memory available to Python. Especially many important combinatorial sequences such as "all possible ways to choose a bridge hand of thirteen cards from the deck of 52 cards" could not possibly fit in the memory of any actual computer.

Lazy sequences are better for situations of such exponentially explosive nature. As far as the rest of the Python language knows, lazy sequences are sequences that can be processed linearly from beginning to end using the same old for-loop that we previously used to process the elements of eager strings and lists. However, instead of all elements of such sequence existing simultaneously in computer memory, lazy sequences generate their elements one at the time only when requested, and use computational means to do this based on some smaller amount of state data that is enough to compute the next element. Such sequences therefore have no inherent limit for their length, and could even be infinitely long. Decorators that transform lazy sequences into different lazy sequences can then extract from this infinity the finite part that they need.

Topics

1. The fundamental difference between lazy and eager sequences, and the advantages and downsides of each representation.
2. Defining lazy sequences easily with generator functions that `yield` their results one at the time.
3. Generator decorators that can be applied to transform an existing sequence into some different lazy sequence.

4. The highly useful generator decorators of the Python `itertools` module.

Learning objectives

After this module, students can explain the difference between eager and lazy sequences, and what type of situations each kind of sequence is appropriate. They know how to turn their ordinary functions to print out values into generators that `yield` these same values one at the time for further processing. Since these generators are functions that take arbitrary arguments, even other generators, students then automatically know how to write generators that decorate an existing sequence given to it as an argument. They are aware of the Python `itertools` module and can explain its general purpose, and can consult the documentation to use the decorators `islice` and `takewhile` to turn an infinite lazy sequence into a finite one. They know how to consult this documentation to use the combinatorial generators in the `itertools` module to loop through all possible pairs, triples, subsets, permutations and other basic combinatorial structures based on some existing sequence, such as a deck of playing cards.

Readings

1. "Python Lecture Notes", section "7.1. Generators", "7.2. Functional programming tools", "7.3. Dynamic evaluation of text as code".
2. GitHub repository `ikokkari/PythonExamples`: `generators.py` `reservoir.py` `hamming.py`
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "7-1: To get all I deserve and to give all I can", "7-2: Let George do it", "7-3: Zero knowledge"

Lab problems

The three lab problems chosen for this week from the collection "109 Python Problems for CCPS 109" continue the theme of dealing with sequences with more complicated operations that combine ideas from previous lectures.

1. Revorse the vewels (`reverse_vowels`)
2. Boustrophedon (`create_zigzag`)
3. Bulgarian solitaire (`bulgarian_solitaire`)

Module 8: Recursion

Introduction

Recursion, the act of some function calling itself for smaller argument values as part of its own execution, has the reputation of being the most mysterious and esoteric technique learned in a

first programming course. It also has a reputation of being useless, but this follows from the fact that most introductory programming courses only ever teach simple linear recursions with the cliches factorial example, leaving the students with an impression that recursion is just a needlessly convoluted and mathematical way to do the job of a for-loop.

The power and usefulness of recursion lie in its ability to branch into several directions and combine the results of the recursive calls into the answer returned from the current level. After the factorial example used to teach the concept of self-similarity of a problem that makes it suitable for recursion, the examples of this module involve problems that greatly benefit from recursive branching in solving them in a manner clearly superior to the traditional imperative approach.

Topics

1. Self-similarity of computational problems as the key to unlock the recursive implementation to solve that problem.
2. Downsides of recursion compared to solving that same problem with loops.
3. Recursions that branch into two or more directions to solve the original problem.
4. Branching recursion to explore an exponential space of solutions.
5. Decorating functions with functions in Python.
6. Recursions that take exponential time due to needlessly solving the same subproblems, and how such wild recursions are tamed with memoization using `@lru_cache`.

Learning objectives

After this module, students recognize and can explain what makes a particular function recursive, and why such recursive functions are the natural choice for solving computational problems that exhibit self-similarity. They know why deep linear recursions are no good in practice, and know the rule that every such linear recursion should have been written as iteration to begin with. They can recognize a branching recursion that suffers under the exponential burden of needlessly repeated subproblems, and can decorate their recursion with `lru_cache` to make the function remember the subproblems that it has solved so far, so that the next time around their results can be simply looked up instead of fully recomputed all the way down from scratch. They have also seen enough examples of branching recursions for interesting computational problems to appreciate the qualitative difference between recursion and iteration in practical coding.

Readings

1. "Python Lecture Notes", sections "8.1. Recursive functions for self-similar problems", "8.2. Practical application of recursion", "8.3. Downsides of recursion", "8.4. Applying functions to other functions", and "8.5. Defining small functions as lambdas".
2. GitHub repository `ikokkari/PythonExamples`: `recursion.py` `functional.py`

3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "Python 8-1: A self of selves", "Python 8-2: Remember to remember", "Python 8-3: If you can't beat them, join them", "Python 8-4: Until time itself runs out", "Python 8-5: Twisty little passages", "Python 8-6: Paint every corner"

Lab problems

Recursion is not a separate language feature on its own in Python or any other programming language, but merely another way of using and putting together the existing features of that programming language. It can therefore be freely combined with all other language features to do the work that it does best.

Any lab problem from "109 Python Problems for CCPS 109" and elsewhere that you have solved and will be solving with loops could have also been solved with recursion instead, at least in theory if not always in practice. However, unless genuine branching into multiple directions is involved, there is no point in having recursion do the work of a simple loop and that way make your function crash with a stack overflow error once the recursion gets deep enough. Linear recursions should be used only for practice to get students accustomed to the idea of recursion and its general use.

Some problems in the 109 lab problems collection that benefit from the use of recursion are the following. There are also several more, although all of them have been all placed in the second half of the collection.

1. Lattice paths (`lattice_paths`)
2. Split within perimeter limit (`perimeter_limit_split`)
3. Sum of distinct cubes (`sum_of_distinct_cubes`)

You should definitely use the `lru_cache` decorator in the first two problems. However, it will not help you in the third one, but you need to think of other ways to speed up the computation by not trying out more combinations than you need.

Module 9: String processing

Introduction

Even though everything inside a computer is made of mere small numbers under various disguises, programs that deal with the real world and especially its human aspects have to deal with textual data. In this course, the wordlist `words_sorted.txt` serves tons of interesting textual data for our Python functions to derive interesting results from. When combined with

recursion and lazy sequences from the previous two modules, computation of complex questions about recreational linguistics becomes a breeze.

Topics

1. Applying the ability to slice and dice string objects and put these pieces together into words from a very large list of words, to find all the words that have some curious property from recreational linguistics.
2. Using binary search to speed up the search for words that start or end with the given prefix.
3. Using simple regular expressions to look for patterns inside words.
4. Using recursion to generate all possible strings that match the given rule.
5. Combining recursion and lazy sequences to systematically visit all possible solutions to the given recursive problem, instead of returning merely the first or the best such solution.

Learning objectives

After all the multitude of examples from this module, students are confident in their abilities to operate on character sequences. They can explain why binary search is a good algorithm to use when they know the data to be sorted. They know that regular expressions exist as a universal text pattern matching mechanism and which Python module contains functions to harness this power. They can use the branching possibilities of recursion to extend the existing string prefix into all possible continuations within the given constraints.

Readings

1. There are no lecture notes for this week. Instead, this week we go straight to the source, so to speak, and learn to study the Python standard library documentation. After that, our example classes showcase the power of Python string operations, especially when combined with recursion and lazy sequences.
2. GitHub repository `ikokkari/PythonExamples`: `wordproblems.py` `morse.py` `wordfill.py`
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "Python 9-1: Trie as you may", "Python 9-2: A multitude of words", "Python 9-3: Steamy computations", "Python 9-4: It means what I choose it to mean".

Activities

Read through the documentation page "Text Sequence Type - str" to find out what kinds of handy things the ordinary Python strings can do for you. There is never any point memorizing any of the standard library documentation, but you should get a general overview of the things that are available there so that you don't end up needlessly reinventing these wheels all the time.

After all, for some function to gain the exclusive membership in the standard library, it must have a general and useful nature in doing something that is not specific to any particular problem domain. Functions become part of the standard library to scratch very real itches that every programmer has constantly. Chances are that you have already had those same itches several times, as can often be seen by googling an answer from Stack Overflow and watching how quickly Google auto-completes your search, but perhaps having been unaware of the existence of the scratchers, some of which have quietly existed for decades now.

Lab problems

In the spirit of the topic of this module, the lab problems from "109 Python Problems for CCPS 109" for this week deal with text strings and collections of strings.

1. Uambcsrlne the wrod (`unscramble_word`)
2. Expand positive integer intervals (`expand_positive_intervals`)
3. Possible words in Hangman (`possible_words`)

Module 10: Numerical computation with numpy

Introduction

The numpy extension, already included in both the Anaconda package manager and the repl.it cloud-based Python programming environment, can be used to perform big and heavy numerical calculations that would require excessive time or space done with bare Python. The flexibility of data in Python requires its internal data representation to need more space than the storage of that information would technically require. However, for many data sets coming from the real world, the numerical nature of this data is known and certain. This knowledge may allow more efficient and compact internal representations as numpy arrays that pack these homogeneous data items together like sardines, mercilessly cutting off all the parts that don't fit inside the uniform Procrustean bed that is assigned for each individual data element.

These low-level operations performed close to hardware are still fully integrated as native citizens of Python language. The numpy array data type can represent not such vectors but matrices and tensors of arbitrary number of dimensions, and mathematical functions for numerical operations do the right thing even when applied to operands of incompatible dimensions. Even better, the scipy library for numerical analysis and all the domain-specific libraries such as scikits-learn built on top of it allow amateurs and scientists alike to trust the correctness of their numerical analysis.

Topics

1. Using the `sys` module for introspection to find out things about the current state of computation.
2. The idea of flexible representations of data whose exact nature we cannot predict, versus inflexible but more compact representations for data of known fixed nature.
3. Creation and manipulation of low-level numpy arrays of arbitrary dimensions of uniform elements of data.
4. Mathematical operations on numpy arrays.
5. Image processing with numpy, with the acceptance of such images as actually being matrices of small numbers under all that colourful presentation.

Learning objectives

This one module by itself does not turn anyone into a data scientist, but the educational value is understanding the qualitative difference in the contrasting approaches of core Python and numpy to the issue of what data even is and how it is represented inside programs. However, after this module the students can read through simple numpy examples without any fancy operations involved in them and explain, often after quickly consulting the numpy documentation that they now know to keep open in another browser tab, what each statement does and why it is there. Aided with the documentation, they can create numpy arrays and perform basic mathematical calculations on them at whichever level they are in their mathematical knowledge outside this course. They can appreciate the power of numpy in its ability to perform image processing, and that a pixel image is just a matrix of numbers that encode the colours.

Readings

1. "Python Lecture Notes", sections "10.1. The numpy array data type", "10.2. Operations over numpy arrays", "10.3. Processing pixel images as numpy arrays".
2. GitHub repository `ikokkari/PythonExamples`: `sysandos.py` `numpydemo.py` `imagedemo.py` (students who catch the numpy fever from this module can check out more advanced examples `fractals.py` `matplotdemo.py` `scipywords.py` that showcase the capabilities on numpy, scipy and matplotlib)
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "Python 10-1: The cost of everything", "Python 10-2: Bytes laid bare", "Python 10-3: Pie pants, one size fits all".

Activities

Of all the functions that you have written so far or seen as examples in the course material that receive a list of integers as their argument, rewrite some of these functions of your choice to

receive and handle numpy arrays as parameters instead. As you perform this conversion for various functions, especially those that create and return another list built with a repeated application of `append`, pay attention to how drastically the internal logic and structure of that function can change when it is forced to deal with fixed-size arrays for data representation, the way that all programming was done (and still is) in lower-level programming languages such as C++ and Java.

Lab problems

Since the collection "109 Python Problems for CCPS 109" was designed so that each problem could be solved with only the core Python language and recursion for those problems that gain from it, it contains no problems that would make the function deal with numpy arrays and their operations. Therefore, the following three problems have been chosen for their instructive and educational value, but have at least a little bit of numerical flair so that they could be used as examples in the activity of converting a Python function to numpy.

1. Bulls and cows (`bulls_and_cows`)
2. Fibonacci sum (`fibonacci_sum`)
3. Aliquot sequence (`aliquot_sequence`)

Module 11: Classes and objects in Python

Introduction

When our problems are simple enough, low-level representations of data laid bare are sufficient to deal with them. However, for a data type intended to represent some important problem-domain concept such as an ordinary playing card would be useful to exist as a code as an abstraction on its own, rather than our code having to deal with the low-level representation as "a tuple of a suit and a rank that you must handle with string operations". Once the concept of a playing card exists as a data type on its own, the user code can perform its operations on that concept at higher level, and does not have to hardcode and implement the internal representations of playing cards as a tuple of strings.

Even though this fact lies hidden below the surface of the deceptively simple syntax, Python is very much of an object-oriented programming language, more so than most programmers with experience from other classic mainstream languages such as Java or C++ would assume. This last module shows how to define your own high-level data types as classes, of which any number of individual instances can be created. The methods defined inside the class allow the implementations of the operations on that data type to be hidden inside the type, instead of the functions operating on the data type on the outside aware of its internal representation details. All operators in the Python language can be made to work with these classes merely by defining

certain standard "magic methods" inside the class. The Python compiler will then translate those operators into these corresponding magic method calls.

Topics

1. Thinking in higher-level abstractions about concepts, instead of their low-level representations inside our language.
2. Defining classes in Python. Methods and data inside classes and objects.
3. Constructors and other magic methods that turn the class into a truly naturalized citizen of the language.
4. Dynamically adding and modifying properties in classes and objects with the "monkey patching" mechanism.
5. Extending existing higher-level types into more specialized subtypes.

Learning objectives

After this module, students can be given a simple toy model concept in the style of "bank account" or "animal", and they could list some methods that would be verbs associated with the concept in the problem domain. They are able to write that concept as a Python class with the appropriate methods `__init__` and `__str__`, along with the methods that correspond to those chosen verbs. They would know how to look up which magic method corresponds to the language operator that they would like to work with that data type, and implement that magic method in that class. They can explain the dynamic nature of Python as names existing inside the objects, and how this reality naturally allows some methods to be "monkey patched" to behave differently just for one object as they do for other instances of that class.

Readings

1. "Python Lecture Notes", sections "11.1: Thinking in abstractions instead of implementations", "11.2. Defining your own classes", "11.3. Naturalized citizens of Pythonia", "11.4. Patching in new properties", "11.5. Subtyping existing concepts", "11.6. Multiple inheritance"
2. GitHub repository `ikokkari/PythonExamples`: `cards.py` `temperature.py` `shape.py`
3. YouTube playlist "CCPS 109 Computer Science I, Python" videos: "Python 11-1: Brothers from the same mold"

Lab problems

Again, since the problem collection "109 Python Problems for CCPS 109" was designed so that each problem can be solved using only the core Python language with an occasional sprinkle of recursion (but only for those problems that actually gain something from that), it contains no problems where it would be necessary, or even remotely sensible, to define your own data types for complex concepts of that problem domain. Instead, these last three problems were chosen

for this module for their educational value. Each problem also illustrates some techniques of algorithmic spirit that might come handy in future problems.

1. Bridge hand shorthand for (`bridge_hand_shorthand`)
2. Manhattan skyline (`brangelina`)
3. Autocorrect for stubby fingers (`autocorrect_word`)