Python Lecture Notes for CCPS 109

These lecture notes go through the important things about the core Python programming language and using that language to solve problems, as taught by Ilkka Kokkarinen for the course **CCPS 109 Computer Science I** for the Chang School of Continuing Education, Ryerson University, Toronto.

This version of the document was released **July 29, 2021**, eliminating most of the verbosity of the previous versions. These bullet points should cover all of the important ideas and all of the minor issues that we encounter during the course. However, the example programs in the GitHub repository `ikokkari/PythonExamples` are at least of equal importance to these notes, and absolutely necessary to see how all these theoretical concepts work out in practice.

This document and all our example programs use Python 3 throughout, and are not compatible with the older Python 2. All this teaching material, including both these notes and the associated example programs, is released under GNU Public Licence v3.

Module 1: Python as scriptable calculator

1.1. Arithmetic expressions

- In the **interactive mode**, the Python **REPL** reads in **expressions** one at the time, and immediately **echoes** the results each expression.
- More complex expressions can be built up from **constant literals**, **arithmetic operators** and **function calls**. Same as in other major programming languages, the language of **integer arithmetic** is embedded inside the Python language.
- Python expressions are always evaluated **inside out**.
- The **precedence** and **associativity** of mathematical operators behave as you learned in basic mathematics, and can be bypassed with **parentheses**.
- **Exponentiation** is denoted by the `**` operator.
- For no rhyme or reason, the character `%` has nothing to do with percentages, but denotes the **remainder** operator of integer division. For example, `10 % 3` equals `1`.
- Every expression in Python evaluates to some **object**, although this result can also be the special placeholder value `None` denoting the absence of an object.
- Expressions can be **nested** to arbitrary depths inside each other. It is often better to break it down into a series of simpler expressions that are executed in **sequence**.
- Python stores integers using a flexible encoding that makes even behemoths such as `1234**5678` a breeze to evaluate.

1.2. Scripts

- When given an entire **script** to execute as a **program**, Python does not automatically print out the results of the individual expressions. You certainly don't want your screen flooded with the results of possibly billions of evaluated expressions!
- The intermediate results are displayed for the user only when the expression calls the `print` function to output them.
- Modern IDE's such as **Spyder** or **PyCharm** use **syntax highlighting** with colour and font effects to make the structure of the code easier to discern for the human eye. These effects are dynamically generated by the editor for its display, and are not encoded inside the actual raw text script file unlike they would be in a Word document.
- Python functions know to do the right thing when applied to different types of arguments.
- For example, `2+2` gives `4`, and `'hello'+'world'` gives `'helloworld'`.

- Python is **case sensitive**, so that `print` and `Print` are different names.
- Python actively uses **whitespace** to denote the **nesting structure** of the code.
- Individual statements are separated by line breaks. However, if the statement is obviously **syntactically incomplete**, such as not yet having closed an earlier open parenthesis, the line break is treated as ordinary whitespace.
- So that other people (including all future versions of yourself) can understand what your code does, write **comments** to explain the purpose of the code.
- PEP 8 is the **official style guide** of Python. PyCharm and Spyder allow code inspections to ensure that the code conforms to this style.

1.3. Naming things

- **Names** (also called **variables**) are given to data objects to talk about them.
- Python doesn't care what these given names "mean" in our language. You should still aim to use names that are meaningful for human readers.
- In the interactive REPL, the special name `_` refers to **most recent result**.
- Using a name not yet defined crashes the script with `NameError`
- Each name comes to existence into your current **namespace** at the first **assignment** to it.
- A **namespace** is a structure that keeps track of names the objects they currently refer to.
- Any name can be **reassigned** to point to some other object.

1.4. Assignment operator

- The **assignment** operator `=` first evaluates its **right hand side,** and makes the **name** on the **left hand side** refer to the result.
- It is different to say `a=b` than `b=a`. Despite the `=` character denoting assignment, **assignment does not behave like equality in mathematics**.
- **Python is not a spreadsheet.** At all times, **every name only points to its associated object**, but remembers no history about where that value came from.
- The operator `is` checks if two names refer to the exact same object, as in the canonical example expression `clark_kent is superman` to illustrate a situation where the same object is referred to by two separate names.
- The built-in function `input` reads in a line of text input from the user. Since your program has zero control over the user, they can enter anything they want.

1.5. Types

- Types never show up explicitly in the source code. Python is still a **strongly typed** language, so that every object has a **type** that determines what operations are possible.
- Python typing is **implicit** in that the language itself does not talk about the types of objects. The type of an object is determined by the expression that creates that object.
- For example, the expression `2+2` gives an integer, whereas `2<3` gives a **truth value**.
- **An object, once created, cannot change its type or memory address as long as that object exists in the memory.** The contents of **mutable** objects can change without changing the identity of that object, though.
- To find out the type of some object, use the built-in function `type`.
- If `num` is an integer expressed as a string of digits, such as `"42"`, `int(num)` extracts this number. The function `str` turns it back into a string.
- What about `int("Hello")` ? When an expression cannot have a meaningful result, Python **raises an error** instead.

1.6. Importing functionality from modules

- You don't want to be constantly reinventing the endless wheels, but **import** as much existing code as possible.
- The `import` statement executes a module. Its names are stored in a new namespace that becomes a **module object** in your current namespace.
- To access a `name` inside an `object`, use `object.name`.
- If you need only one or two functions, use `from Foo import bar`. The script is still executed in a separate namespace, but the name `bar` is lifted to your current namespace.
- The **wildcard** version `from Foo import *` is only ever used in the REPL, never in scripts, and modern IDE's will highlight it as a style violation.
- According to PEP 8, all `import` statements should be placed at the beginning.

1.7. Dealing with fractional numbers

- Python represents decimal numbers using **floating point** with 64 bits per number. The floating point is an excellent way to **approximate** a wide range of decimal values with dynamic precision that gets less granular the further away you get from zero.
- The expression `0.1 + 0.2` does not equal `0.3`, but `0.30000000000000004`.
- Even worse, floating point operations are not **associative**. For example, the expression `(0.1+0.2)+0.3` gives a different result than `0.1+(0.2+0.3)`.

- This is not Python's fault, but inherent to the floating point encoding itself. Floating point arithmetic is done on the processor hardware, Python just reporting the outcomes.
- Unlike integers, floating point has special values for **positive and negative infinity**, and the special value `NaN` ("not a number") for an operation with no meaningful result.
- The module `fractions` defines the datatype `Fraction` for exact integer fractions. For example, `3*Fraction(1,3)` equals exactly `1`, not one iota more or less.
- When creating `Fraction` objects, remember to separate the numerator and the denominator with a comma, instead of the floating point division slash.

Module 2: Functions that make decisions

2.1. Defining your own functions

- A **function** is a **named block of statements** that performs some operation that we intend to perform more than once.
- The `def` statement associates a new name into the body of that **function object**.
- Whenever some function is **invoked** (**called**), the statements in its body are executed.
- A function can expect **parameters** from the caller, who must provide the actual values as meaningful **arguments** at time of the call.
- Even when the function takes no parameters, its call still requires the empty pair of parentheses. The function name is only a reference to the function object in the memory.
- Named parameters can be given **default values** for when when the caller does not provide those arguments.
- Every function will `return` a result at the end. If nothing explicitly returned with a `return` statement, the special value `None` gets silently returned.
- Names defined in the function body are created in a **local namespace** of the function that exists only during that call, and ceases to exist when the control returns to the caller.
- The keyword `global` causes a name to be accessed in the global namespace.
- If the first statement inside a function is a text string, it is treated as a **docstring** that explains what the function is supposed to do.

2.2. Simple two-way decisions

- **Two-way decisions** are the fundamental building blocks of all computational operations.
- Even adding two integers or comparing them for order internally breaks down into two-way decisions executed by the electronic logic gates of the physical computer.
- The `if-else` statement executes **precisely one of its two branches**, never both or neither, depending on whether its **condition** is **truthy** or **falsey** at the time.
- For integers, zero is falsey, and all other integers are truthy.
- More complex choices must be broken down into simpler two-way decisions. This could generally be done in multitude of equivalent ways.
- To embed a small decision inside a statement, use `expr1 if cond else expr2` that evaluates to either `expr1` or `expr2` depending on `cond`.
- The special value `None` is always considered to be falsey.
- Conditions can be built from **comparison** operators <, >, <=, >=, == and `!=` . Equality comparisons are denoted by == instead of =, since the latter already means assignment.

- Operator == always compares its operands **for their content, not their identity**.
- Operator `is` checks whether its operands are the **same object** in same memory address.

2.3. Complex conditions

- Python allows comparisons to be **chained**, in style of `a < b < c`. This is a special case of building complex conditions with the **logical** operators `and`, `or` and `not`.
- The expression `a and b` gives its first argument if it is falsey, and otherwise gives its second argument. The expression `a or b` is the opposite in that it gives its first argument if it is truthy, and otherwise gives its second argument.
- Usually `a` and `b` are proper truth values `True` or `False`, but they don't have to be.
- Just like multiplication has a higher **precedence** than addition, the operator `and` has a higher precedence than `or`.
- Unless your logic was redundant to begin with, **the two operators `and` and `or` are never interchangeable**. Replacing one operator with the other one will **always** change the behaviour of your function for some argument values.
- The operator `not` reverses the truth value of its operand. Make sure to use enough parentheses around this operand when it is a complex expression to avoid surprises.

2.4. If-else ladders

- A multiway decision from a fixed number of known possibilities is often best written as an **if-else ladder**, with the keyword `elif` denoting the steps in the middle.
- The first condition that is `True` decides which branch of this ladder gets executed.
- After executing the body associated to that condition, the execution skips all the remaining branches, even those whose conditions are also `True`.
- During the design of an if-else ladder, each step and its condition can be written while enjoying the airtight guarantee that all previous conditions have been `False`.

Module 3: Sequence types string, list and tuple

3.1. String literals from Unicode characters

- All computers merely store and move around small integers as **bytes** inside the computer memory. Higher data types must be internally expressed as **aggregates** of bytes.
- Python 3 represents characters and text strings internally in **Unicode** to guarantee portability of text processing between computer systems. The internal representation of these Unicode strings is out of sight, out of mind of Python programmers.
- A **text string literal** inside an expression is given between a pair of either **single** or **double quotes**, such as `'Hello there'` and `"I'm here!"`
- **Escape sequences** that start with the backslash character embed **special characters** inside text strings. The most common escape sequences are `\t` and `\n` to produce tab and newline characters, and `\"` and `\'` to embed a double or single quote character inside a string that was delimited in the source code with that quote character.
- To embed an arbitrary Unicode character, use the extended escape sequence `"\uXXXX"` where `XXXX` is the **Unicode code point** of that particular character expressed in **hexadecimal**, available on Unicode tables online.
- For emojis and other characters in the higher planes, use the form `"\UXXXXXXXX"`.
- String literals delimited with **triple quotes** can span multiple lines. Line break characters (they are characters just like any other Unicode character) are taken as part of that string literal that continues until the closing triple quote.

3.2. String literals from arbitrary pieces

- Modern **formatted strings** with the prefix `f` replaced **format placeholders** denoted inside the string with curly braces with the evaluated value of the expression inside the braces.
- **F-strings** are most commonly seen inside a `print` statement, but they can be used anywhere that you want to create a string literal from smaller components.
- Since text processing is so important in computing, Python strings have a ton of operations built in the language and as methods inside the string objects, as demonstrated in the example scripts.
- The library modules `string` and `re` (for **regular expressions**) provide even more powerful operations.
- Non-string objects an be automatically converted to strings for human consumption. For example, the integer object `42` becomes the two-character string `"42"` when `printed`.

- The integer object 42 in memory does not consist of any characters, but the bytes that encode an integer value consist of something entirely different!
- To explicitly convert any object to a string representation, use the built-in function `str`. Again note that this function does not modify the content or the type of the original object, but produces a whole new object as result of conversion.

3.3. Slicing

- Any desired part of the string or any other **sequence** data type in Python, can be extracted by **slicing** with the **square bracket** operator.
- Inside the sequence, the position offsets are numbered from zero. For example, the five-character string `"Hello"` has five possible offsets 0, 1, 2, 3, and 4.
- If the sequence is empty, it has no legal positions to access its elements.
- To extract a longer substring than just once character, give the **start** and **end** offsets separated by the **colon** character to define a **slice**.
- **The end offset of a slice is exclusive**. This convention simplifies the mental arithmetic on these slices, such as computing the length of the slice without an **off by one error**.
- Python uses negative offsets to count the **positions starting from the end**.
- For example, the expression `"Hello"[-4:-1]` gives`"ell"`.
- **Step size** or **stride** can be given as a third operand, a negative step size iterating the elements in reverse order. For example, `"Hello"[0::2]` evaluates to `"Hlo"`, and `"Hello"[::-1]` evaluates to `"olleH"`.
- `'Hello world'[3:]` evaluates to `'lo world'`.
- Slicing a substring **out of bounds** is not an error, as this operation simply treats the nonexistent part to be the empty string.

3.4. Lists of things

- Every programming language needs to have a mechanism to represent an arbitrary large number of objects under a single name. In the Python language, **lists** are special objects that contain an entire **sequence** of **objects** inside them.
- Python lists are **heterogeneous**; the same list can simultaneously contain objects of different types. These objects could even themselves be lists, resulting in a **nested list**.
- Unlike immutable strings, Python lists are **mutable** in both length and content.
- Python lists allow speedy `append` of new elements, and `pop` to remove them.
- Adding or removing an element inside a list can be slow, since all remaining elements have to take one step forward or backward to ensure contiguous representation.
- The built-in function `len` tells the length of a sequence.

- Python lists are also more flexible than **arrays** of other programming languages in that they can be **concatenated** and **extended**.
- A common programming idiom in writing a function that returns a list of answers is to initialize the local `result` list to be empty, and then `append` individual solution elements to the `result` list whenever a new one is found.
- If no solutions exist, the `result` list remains empty when returned.
- The canonical trick to create a separate but identical copy of the list `a` is to say `a[:]`.

3.5. List comprehensions

- A **list comprehension** takes an existing sequence to serve as a **source of elements** that will be **transformed** and **filtered** to create the result.
- The syntax of list comprehension consists of square brackets, followed by (1) the expression to transform each element, (2) the **for** to iterate through the elements of the existing sequence, and (3) the optional **if** to filter out unwanted elements.
- The lazy iterator `range` is often used as the source sequence of integers. For example, `[x*x*x for x in range(1, 11)]` gives the list of some cubes.
- A list comprehension can feature more than one for-block. Such expression will the iterate through all possible **combinations** of the values produced by these for-blocks.

Module 4: Sequence iteration

4.1. Sequential processing with for-loops

- Many computational problems performed on a sequence of data can be expressed as performing the exact same computation to every element, one element at the time.
- The **for-loop** is the powerful and flexible Python language feature to automatically execute the **nested body of statements** for every element of the given sequence.
- The for-loop is **polymorphic** so that the same statement can handle any sequence.
- The same nested body of statements will be executed once for every element of the sequence. Unlike in a list comprehension, this body can be arbitrarily complex.
- Sometimes processing either the first or the last element of the sequence has to be done in some special way. This situation is known as **a loop and a half**, since we often end up duplicating some of the code in the loop body either before or after the loop body.
- The more of your own code you end up duplicating, the more you ought to feel like being in a state of sin of not solving your computational problem the best way.
- **Sequence decorators** such as `enumerate`, `zip`, `reversed` and `sorted` make your code more concise and Pythonic.
- The `enumerate` decorator should be used whenever the processing of the current element depends on **both the element value and its position in the sequence**.

4.2. Iterator objects

- An **iterator** is a special object that has the ability to **produce a sequence of objects on command, one object at the time**. This sequence can be arbitrarily long, even **infinite**.
- Iterators can be created from existing sequences. Such iterators consult that sequence every time they need to produce a new object.
- An iterator can be explicitly requested to produce its next item of the sequence with the built-in function `next`. If no more values exist, a `StopIteration` error is raised.
- The sequence may even be empty. **Whenever there is nothing to do, the correct thing is to do nothing.**
- Iterators do not support **random access operations** such as **slicing** the elements of the sequence at arbitrary positions.
- To find out the element in some desired position, all preceding elements must first be produced one by one, with no “royal road” leading directly to the desired element.
- Always do your best to maintain the laziness of the argument sequences by using `for` and `enumerate`, but never `len` or slicing that cause the sequence to become explicit.

4.3. Integer sequences as range objects

- The `range` function creates a new object that lazily produces an **arithmetic progression** one number at the time. Compared to representing these integers explicitly in a list, the memory use of `range` is compact even for humongous sequences.
- The arguments for `range`, separated by commas, are the same as for the string slicing operator. Note again the exclusive end index that is often a silent pitfall.
- Given just one argument `n`, `range` produces the sequence `0, 1, ..., n - 1`.
- The idiom `for _ in range(n):` will execute its body exactly `n` times, without caring which round of iteration it is currently on
- These objects use arithmetic to determine that `-1 not in range(10**100)`, and that `len(range(10**10))==10**10`.
- The built-in function `list` builds the full list of the elements that its argument iterator produces so that all these elements exist in memory simultaneously. Unlike an iterator that steps through the values one at the time, this conversion can run out of memory in situations where the iterator produces a long series of objects, since all these objects will have to exist simultaneously in separate memory locations for the entire list to exist.

4.4. Files as sequences

- To open a file for reading or writing, use the built-in function `open` that returns a **file handle object** to represent the file inside the Python interpreter.
- The keyword argument `encoding` determines how the raw bytes of the file are interpreted as **Unicode** characters. In practice, the most common and space-efficient character encoding is **UTF-8**, the present *de facto* standard of Unicode encodings.
- This file object works as an iterator to the contents of that file. When iterating through a text file, the iterator gives the contents **one line at the time**.
- For more sophisticated processing of files, the file object has methods that can be used to move around the file, and read and write characters and bytes into the file.
- Python's compound statement `with` declares a **context manager** that guarantees that the file will be `closed` automatically after this statement no matter which way the execution leaves the body of the `with` statement.

4.5. Tuples as immutable sequences

- **Tuples** are heterogeneous immutable sequences of fixed size, usually two or three.

- In practice, tuples tend to be much shorter than strings, lists and other sequence types due to their typical use cases. For example, we would represent a playing card as a **pair** of `(suit, rank)`, and three-dimensional coordinates as a tuple `(x, y, z)`.
- Ordinary parentheses are placed around the tuple elements, but in many cases this is optional. Parentheses are necessary only in when creating a **singleton** tuple such as `(42,)`, and when a tuple is passed as an argument to a function.
- A handy quirk of the Python language is that multiple assignments can be performed with assignment into a tuple made up from the names to be assigned.
- For example, swapping the values that the two names `x` and `y` are currently associated with can be achieved in a single swoop with the assignment `x,y = y,x`.
- Tuples and other sequences are **order-compared lexicographically**. For example, `'aardvark' < 'zebra'`, and `(9, 1) > (1, 2, 3, 10**100)`.

4.6. Members only

- Instead of keeping all your data in one unsorted list, many programming tasks become easier when aided with good **data structures**.
- A **dynamic set** can be created either by calling `set()` for an initially empty set, or by writing a **set literal** by listing its **keys** inside **curly braces**.
- The operations `x in coll` and `x not in coll` to determine whether `x` is a member of that `set` are extremely fast even for millions of elements.
- The `set` type also offers important **methods** `add` and `remove` to mutate its contents.
- Unlike lists and strings, **sets are not sequences**, so they cannot be sliced or indexed. The question "What is the seventh element of this set?" is nonsensical from the get-go.
- `for x in coll` iterates through the keys in their **insertion order**.

4.7. From keys to values

- Instead of storing individual keys, a `dict` maps **keys** to **values**. For example, a phone book is a dictionary that maps names to phone numbers.
- A dictionary can hold at most one value for each key. Mapping a new value to the same key discards the previous mapping.
- A dictionary from keys to integers can be used as a **counter map** to keep track of how many something has been seen or done.
- Square brackets act as handy shorthand for accessing keys and values.
- Two special Python built-in functions `locals` and `globals` return dictionaries that contain the names in the current local and global namespaces, respectively.

Module 5: General iteration

5.1. Unlimited repetition

- Some functions need to do something **an unknown number of times that cannot possibly be known at the time the script is written**, so that this number of repetitions cannot be hardcoded into the function.
- The purpose of a while-loop is to reach some **goal** that we cannot directly just jump to, but have to approach with simple steps expressible as Python statements.
- The behaviour of a while-loop is best understood as "**As long as you have not reached your goal, take one step that will take you closer to that goal.**" Assuming that you can recognize the goal when you get there, this logic will reach the goal whenever the goal is reachable to begin with, and take exactly as many steps as needed, no more and no less.
- The while-loop will **correctly do nothing if you are already standing on the goal to begin with**, since its condition to keep going was falsey from the get-go.
- In this course, **infinite loops** that but run forever are a **logic error**.

5.2. Breaking out of loops prematurely

- Any loop in Python can be prematurely terminated with the `break` statement.
- The `continue` statement will go directly to the next round of iteration.
- Python syntax allows an `else` block immediately after a loop. It gets executed if the execution of the loop reaches its natural end without encountering a `break`.
- Such an `else` block can come in handy when the loop looks for a counterexample for something, terminating as soon as the first counterexample is found, since finding more counterexamples would not change anything.
- For more about how to write loops properly in Python, see the talk "Loop Like a Native" for the Pythonic constructs for various iterative situations.

5.3. Binary search

- To find an element from an unsorted list, there can be no essentially faster approach than **linear search** through that list one element at the time, comparing the current element to the element being searched until either you find it somewhere in the list, or you have looked at every element without success. This will be inefficient when the lists are large.
- If the elements are known to be in sorted order, **binary search** finds an element in a sequence must faster.

- In a sorted list, comparing any element to a value tells you something about all the elements that come before it, and about all the elements that follow after it.
- Binary search is a funny algorithm in the sense that everybody most certainly already knows it since they were little kids. They just don't know that they know it!
- Binary search repeatedly compares the **middle element** of the remaining subsequence to the value that is being searched for. This comparison eliminates either the left half or the right half from consideration.
- Python standard library module `bisect` contains battle-tested implementations of binary search to find the leftmost or rightmost occurrence of the particular element.
- In absence of the said element, these functions return the position where that element would have to be inserted to maintain the sorted order.

5.4. Repeated halving in action

- Binary search and other **repeated halving** algorithms are tremendously fast even for astronomically large lists. Each comparison cuts the problem in half by discarding half of the elements still in contention for becoming the final answer.
- For example, binary search in a list of one billion elements would require roughly thirty comparisons to pinpoint the desired position, since 10^9 is roughly equal to 2^{30}.
- If you were granted a "God's eye view" onto the entire universe, you would have to cut this view in half roughly only 250 times for a single elementary particle to remain.
- The powerful ideas of repeated halving and binary search can be applied more generally to other search problems that have a sorted range of possible answers for you to find the correct one. For example, if you are supposed to **guess the secret number** between `a` and `b`, make your first guess to be `(a+b)//2`, and continue accordingly depending on whether the secret number was smaller or greater than your guess.

5.5. Nested loops

- It is not only possible but perfectly legal and good technique to solve many problems by **nesting** an **inner loop** inside an **outer loop**.
- The language sets no limit to the depth of this nesting. Following the **zero-one-infinity principle**, the moment that two of something is allowed, any number should.
- To get an intuitive idea of how nested loops work, think of the behaviour of an ordinary clock that measures hours, minutes and seconds, implemented as three nested loops.
- The outermost loop counts hours from 0 to 23. For each such hour, the inner loop for minutes would go through its entire range from 0 to 59.
- For each such minute, the innermost loop would go through its entire range from 0 to 59.

- In some nested loops, the inner loop will run for a different number of rounds each time depending on the value of its outer loop counter.
- The operations `break` and `continue` always apply only to the innermost loop that they are in. (For clever ways to get around this restriction, see "Breaking out of two loops".)

Module 6: Some educational example scripts

6.1. Different levels of programming errors

- **Syntax error**: the program code does not conform to the syntax rules of the Python language. These are detected during the compilation and prevent the execution of the script, even the parts that precede the syntax error.
- **Runtime error**: the program crashes during execution because it tries to do something logically impossible, such as divide the string `"Hello"` by the integer `17`.
- Runtime errors can be handled dynamically with the `try-except` mechanism in some situations where it is still possible to recover from the error. Uncaught errors terminate the execution of the program.
- The `try-except` block can be followed by a `finally` block that is guaranteed to be executed no matter which way the execution tries to leave the `try-except` block (returning from function, raising an error, flowing through normally), provided that the Python interpreter does not itself crash or terminate.
- **Logic error**: the program is legal and produces some results without crashing, but at least for some possible inputs, the produced result is not what the programmer wanted it to be.
- Logic errors are by far the most difficult level of these programming errors, in fact the bane of our existence as programmers.
- Above the logic errors there exists even the higher level of **specification errors** where you have understood and defined the problem incorrectly, thus by definition making it impossible for you or anybody else to write the correct program to solve your problem.
- Our automated test suite for the graded labs tries out all functions with a large number of pseudo-randomly generated test cases that are always the same in all Python environments, regardless of the underlying hardware and operating system.
- A **cryptographic checksum** is computed from the results that your function returned to these test cases, and that checksum is compared to the checksum generated from the answers produced by the private model solution by the instructor.

6.2. JSON

- **JSON** (for JavaScript Object Notation, pronounced "Jason") is a widespread standard to **encode arbitrary structured data into linear Unicode text** in an unambiguous fashion that is portable between different types of computers and programming languages.
- JSON is designed to be **language-independent** so that the same JSON text files can be handled by programs written in different programming languages.

- Sharing this advantage with even simpler YAML, JSON is readable by humans, and can be further processed with any text editor and Unix command line scripts.
- Redundant whitespace can be eliminated from JSON files meant only for machines.
- The JSON format expresses structured data using JavaScript lists and dictionaries, whose syntax happily coincides with that of Python.
- The **atomic elements** of the encoded structure in a JSON file can only be **strings, truth values, integers and floating point numbers**, so that no executable code is allowed. This prevents various forms of **code injection** attacks.
- The atomic elements can be combined into lists and dictionaries. These lists and dictionaries can be nested arbitrarily deep, although usually just one or two levels.
- The function `load` in the `json` library module reads the given file and returns the object structure encoded in it. The **parser** inside the module treats the JSON data as pure text, and will never execute any part of it as code.
- Any data can still be **semantically misleading** in the sense that behaving as if that data were actually true will result in less than optimal outcomes.

Module 7: Lazy sequences

7.1. Generators

- A **generator** is a function that uses the keyword `yield` instead of `return` to return the result. When called as a function, it creates a new **iterator** object. The body of the function is not yet executed at this call.
- When this iterator needs to produce its next element, the function begins executing. When a `yield` statement is reached, the execution pauses and the yielded value is given out.
- The execution state of that generator instance remains in memory to produce the next element of the sequence.
- The execution resumes continues after the previous `yield` statement.
- The sequence ends when the generator reaches the end of its body. Some generators produce **infinite** sequences where that never happens.
- Multiple instances of the same generator can coexist, each with its own local namespace and execution state. They produce their values independently of each other, instead of being forced into lockstep.

7.2. The itertools standard library

- With the functions defined in the `itertools` standard library module, many functions can be expressed far more succinctly using functional programming applied to iterators.
- The function `islice` extracts a subsequence from a lazy sequence. Other functions combine and transform lazy sequences into new lazy sequences.
- The example implementations of `itertools` functions, followed by the recipes for more functions that did not make the cut of being important enough to be included into this module, make great studying of how to solve problems in the spirit of iterators.
- The module `more-itertools` offers even more such shorthands for concise expression of many common patterns of loops and iteration.
- If you choose to install additional packages from Python Package Index, make sure to use the `conda` installer instead of `pip` if you are working in the Anaconda environment.

7.3. Pseudorandom bits

- Since **deterministic** computers cannot produce genuine randomness anyway, **pseudorandom number generation** is done algorithmically.
- Once the **seed** value of the generator has been set, the random bits produced by such generator are as deterministic as a train running on its tracks.

- The internal operation of such a generator is highly **chaotic** so that its future sequence is highly sensitive to the details of its initial state.
- If no seed value is explicitly given, it is initialized from the system clock inside your computer. The millisecond precision guarantees that if the same program is run twice, the different starting times quickly makes their future random bits different.
- Python 3 is standardized to use the **Mersenne Twister** generator, independent of the underlying operating system and physical hardware. This makes random number sequences from the given seeds independent of the platform.
- The function `getrandbits` in the module `random` is used by the other functions in that module to generate the random bits they will then combine into values from other probability distributions.
- Some programs benefit from the existence of separate instances of the type `random.Random` that can advance independently of each other.

7.4. Random choices of integers

- Functions `randint` and `randrange` produce random integers from the given range, such as rolling a die with the possible results between one and six.
- These two functions internally get enough random bits from the `getrandbits` function to produce their result integer value fairly from the given range, even if the width of that entire range were somewhere in the googols.
- The common task of **random sampling** of elements from the given sequence are handled by the functions `choices` and `sample`, respectively **with and without replacement**.
- The task of **shuffling** the elements of the given list into a random permutation seems deceptively simple. However, the obvious algorithm of "repeatedly choose two elements from the list and swap them, repeat this for some energetic hand waving number of times") is not only inefficient but incorrect.
- The correct **Knuth shuffle** algorithm to produce each of the $n!$ possible permutations with the same probability of $1/n!$ is implemented as the function `shuffle`.

Module 8: Recursion

8.1. Recursive functions for self-similar problems

- A thing is **self-similar** if it contains a strictly smaller version of itself inside it.
- **Recursion**, a function calling itself for smaller parameter values, is often a natural way to solve self-similar problems.
- The exposed self-similarity allows **recursive definitions** for self-similar phenomena.
- For example, the non-recursive definition $n! \triangleq 1 * 2 * \ldots * (n - 1) * n$ turns into an equivalent recursive definition $n! \triangleq (n - 1)! * n$.
- To avoid **infinite regress** in principle and **stack overflow** in practice, every recursive method must have at least one **base case** where no further recursive calls are made.
- For the factorial function, the base case is $0! = 1$.
- **The secret of understanding recursion is that there is no secret**: nothing happens in a recursive function call that would not also happen in any other function call!
- Each recursive invocation of the same function has its own local namespace, so that the same names can exist independently at different levels of recursion.
- Any **linear recursion** where each function call produces at most one more function call should always be converted to straightforward loops.
- The true power of recursion is unleashed in its ability to **branch** to two or more different directions, to explore more than one possibility without committing to either one.

8.2. Downsides of recursion

- Recursion can be a powerful friend, but has with **three undeniable downsides**.
- First, a deep recursion doing the work of a loop will cause a **stack overflow**.
- Second, even without a stack overflow, the internal bookkeeping of the stack makes the recursive method a constant factor slower compared to the equivalent iterative solution.
- Third, a recursive method where each recursive call generates two or more smaller recursive calls can cause an **exponential chain reaction** of function calls.
- Easiest way to fix this is **memoization**: use an auxiliary data structure to remember what subproblems you have already solved, and when the recursion comes to those subproblems again, just look up the previously cached result and return that as the result.
- Memoization is similar to solving a problem with a **lookup table**, except that the lookup table is filled dynamically as encountered, not already hardcoded into the source code.

8.3. Applying functions to other functions

- Python functions are objects same as strings and integers, and can be passed as arguments to other functions and returned as results from them.
- A sort function tan be given a **key function** to determine how each element should be treated in order comparisons.
- The key function allows **sorting by multiple criteria** by having this key function return a tuple, since the Python tuple order comparison is lexicographic.
- A **function decorator** is analogous to sequence decorators in use and spirit.
- The function decorator `lru_cache` in the `functools` module can **memoize** any function to quickly look up the previously computed results.
- To ensure that the memoized results don't stick around to waste memory after the desired result has been achieved, a good technique is to define the memoized recursive function inside the actual function solving the problem. Once the local namespace ceases to exist at return, the memoization cache will be released from memory.

8.4. Small anonymous functions

- An **anonymous function** can be defined using the keyword `lambda` followed by the parameter names separated by commas, a colon and the expression body.
- For example, `lambda x,y: x*x+y*y` defines an anonymous function that takes two parameters and returns the sum of their squares.
- Python lambdas are restricted to only one-liners.
- **Operators** such as + or < are not objects that could be passed back and forth.
- To turn an operator into a proper function object, wrap it inside a lambda expression, such as `lambda x,y: x+y` for addition.
- The module `operator` defines named lambda expressions for all built-in operators.

Module 10: Efficient numerical computation

10.1. The numpy array data type

- The flexibility of data heterogeneous lists and dictionaries can be inefficient in both time and memory when dealing with large sets of data.
- The important Python extension **numpy** defines a powerful data structure `ndarray` (short for "n-dimensional array") to represent large 1-D **vectors**, 2-D **matrices** and even arbitrary higher-dimensional **tensors** of **homogeneous** elements.
- Since all elements inside the same numpy array are guaranteed to be of the same numerical type, they can be stored in memory compactly consecutively.
- Elements of `ndarray` have a maximum value that depends on the `dtype` of that array.
- If some computation produces a value larger than this space, the result **silently overflows** and is truncated in a Procrustean fashion.
- Type `intX` (**signed**) or `uintX` (**unsigned**) where `X` stands for the number of bits used, either 8, 16, 32, or 64. For example, the type `int32` is a **signed** four-byte integer.
- **Unsigned types** should be used to store known nonnegative values, since the highest bit is not needed to store the sign of some integers that are known to never be negative.
- The element types `int8` and `uint8` pack each element into one byte, thus offering the possible ranges of -128, ..., +127 and 0, ..., 255 respectively.

10.2. Operations over numpy arrays

- **Universal functions** operate **element-wise** on the entire `ndarray` at once.
- Whenever some numpy function is applied to numpy arrays where the other array has a lower dimensionality, that smaller array is **broadcast** into virtual multiple copies to make the dimensions of the two arrays match.
- Adding a two-dimensional matrix with shape `(10,20)` to a one-dimensional vector with shape `(20,)` broadcasts the second array into compatible `(10,20)` shape.
- Any **scalar** value can be thought of as a 1-element numpy vector for broadcasting.
- Matrix multiplication as the `dot` product of individual vectors is of broadcasting.
- From the outside, the `ndarray` data type looks and feels like a Python object.
- Slicing produces another **view** to the same underlying data, instead of copying the data to the new `ndarray`.
- Important extensions of numb are **scipy** for numerical analysis, **matplotlib** for rendering pretty graphs based on numeric data, and **pandas** for data analysis.

- **Scikits** are third party modules built on top of scipy, specific to some particular field of science. Extension scikit.learn offers a host of **machine learning algorithms**.

10.3. Processing pixel images as numpy arrays

- A **pixel raster image** is a three-dimensional **cuboid** of numbers. Two dimensions represent the pixel coordinates. Third dimension encodes the colour of that pixel in one or three **colour components**.
- **RGB** (red, green and blue) is used directly in the computers and their display hardware. **HSB** (hue, saturation and brightness) is more intuitive for humans to think about.
- In a **grayscale image**, the colour dimension needs only one brightness value, and the entire image is a two-dimensional `ndarray`.
- The colour components are either floating point numbers from 0 to 1, or **unsigned integers** from 0 to 255.
- In principle, any image operation could be written as a Python function that operates on these numbers, and this way theoretically build up all of Photoshop from scratch. A whole bunch of common image operations are hosted in `scipy.ndimage`.
- **Image convolution** with the given **kernel matrix** is a powerful operation to achieve a multitude of cool things. Many important image processing operations such as **edge detection** can be expressed as convolutions with a suitable kernel.

Module 11: Classes and objects

11.1. New types as classes

- Brand new data types and their operations can be defined as **classes**.
- The statements inside a `class` statement are executed normally, but the names they define are stored in a new namespace for the class object being created.
- Use the class name as a function to create a new **object** instance of that class.
- Functions inside the class are **methods** to be called for the instances of your class. Their first parameter `self` refers to the object for which the method is being called.
- A class can have **class attributes** to represent data that is the same for everyone, instead of being stored separately for each individual object.
- For example, inside a `BankAccount` class, `balance` would be an instance attribute so that every instance could have its own separate `balance`. The total `count` of how many bank account instances exist would be a class attribute.
- The convention of starting an attribute name with **a single or double underscore** means that the name is an implementation detail that may work today when accessed from the outside, but might change at any time in future versions of this class.

11.2. Naturalized citizens of Pythonia

- The arguments are passed on to the **constructor** method `__init__` that is automatically called for every a new object. This method typically creates the new object's **attributes**.
- Special **dunder methods** (complete list, also known as "**magic methods**") can be used to define the behaviour of Python's arithmetic operators, comparisons, indexing, ...
- For example, the Python function `str(x)` internally calls the method `x.__str__()` and then returns whatever that method returns.
- The built-in operators of the language, such as `a+b`, are silently turned into equivalent method calls, such as `a.__add__(b)`.
- Many operators impose an implied **contract** on the corresponding dunder methods to behave a certain way. For example, the order comparison `__lt__` should be **transitive** so that `a<c` whenever `a<b` and `b<c`.

11.3. Managing properties

- **Monkey patching** reassigns a name in one object intended to be special.

- Attributes can be turned into **managed properties** with `@property`. From the outside, managed properties look and feel like ordinary attributes, but reading and writing them causes the associated function to be silently executed.
- This makes the access to those properties more uniform. Inside the same `Temperature` object, both `K` and `C` are accessed as attributes, even though one of them is a method.
- The **property setter function** is typically used to enforce some constraints on the possible intended values of that property. For example, no `Person` should have a negative `height` or `weight`.
- The **property getter function** can make that property **virtual** so that its value is not actually stored anywhere inside the object, but is computed on the spot every time it is needed, based on the values of other attributes stored inside the object.

11.4. Subtyping with inheritance

- To make your class a **subclass** of an existing class, write the superclass name in parentheses after the name of the class, such as `class Car(Vehicle)`.
- For the class `B` to reasonably be a subclass of `A`, every instance of the problem domain concept *B* must also be an instance of the problem domain concept *A*.
- For example, every car is a vehicle, but a car is not an engine even though it has one.
- Whenever some name is accessed inside some object, Python climbs up the inheritance chain to find the first occurrence of that name. This way, each subclass needs to define only the new functionality of that subclass.
- You can **redefine** these method names in the subclass with a different implementation.
- **An abstract superclass** represents a problem domain concept that is too **abstract** to allow any concrete instances to exist, because some of its methods cannot be given a reasonable implementation at the level of this abstract concept.
- To make a Python class abstract, first have it subclass `ABC` from the module `abc`, and then decorate those methods that you intend to be abstract with `@abstractmethod`.
- Python syntax still requires these methods to have some kind of body, typically `pass`.

11.5. Multiple inheritance

- Python supports **multiple inheritance**, so that subclasses can have more than one **immediate superclass**.
- The **method resolution order (MRO)** algorithm in the Python virtual machine determines which method gets executed when the same method name is inherited from multiple superclasses. This usually does the right intended thing.

- Multiple inheritance is most commonly seen with **mixins**, small superclasses that define one particular method (or a handful of methods closely related to each other) and possibly some data to represent some concepts from the problem domain.
- Other classes, regardless of what they might otherwise themselves be about, can then simply extend these mixin classes to get that functionality for themselves for free.
- Functions defined inside a mixin need to be written only once inside the actual mixin, instead of having to duplicate their code inside all the classes that use those functions.
- The grand theme that permeates throughout *CCPS 209 Computer Science II* after this is the following: **DON'T REPEAT YOURSELF.** See you there.