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 28, 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

- When used in the **interactive mode**, the Python **REPL** interpreter reads in **expressions** entered one at the time, and immediately **echoes** the results of evaluating these expressions. This makes Python handy for small scale experimentation.
- 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.
- 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. In this special case, the interactive Python environment does not print out anything.
- Expressions can be arbitrarily long and complicated, since they can be **nested** to arbitrary depths inside each other. However, instead of trying to write a complicated computation as such a **one-liner**, it is often better to break it down into a series of simpler expressions that are executed in **sequence**.
- Python internally stores integers using a flexible encoding that makes life more convenient for programmers when the magnitude of integers is limited only by the memory available for the Python interpreter. Even behemoths such as `1234**5678` are a breeze to evaluate in a blink of an eye.
- (The fact that Python integers actually do work the way that we expect integers to mathematically work is never news to genuine beginners. It is news only to the lost souls from a more primitive era who were gaslit and browbeaten their entire lives to believe in the absurdity that in the hall of funhouse mirrors inside every computer, one billion is a large number, but three times that is something negative.)

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!
- Inside a script, the intermediate results are displayed for the user only when the expression calls the built-in `print` function that outputs its argument as characters.
- A Python script file must consist of **raw text**. Any type of **rich text document** (for example, Microsoft Word or Google Docs documents) that also contains formatting instructions such as font or paragraph styles embedded in the document itself cannot be executed as Python.
- As a rule, if opening the script file directly into a **raw text editor** such as Notepad or TextEdit shows any weird characters, your script file is not legal Python.
- When displaying a Python script, all 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 visual 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.
- Many Python functions such as `print` are **parameter polymorphic**; the same function can be applied to arguments of vastly different types, and the function still knows to do the right thing in each case.
- For example, `2+2` gives `4`, and `'hello'+'world'` gives `'helloworld'`.
- Python is **case sensitive** everywhere, so that `print` and `Print` are different names. However, 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. This allows long statements to be broken into multiple lines for readability.
- 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. Modern Python IDE's come with a built-in **linter**, a behind-the-scenes tool that looks for constructs that are legal in the language itself and can be executed as Python code, but go against the official uniform style and are occasionally symptoms of something being hinky in the underlying approach.

1.3. Naming things

- Every programming language needs the ability to create and give **names** (also called **variables**) to otherwise nameless data objects, so that expressions can unambiguously talk about those objects. This is achieved with an **assignment** such as `x = 42`.
- Names can consist of arbitrary **letters** (although in practice, only the twenty-six letters of English), digits and underscore characters.
- Python doesn't care what these given names "mean" in our language. Of course you should aim to use names that are meaningful and informative for human readers.
- In the Python interactive **REPL**, the special name `_` of a single underscore refers to the result of the **most recently evaluated expression**.
- Dan Bader's excellent Python tip page explains the full rules for underscores in Python.
- Each name automatically comes into existence into your current **namespace** at the first assignment to it, and refer to the object that was assigned to it.
- Trying to use a name not yet created crashes the script with `NameError`.
- A **namespace** is a data structure internal to the Python virtual machine that keeps track of what names exist, and which objects those names currently refer to.
- **A name is a separate and qualitatively different thing from the data object that it points to**. Any name can later be **reassigned** to point to some other object that does not need to be the same type as the original. The original data object still continues to exist in memory, and might be simultaneously referred to by other names (**aliasing**).
- To find out what names exist in the current namespace or inside some object, use the function `dir`. This function has nothing to do with your local **filesystem** or disk access, but lists the names that exist in the **global namespace** (if called without an argument), or some particular object given as argument.

1.4. Assignment operator

- Executing the **assignment** operator `=` evaluates its **right hand side** that can be an arbitrary expression, and then makes the **name** on the **left hand side** refer to the object given by the right hand side.
- Note how different it is to say `a=b` than `b=a`. Despite the unfortunate choice of the `=` character to denote assignment, **assignment does not behave like equality in mathematics**. This can be confusing to beginners.
- **Python is not a spreadsheet.** At all times, **every name only points to the associated object**, but remembers no history about where that value came from.
- Names allow us to talk about objects in memory, since these objects themselves don't have any inherent names. Some programs could create millions of objects during their

run. You certainly would not want to think up and painstakingly type in a new distinct name for each and every one of these objects in your source code!

- 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 and his sausage fingers, the characters that are entered can be anything, even if you explicitly ask the user to pretty please enter an integer. They might still type in `'Hello world'` due to an accident or malice, or just to find out if your program is prepared for that possibility.
- Tis author's heartfelt plea: **never use “input” as a verb**, especially in past tense where “inputted” sounds like something happening in miniature golf. And you **most certainly can't use the same verb "input" for both directions**, as in "The program inputs a number" and "The user inputs a number".
- The correct and unambiguous verbs for passing data back and forth are "**read**" and "**enter**", as in "The program reads the input that the user enters."

1.5. Types

- Despite the fact that these types don't show up explicitly in the source code, Python is a **strongly typed** language; every object in memory has a **type** that determines how that object behaves and what operations are possible to do with it.
- Python typing is **implicit** in that the Python language itself does not talk about the types of objects that it performs computations on. Python will know the type of the resulting object after evaluating the expression that creates that object.
- For example, the expression `2 + 2` would give an integer, whereas `2 < 3` would give 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`. The result of this function is an object (in Python 3, everything is an object)... but what is the type of this type object itself? Try it out!
- Some built-in functions to create a new object of that type. For example, if the name `num` contains some integer expressed as a string of digits, such as `"42"`, the expression `int(num)` extracts this number.

- What about expressions such as `int("Hello")` ? When an expression cannot give any meaningful possible result, Python **raises an error** instead.

1.6. Importing functionality from modules

- Python famously comes with “batteries included”. You don't want to be constantly reinventing the endless wheels spinning inside larger wheels, but rather **import** as much of existing top shelf stuff as possible.
- The `import` statement executes the given module. Any names created in the script are stored in a new namespace that, once executing the module is complete, becomes a **module object** in your current namespace that contains these names.
- To access a `name` inside some `module` object, use the form `module.name`.
- The same syntax can be used to access a name inside any object. Modules are also objects inside Python, the exact same way that integers, strings and files are objects.
- The naming convention of **starting a name with an underscore** informs the reader that the name is an **internal implementation detail** that should not be accessed from the outside, at least not without a good reason.
- If you need only one or two functions from a given script, you can use the alternative form `from Foo import bar`. The script is still executed first in a separate namespace, but afterwards, the name `bar` is brought up in your current namespace.
- The **wildcard** version `from Foo import *` is only ever used in the REPL, never in scripts, and modern IDE's will even mark it as a style violation.
- According to PEP 8, all `import` statements should be at the beginning of a script.

1.7. Dealing with fractional numbers

- Important library modules such as `math` provide functions needed in many programs.
- Python represents decimal numbers using **floating point** in 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 floating point encoding cannot represent exactly some numbers that we humans would consider "simple". For example, the expression `0.1 + 0.2` does not equal `0.3`, but `0.30000000000000004`.
- None of the three seemingly simple values 0.1, 0.2 and 0.3 has an exact representation in floating point using **base two**, even though they are trivially representable in **base ten**.
- 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 in the floating point encoding itself. All floating point arithmetic is done on the processor hardware anyway, with Python merely reporting the outcomes.
- The floating point encoding also has special values for **positive and negative infinity**, and the special value `NaN` ("not a number") for when some arithmetic operation cannot have any meaningful result.
- The module `decimal` defines the data type `Decimal` for arbitrary precision decimal numbers expressed in base 10. This `Decimal` data type is even smart enough to understand the concept of **significant digits** of a number, so that the nominally equal values 0.5 and 0.5000 represent the same quantity in different precision.
- Another highly useful 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, not the floating point division slash.
- To generate **random numbers** of various distributions, use the functions defined in the library module `random`.
- The modules `datetime` and `calendar` define data types and functions for calculations on dates and times, often needed in applications that deal with things in the real world.
- The modules `os` and `sys` define functions that allow your code to access the underlying computer and its file system.

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 function object acts as a **black box** that hides its internal implementation from the caller who cares only about the results produced by that function, but not the mechanism that produced those results.
- Function objects are defined using the special `def` statement that first creates the function name in the current namespace, and then immediately assigns this name to refer to the object that contains the function body **compiled** to a more efficient internal form.
- Functions are objects in Python, and can therefore be assigned to names the same way as strings or integers.
- Whenever some function is **invoked** (**called**), the statements in its body are executed in the order that they were written into the program source code..
- A function can expect **parameters** from the caller, who must provide the actual values as meaningful **arguments**.
- Even when the function takes no parameters, the syntax for its call still requires the empty pair of parentheses. The function name itself is just a reference to the function object in the memory.
- Named parameters can be given **default values** to be used when the caller does not provide a value as argument for them.
- In Python, every function will **return** some result at the end. If no result is explicitly returned with a `return` statement, the special value `None` is automatically returned.
- All names defined in the function body are created in a **local namespace** of the function that exists only during that function call. This namespace ceases to exist when the execution returns to the caller.
- The keyword `global` causes the following name to be created and 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

- Almost all functions need to make **decisions** to produce different results in different situations. **Two-way decisions** are the fundamental building blocks that all computational operations are made of.

- Within the hardware of the computer processor and memory, all operations such as adding two integers or comparing them for order internally break down into two-way decisions executed by the electronic logic gates as the laws of physics dictate.
- Python two-way decisions are created with the `if-else` statement. It executes **precisely one of its two branches**, never both or neither, depending on whether its **condition** is **truthy** or **falsey** the time the execution enters the statement.
- To choose from three or more possibilities, this choice must be broken down into simpler two-way decisions. Your programming and thinking styles ultimately determine which one of the equivalent ways to organize a **decision tree** you end up writing.
- To embed a small decision inside some larger statement, you can use `expr1 if cond else expr2` that evaluates to either `expr1` or `expr2` depending on `cond`.
- When `x` is already a proper truth value, beware the common pointless redundancy of writing the test in the form `x == True`.
- The simplest and clearest way to test whether `x == False` is to write `not x`.
- The special value `None` is always considered to be falsey. This allows the handy Python programming idiom of defining some function to take an optional argument whose default value is `None`, and then treating the argument as a condition inside the function to cause special behaviour if and only if the caller provided some value to that argument.

2.3. Equality and its outcomes

- Conditions can be built from **comparison** operators <, >, <=, >=, == and `!=` . Equality comparisons are denoted by == instead of =, since the latter already means assignment.
- The operator == compares its operands **for their content, not their identity**.
- The operator `is` checks whether its two operands are the very same object in the same memory address.
- Even if you consciously avoid using any floating point arithmetic, you might still accidentally end up creating floating point values by using the floating point division operator `/` to divide an integer by another integer, instead of the correct **integer division** operator `//`.
- Like all programming languages, the rules of Python also have occasionally counterintuitive consequences. Interested students can check out the collection "WTF Python" to amuse themselves, perhaps winning an occasional beer bet or similar from their friends with the most astonishing examples.

2.4. 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.

2.5. If-else ladders

- A multiway decision from a known fixed number of possibilities is often best written as an **if-else ladder**, with the keyword `elif` denoting the steps in the middle.
- When the execution of a script reaches the if-else ladder, its conditions are evaluated in the order in which they are listed in the source code. The first condition that is `True` decides which branch is executed.
- After executing the body of that condition, the execution skips all the remaining branches, even those whose conditions are also `True`.
- The order of the steps of the if-else ladder is immensely important. Rearranging them will usually change the behaviour of the ladder and the function that it computes.
- 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`.
- An if-else ladder whose last step is not an unconditional `else` is perfectly legal, but it still seems a bit hinky, and is often somehow wrong. (This is not an airtight rule either, though.) Such a ladder will then behave as if there really were an unconditional `else` branch at the end whose body simply does nothing at all.

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. String slicing

- In general, your program needs to define a separate name for each separate item of data that it deals with. In programs that need to deal with millions or even billions of items of data, we would rather not type these names into the program code.
- Individual elements of a `print` are accessed by their **position offsets** counting from the beginning, analogous to how apartment numbers are used to identify the individual households inside the condo tower whose street address all these apartments share.
- 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 positions offsets are numbered from zero, since the first character is located zero steps away from itself. For example, the five-character string `"Hello"` has five possible offsets 0, 1, 2, 3, and 4.
- The last legal position is always exactly one less than the length of the sequence. If the sequence is empty, it has no legal positions to access its elements.
- To extract a single character from a string, use square brackets with the desired position. To extract a longer substring than just once character, give the **start** and **end** offsets separated by the **colon** character to define a **slice**.
- Annoyingly, **the end offset of a slice is exclusive**, one step past the last character that you want to include in the slice. This convention simplifies the mental arithmetic on these slices, such as computing the length of the slice without any danger of **fencepost error** to create an **off by one error**.
- Negative offsets would be impossible by definition, akin to asking "What letter comes before the letter A in the English alphabet?" Python uses negative offsets to count the **positions starting from the end** instead of the beginning.
- For example, the expression `"Hello"[-4:-1]` gives`"ell"`.
- Even when using a negative offset, the end index of a slice is still exclusive.
- **Step size** or **stride** can be given as a third operand to slicing, with the negative step size defined to iterate the elements in reverse order. For example, `"Hello"[0::2]` evaluates to `"Hlo"`, and `"Hello"[::-1]` evaluates to `"olleH"`, this being the canonical way to reverse the given Python string.

- (To iterate through a reversed sequence without creating a separate copy of it, use the sequence decorator `reversed`.)
- When using the colon character, leaving out the start offset means to start from the beginning, and leaving out the end offset means to continue all the way to the end. For example, `'Hello world'[3:]` evaluates to `'lo world'`. Leaving out the start offset similarly means "all the way from beginning".

3.4. Random access of sequence elements

- For strings and lists, slicing operates on the principle of **random access**, meaning that it will need the same amount of time to access the first element as it needs to access the millionth element.
- Many **lazy sequences** do not allow random access to their contents, but only the operation of producing the immediate next element. There are no no boats, no trains, no planes that would take you directly to the millionth element, but the only way to get there is to walk one step at a time.
- Since Python string objects are **immutable** so that their content cannot change after object creation, **slicing will always create a separate string object to represent the result**. The original string object still continues to exist as unbroken original whole.
- **Immutability of data has several surprising benefits in programming.** For example, string objects created by slicing and dicing, theoretically even by concatenating, could share the underlying bytes that are set in stone, instead of redundantly duplicating these bytes for everybody. This makes many operations such as slice extraction more efficient in time and space.

3.5. Lists made 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 can even themselves be lists, resulting in a **nested list** or a **multidimensional list structure**, depending on how you wish to view it.
- Unlike the immutable strings, Python lists are **mutable** in both length and content.
- Python lists allow speedy `append` of new elements by maintaining an unused **slack space** as spare room. Once this slack space runs out, Python allocates more room in some other corner of the memory, and copies the data stored inside the list there.

- 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.
- Trying to slice out a substring **out of bounds** is not an error, as this operation simply treats the nonexistent part to be the empty string. For example, the expression `"Hello"[2:100]` gives the result `"llo"`.
- The built-in function `len` tells you the length of the given sequence.
- The canonical trick to create a separate but identical copy of the list `a` is to say `a[:]`.
- 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.

3.6. List comprehensions

- **List comprehensions** create lists in a concise way inspired by the mathematical notation used in set theory in math textbooks.
- A list comprehension needs some 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 three components of (1) the expression used to transform each element, (2) the **for-block** to iterate through the elements of the existing sequence, and (3) the **if-block** to filter out the elements that we do not wish to process but skip out entirely.
- The lazy iterator of integer objects `range` is often used as the source sequence. For example, the comprehension `[x*x*x for x in range(1, 11)]` would produce the list of cubes of the integers from 1 to 10.
- The if-part is optional when all the elements of the source sequence are to be unconditionally transformed.
- A list comprehension can have more than one for-block. That 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.
- In computing, the problem is always coming up with a way to do something just once. Once you have discovered some way that works, you can place a **loop** around that code to perform that thing however many times you want!
- 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, be it a list, tuple, string or whatever future sequence type anybody will think up.
- The same nested body of statements will be executed for every element of the sequence. However, unlike in a list comprehension, this body can be arbitrarily complex.
- A common situation has the processing of the current element depend on the value of both that current element and the element processed in the previous round. Maintain a local name (canonically named `prev`) for the element from the previous round of the loop. At the end of the body, the current element is assigned to become this `prev` element for the next round.
- Giving the variable `prev` some tactically chosen initial value instead of `None` may allow processing each element, first, middle, or last, with uniform code.
- 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.
- As a general rule, 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.
- Python offers **sequence decorators** such as `enumerate`, `zip`, `reversed` and `sorted`. These make your code more concise, readable and Pythonic.
- The sequence decorator `enumerate` should be used whenever the processing of the current element depends on **both the element value and its position in the sequence**.
- Since the body of the for-loop can contain any statements, other loops can be nested inside the loop body, same as with nested list comprehensions.

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 as lists and strings. These iterators consult that sequence every time they need to produce a new object, remembering their current position in that sequence.
- An iterator can be explicitly requested to produce its next item of the sequence with the Python built-in function `next`. If the sequence is complete so that no more values remain, this function raises the `StopIteration` error.
- Since this generation is **lazy** so that the next object of the sequence is generated only when requested, this will not run out of memory even for infinite sequences.
- Every iterator object remembers its **internal state** of where it currently stands in the production of its sequence. Even if the execution of the program moves on to do something else altogether, the iterator object will sit there paused in memory, waiting for somebody to ask for the next value in its generated sequence.
- The sequence may even be empty. **Whenever there is nothing to do, the correct thing is to do nothing.**
- Iterators in general do not support **random access operations** such as **slicing** the elements of the sequence at arbitrary positions, but the elements must be generated and processed in strict order from beginning to end.
- 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.
- Try to use operations that maintain the laziness of the argument sequences, such as `for` and `enumerate`, but not `len` or slicing.
- Iterators in general cannot know how many more items they will be producing, and therefore cannot play along in the `len` function calls.
- Once an element has been consumed from the iterator sequence, it is gone for good, since no memory is wasted storing the elements that the iterator has previously produced. Iterators do not support **rewinding** or restarting the sequence, but an entirely new iterator object has to be created.

4.3. Integer sequences as range objects

- Python's built-in function `range` creates a new object that lazily produces an integer sequence one number at the time. Compared to representing these integers inside a list, the memory use of `range` is extremely compact even for humongous sequences.

- The arguments for `range`, separated by commas, work exactly the same as for the string slicing operator. Note again the exclusive end index that is often our pitfall.
- Given just one argument `n`, `range` produces the sequence `0, 1, ..., n - 1`.
- Additionally, you can give the desired **stride** as the third argument to `range`.
- A negative stride makes the sequence count down instead of up. For example, the call `range(5, 0, -1)` would produce the sequence `5, 4, 3, 2, 1`.
- The idiom `for _ in range(n):` will execute its body exactly `n` times.
- These `range` objects use integer arithmetic to quickly determine that `-1 not in range(10**100)`, and that `len(range(10**10))==10**10`.
- In general, answering these same questions for arbitrary lazy iterators requires producing all objects one at the time to find out what happens, but the special case of `range` is easy enough to handle with integer arithmetic.
- However, since these quantities can be easily computed with integer arithmetic, `range` objects implement `len` and slicing efficiently even while maintaining their laziness. The other lazy sequences are not equally fortunate.
- 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** that represents the file inside the Python interpreter.
- The keyword argument `encoding` can be used to choose 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 *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 proceeds **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

- Sequences of fixed size whose contents are intended to be immutable are often more memory-efficient to represent as **tuples**.
- Tuples are also heterogeneous, and can contain any kinds of elements in them, including lists and tuples, although in practice this is rare.
- Most of the time ordinary parentheses are placed around the tuple elements, but in many cases this is optional. Parentheses are necessary only in two cases; creating a **singleton** tuple such as `(42,)`, and when a tuple is passed as an argument to a function.
- Tuples are **random access sequences the same** way as lists, so that all non-mutating operations work for them the same way as they do for lists and other sequences.
- 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)`.
- **Only the tuple object itself is immutable;** the objects that it contains do not magically also become immutable due to their containment inside some tuple. In general, since objects only ever contain **references** to other objects, the same object can simultaneously be "inside" many objects. (Unlike with those Russian nesting dolls, some sequence object could even be its own member without any paradoxical consequences!)

4.6. Tuple fiddling

- 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` without having to use a temporary third name `tmp` to first store one of these values.
- Tuples and other sequences can be **order-compared** to each other **lexicographically**, that is, in **dictionary order**. The first position where the two sequences differ decides the mutual ordering, regardless of the remaining elements.
- For example, `'aardvark' < 'zebra'`, and `(9,) > (1, 2, 3, 10**100)`.
- Comparing two tuples entails comparing their elements pairwise. Therefore, asking whether `(42, 99) < ('hello', 'world')` would cause a runtime error with its meaningless comparison `44 < 'hello'`.
- Sometimes we would like to break up the given tuple into its individual components, typically when we want to pass those components as separate arguments to some function. The call `foo(*a)` looks cleaner than `foo(a[0], a[1], a[2])`.

4.7. Members only

- Instead of keeping all your data in one unsorted list, many programming tasks become easier when aided with good **data structures**. Especially useful are **sets** and **dictionaries**.
- 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**.
- A set is especially powerful for the operations `x in coll` and `x not in coll` to determine whether `x` is a member of that `coll`ection.
- Determining whether an unsorted list contains the element `x` must iterate through the entire list to painstakingly compare its elements to `x` one by one. However, a `set` allows these queries to work at the blink of an eye, even for sets with millions of elements!
- The `set` data type also offers important **methods** `add` and `remove` to mutate its contents dynamically, and many others as documented in the Python library reference.
- 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.
- To directly iterate through the keys of a set, use `for x in coll` statement. In modern Python, this is guaranteed to go through these keys in their **insertion order**.

4.8. From keys to values

- The **dictionary** data type `dict` is similar to `set`, except that instead of storing individual keys, a dictionary stores **associations** to map **keys** to **values**. For example, a phone book could be implemented as a dictionary that maps names to phone numbers.
- The dictionary can hold at most one value for each key. Associating a new value to the same key erases its previous association.
- A dictionary from keys to integers can be used as a **counter map** that can be used to keep track of how many times we have seen something.
- Python dictionaries allow the square bracket indexing as handy shorthand for accessing keys and their associated values.
- Same as lists and other data types in Python, sets and dictionaries are **heterogeneous**. However, the data elements that are used as keys should be **immutable** to guarantee the correct behaviour of the data structure and its internal organization.
- Two special Python built-in functions `locals` and `globals` return dictionaries that contain the names in the current local and global namespaces, respectively.
- Of course, this internal organization comes with a price in both time and space, which is why the data types `set` and `dict` in the core language do not offer these operations. It would be silly to make everyone pay the premium for frills that most will never need.

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 jump to, but can get closer to with some simple steps that we know how to take.
- 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.
- Slightly oddly named `continue` statement will skip the rest of the loop body, and go directly to the next round of iteration.
- Python syntax allows **an `else` block immediately after a loop**. Such a seemingly nonsensical block of statements gets executed if and only if the execution of the loop reaches its natural end without encountering a `break` or `return` in its body.
- 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. Only if no counterexamples were found, we then execute the code placed in the loop's `else` block.
- 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 of the list are known to already be in sorted order, **binary search** is a much faster way to find an element in a sequence. (If the list is not sorted, it can always be sorted with much less effort than would be required to build a `set`.)
- Once the list has been sorted, 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 that guarantee finding the leftmost or rightmost occurrence of the given element.
- In the absence of the said element from the list, these functions return the place where that element would be correctly inserted to maintain the list in 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 effectively cuts the problem size in half by discarding half of the elements that were still in contention for becoming the final answer.
- For example, binary search in a billion-element list would require roughly only 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 250 times for a single elementary particle to remain. Within any realistic framework of computation, a sorted list big enough to make binary search to spend any noticeable amount of time simply cannot exist inside our physical universe.
- 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

- Since the body of either a while- or a for-loop can consist of arbitrary statements, and loops themselves are also statements, 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 how deep this nesting could reach. As the famous **zero-one-infinity principle** of computer programming states, the moment that you allow there to be two of something, you should actually allow any number of that something.
- This important design principle pops out its head all over software engineering in design, programming and execution.
- 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, same as we did with list comprehensions earlier.
- 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 other 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.
- Whenever the checksums of two things are different, that reveals that those two things are somehow different. Unfortunately, the checksum only reveals the existence of at least one difference, but does not tell us how many differences there are in total, nor allow us to pinpoint where exactly those differences reside in the sequences of returned results.

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 normal humans, and trivial to modify with any text editor and Unix command line scripts.
- Redundant whitespace can be eliminated from JSON files that will be read by machines.
- The JSON format expresses structured data using JavaScript lists and dictionaries, whose syntax by happy coincidence just happens to be exactly the same as that of Python.
- For security reasons, the **atomic elements** of the encoded structure in a JSON file can only be **strings, truth values, integers and floating point numbers**, an especially no executable code is allowed. This prevents various forms of **code injection** attacks.
- The atomic elements can be combined into lists and dictionaries with square brackets and curly braces. These lists and dictionaries can be nested arbitrarily deeply.
- 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. This makes it safe for the programmer to read any JSON files even if those files were dug up from the darkest swamps of our shared information superhighway.
- 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. This alone tells Python that when called, this function actually creates a new **iterator** object. The body of the function is not yet executed at this call.
- When this iterator object is asked to produce the next element, the function body starts execution. When this execution reaches a `yield` statement, the execution pauses and the yielded value is the next object of the sequence given out.
- The local namespace and the execution state of that generator instance remain in memory until the next element of the sequence is requested. Unlike an ordinary function that always starts from the beginning each time it is called, the generator object continues to exist in this paused state.
- When the iterator object is eventually asked to produce the next object of the sequence, the function execution resumes from the statement that follows the `yield` statement instead of starting from the beginning.
- The sequence ends when the generator reaches the end of its body. Some generators produce **infinite** sequences where this never happens.
- Multiple instances of the same generator can coexist in memory, each with its own local namespace and execution state. They can all produce their values independently of each other, instead of being forced into lockstep.

7.2. The itertools standard library

- Iterators can be a surprisingly powerful model of computation. Using the functions defined in the `itertools` standard library module, many functions implemented in traditional fashion with loops and conditions 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 without the aid of specialized additional hardware not included in our everyday off-the-shelf computers, **pseudorandom number generation** is done algorithmically.
- Pseudorandom numbers are not truly random. Once the **seed** value of the generator has been set, the random bits produced by such a 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 make their future sequences of random bits completely different.
- Modern pseudorandom generators generate "good enough" randomness in the sense that no program that uses such a pseudorandom generator would become more accurate in its intended task if that generator were replaced by an **oracle** that magically produces truly "random" bits.
- Careful design and mathematical analysis are required to construct a deterministic process that emits bits in such sufficiently random fashion. Randomly chosen methods will rarely produce sufficiently random results!
- Python 3 is standardized to use the **Mersenne Twister** generator, independent of the underlying operating system and physical hardware. This generator passes all major tests for quality randomness. Its **period is so** long that the sequence will never start repeating itself in any possible use case within the confines of our physical universe.
- The function `getrandbits` in the Python standard library module `random` functions is used internally by all other functions in that module to generate the random bits they will then combine into values from other probability distributions.
- Most programs need only one instance of a pseudorandom number generator, and should therefore access the functions of `random` as functions. However, some programs benefit from the existence of separate instances of the type `random.Random` that will advance independently of each other.

7.4. Random choices of integers

- Functions `randint` and `randrange` must surely be the most commonly used functions to produce random integers from the given range, such as rolling a die with the possible results between one and six.
- These two functions internally consult the `getrandbits` function for just enough random bits that they can produce their result integer value fairly from the given range, even if this range is not some convenient power of two, or if the width of that entire range were somewhere in the high 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, once that underlying self-similarity has been spotted. Spotting the self-similarity is the most difficult step of recursive problem solving in practice, but once the self-similarity has been revealed, the rest is nearly mechanical.
- The exposed self-similarity allows us to write **recursive definitions** for self-similar phenomena that we wish to compute out.
- For example, the non-recursive definition for factorial $n! \triangleq 1 * 2 * \ldots * (n - 1) * n$ can be converted 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 recursion is that there is no secret**: nothing happens in a recursive function call that would not also happen in a non recursive function call.
- Each recursive invocation of the same function has its own local namespace. This allows the same names can exist independently at different levels of recursion.
- Also, any **linear recursion** where each function call produces at most one more function call can always be converted to loops and iteration in a straightforward fashion.
- The true power of recursion is unleashed in its ability to **branch** to two or more different directions, going as far and deep in each direction as needed before trying out the next direction, possibly branching even further to easily explore an exponential number of possibilities. Contrast this to the while-loop that will only keep going to one direction, missing the goal unless there exists some goal along this straight and narrow path.
- Every recursive function must begin by checking whether it has reached a **base case**, where that subproblem can be solved on the spot without any more recursive calls.
- A recursive function can have more than one base case, even infinitely many.

8.2. Downsides of recursion

- Recursion can be a powerful friend, but it comes with **three undeniable downsides**.
- Every function call, recursive or not, always creates a new namespace of local names. A deep recursion doing the work of a loop will cause a **stack overflow**.

- Deep **linear recursions** should be implemented with loops that execute inside the same stack frame, instead of using recursion at all.
- 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.
- (The odd spelling "memoize" for this technique instead of "memorize" is intentional, to clearly identify it as the technical term that refers to this particular technique.)
- Memoization is philosophically similar to solving a problem with a **lookup table**, except that the lookup table is now filled dynamically as encountered and needed, as opposed to being hardcoded into the program.

8.3. Applying functions to other functions

- In 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 function that sorts a list can be given an optional **key function** to determine how each element should be treated as in order comparisons.
- The key function can easily implement **sorting by multiple criteria** by having this key function return a tuple. This produces the desired effect, since the Python tuple order comparison is lexicographic.
- A **function decorator** is analogous to sequence decorator in use and spirit.
- The function decorator `lru_cache` in the `functools` module can be applied to any function to automatically **memoize** it to remember the previously computed results for quick lookup.
- The abbreviation prefix `lru` stands for "least recently used". Should the memoization data structure ever become full, the result that has been previously queried the longest time ago and therefore probably will not spark joy in the future is tossed out to make room for storing the next result, in the best spirit of Marie Kondo.
- 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. Defining small functions as lambdas

- It would often be handy to define a small one-liner function on the spot, without having to go through the whole rigmarole of the function definition syntax.
- An **anonymous function**, which are for historical reasons called **lambdas**, can be defined using the keyword `lambda`, followed by the parameter names separated by commas, followed by a semicolon and the function body.
- For example, the expression `lambda x,y: x*x+y*y` defines an anonymous function object that takes two parameters, named `x` and `y` inside the function, and returns the sum of their squares.
- For syntactic reasons, Python lambdas are restricted to one-liners. If you need a more complicated function, you have to define it as a proper function.
- PEP 8 advices that you should only use lambda expressions inside some larger statement, and never uas a substitute for `def` for a function declaration.
- In many situations, you would like to pass some operator of the Python language itself, for example `+` or `<`, as an argument to some other function. You cannot do this directly within the Python syntax, as the language low-level operators are not objects that could be passed back and forth, so Python will only give you a syntax error message trying to apply the operator directly to its surroundings in the source code.
- To achieve this, you have to first wrap that operator inside a lambda expression, such as `lambda x,y: x+y` for addition.
- The module `operator` defines convenient named lambda expressions for all built-in operators in the Python language.

Module 10: Efficient numerical computation

10.1. The numpy array data type

- The flexibility of Python data structures such as heterogeneous lists and dictionaries can be inefficient in both time and memory when dealing with large datasets.
- This flexibility is seldom needed when dealing with real world data known to be some simple uniform types such as integers or decimal numbers, especially when they are known to be inherently restricted to some limited range of possible values.
- 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 will be of the same numerical type, they can be stored in memory compactly consecutively.
- Inside the `ndarray`, each element has a fixed maximum value that depends on the `dtype` of that array.
- If some computation produces a larger value than fits into the fixed space allocated to it, the result **silently overflows** and is truncated to fit inside those bytes in a cruel Procrustean fashion.
- Integer elements can be defined to be of type `intX` (**signed**) or `uintX` (**unsigned**) where `X` stands for the number of bits used, either 8, 16, 32, or 64. For example, the element type `int32` would denote a signed four-byte integer.
- Unsigned types offer twice as large a range of positive 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 a single byte, thus offering the possible ranges of -128, ..., +127 and 0, ..., 255 respectively. (Signed integers encoded in **two's complement** always have one more negative value than they have positive values, zero being the "odd man out" in the middle of this discrete range.)

10.2. Operations over numpy arrays

- The **universal functions** defined in numpy modules operate **element-wise** on the entire `ndarray` at once.
- As a general principle, whenever you are using Python for-loops or other methods outside numpy to iterate over numpy arrays, you are most likely doing something wrong, or at least not anywhere close to the best possible Pythonic way of using numpy.

- 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, so that the matrix addition can be performed element-wise.
- As a special case of broadcasting, any **scalar** value can be thought of as a 1-element numpy vector that broadcasts itself into any number of dimensions.
- Matrix multiplication in the sense of linear algebra using the `dot` product of individual vectors is another special case of broadcasting.
- From the outside, T=the `ndarray` data type is looks and feels like an iterable Python object. However, slicing produces another **view** to the same underlying data, instead of copying the data to the new `ndarray`.
- Extensions have been written on top of numpy to perform even more advanced operations on `ndarray` objects. The most important of these 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. These days, the biggest buzz there is surely about scikit.learn that offers a host of **machine learning algorithms** for anybody to discover interesting underlying regularities in their own datasets.

10.3. Processing pixel images as numpy arrays

- Mathematically, a **pixel image** is a three-dimensional **cuboid** of numbers. Two dimensions represent the pixel coordinates, and the third dimension encodes the total colour of that pixel in either one or three **colour components**.
- The vast majority of times this representation being **RGB** (red, green and blue) and occasionally **HSB** (hue, saturation and brightness). RGB is used directly in the computers and their display hardware, but HSB and similar encodings are more intuitive for humans to think about colours and perform transformations on them.
- To encode a **grayscale image**, the colour dimension needs only one brightness value, and the entire image is therefore a two-dimensional `ndarray`.
- The colour component intensities are given either as floating point numbers from 0 to 1, or as **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. (After all, that is what the Adobe engineers did and still continue to do, although using a very

different programming language!) To get us going, a whole bunch of **filters** and other common image operations have been implemented in the `scipy.ndimage` module.

- **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.
- If the convolution is not supposed to cause the pixel intensity of the image to overflow, the kernel matrix elements should add up to 1.0 to maintain the intensity of the image, but this is not any kind of law of nature or man.

Module 11: Classes and objects

11.1. Thinking in abstractions instead of implementations

- The modern **object oriented** paradigm allows programmers to define brand new data types as **classes**, along with the operations for that data type.
- During the execution of the nested `class` statement, the statements inside it are executed normally, but whatever names they define are stored in the namespace inside the class object that is being defined, instead of in the usual global namespace.
- Executing the nested `class` structure creates a new **class object**, of which an arbitrary number of **objects** can later be **constructed**.
- Functions defined inside the class object are called **methods** that can later be called for the individual instances of your class. These methods are defined as functions, but their first parameter `self` refers to the object for which the method is being called.
- A class can also have **class attributes** to represent data that is the same for everybody throughout the class, instead of being stored separately for each individual object.
- For example, inside a hypothetical `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.
- Python does not support **encapsulation** by dividing the attributes into **public** and **private** subsets so that only the public attributes would be accessible from the outside.
- 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

- To create a new instance of the class `Foo`, just use `Foo` as a function.
- The arguments that you pass to this function are passed on to the special **constructor** method `__init__` that gets automatically called for the new object every time a new object is created. This method typically creates and assigns the new object's internal attributes, in addition to performing whatever other initialization work is necessary.
- 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. (In your own data types, `__str__` is a good method to define right away to make your debug printouts human-readable.)

- Whenever the Python compiler sees the code using some built-in operator of the language, such as `a+b`, it silently turns it into the expression `a.__add__(b)`.
- It is the programmer's responsibility to ensure that `__add__` and others do the right thing for that particular data type.
- Many operators impose an implied **contract** on the corresponding dunder methods so that these methods ought to behave a certain way. For example, the order comparison `__lt__` ought to be **transitive** so that whenever `a < b` and `b < c`, then also `a < c`.
- If these implied contracts are violated, other classes and data structures can silently start behaving in weird ways, resulting in difficult bugs in your program.
- To define your own **iterator** type as a class, simply define the methods `__iter__` to create and return an iterator object (for many types, this method can just `return self`) and in that object, the method `__next__` to either produce the next item of the sequence, or raise `StopIteration` if there are no more elements.
- However, these days it is far easier to write a **generator**, especially if this iterator does not need to allow external access and modification of its internal state.

11.3. Patching in new properties

- If only one particular object within its type needs to do something special, **monkey patching** reassigns the name of some method to refer to some other object than which that name is bound in the class object.
- Alternatively, some special object can be given additional special attributes that the other objects of the same type do not possess in general.
- Python 3 allows some attributes to be defined as **managed properties** using the decorator `@property`. From the outside, these 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 when inside the same `Temperature` object, both `K` and `C` are accessed as attributes, even though one of them is in reality 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 existing concepts

- The power of object oriented programming comes from **inheritance** relationships between classes that allow variety while enforcing certain consistency within all objects constructed from the same class hierarchy.
- To express that your class is a **subclass** of another class, write the superclass name in parentheses after the name of the class. For example, `class Car(Vehicle)`.
- Inheritance is used to denote **subtyping** between the problem domain concepts that the classes represent inside our program, in the sense that 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*.
- Whenever some name is accessed inside some object, and the object itself does not contain that name in its internal dictionary of names, the Python virtual machine climbs up the inheritance chain of classes to find the first available occurrence of that name. This way, each subclass needs to define only the functionality that is new in that subclass, and **inherit** everything else from its superclass.
- When defining a subclass, you can **redefine** its method names to provide a different implementation for them. This allows instances of different subclasses to respond to the exact same request in different ways, depending on their particular types.
- **An abstract superclass** represents some 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. For example, we can imagine and describe what kind of sound a dog or a chicken makes, but we cannot similarly describe what kind of sound an animal makes, without knowing what specific subtype of animal that particular object happens to be.
- 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.