Python Lecture Notes by Ilkka Kokkarinen

These notes go through the important things about Python programming language as taught by Ilkka Kokkarinen. These bullet points list the important ideas, but the example programs listed on the course outline are equally important and necessary to see how these ideas work in practice. This document and all of the instructor's example programs use Python 3 throughout (iterators, Unicode strings, formatted string interpolation), and are not compatible with the older Python 2.

Expressions

- When used in the **interactive mode**, the Python interpreter reads in **expressions** entered by the user one at the time, and prints out the results of evaluating them. This makes Python handy for small scale experimentation and trying out things.
- Expressions can be built up from **constant literals**, **arithmetic operators** and **function calls**. Some expressions are `"Hello" + "world!"`, `2 + 2`, `5 / 2`, `5 // 2`, `2 ** 3` or `5 > 8`, these respectively evaluating to `"Helloworld!"`, `4`, `2.5`, `2`, `8` and `False`.
- Some functions and expressions can have **side effects** that produce output or change the data stored in the memory in a way that can affect the evaluation of later expressions.
- For no rhyme or reason, `%` has nothing to do with percentages, but denotes **integer remainder** operator. In all sanely designed languages, the sign of the remainder is always the same as the sign of the first operand, regardless of the sign of the second operand. In Python, the remainder operator works in a different way that has one giant advantage in practical programming.
- The **precedence** and **associativity** of mathematical operators behave as you learned in basic mathematics, and can be bypassed with parentheses.
- Python can even handle **complex numbers** and their arithmetic operations directly built in the language, with the **imaginary part** denoted with the capital `J` after the number. For example, the expression `(1 + 2J) * (1 - 1J)` equals `3 + 1J`.
- Python internally stores integers in 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 `1234**5678` and other similar humongous integer power behemoths become a breeze to compute.

- Every expression in Python evaluates to some **object**, although this result can also be the special 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. However, instead of trying to write a complicated computation as a **one-liner**, it is in many ways better to break it down into a series of simpler expressions executed in sequence.

Scripts

- When given an entire **script** to execute as a **program**, Python does not automatically print out the results of the individual expressions (since you certainly don't want your screen flooded with the results of possibly billions of evaluated expressions), but make these results visible for the user only when the expression contains a call to the special `print` function that outputs its argument as characters.
- A Python script file absolutely must consist of **raw text**, only the actual Python source code characters inside it and nothing else. Any kind of **rich text document** (for example, Microsoft Word or Google Docs documents) that contains formatting instructions such as font or paragraph styles do not work. As a rule, if opening the script file directly into Notepad/TextEdit shows any weirdness, your script file is not legal Python.
- (When displaying such a raw text Python script, all modern text editors use **syntax highlighting** with colour and font effects to make the structure of the code easier to discern for the human eye. However, all such effects are dynamically generated by the editor for its display, and are not encoded inside the actual raw text script file.)
- Many built-in functions such as `print` are **parameter polymorphic**, that is, the same function can be applied to arguments of vastly different types, and the function still does the right thing in each case. For example, `2 + 2` equals `4`, and `'hello' + 'world'` equals `'helloworld'`, the operator `+` doing the right thing in both cases.
- Python is **case sensitive** everywhere, same way as most other programming languages. However, Python differs in style and spirit from most other programming languages in that it actively uses **whitespace** and **indentation** to denote the structure of the script.
- Almost all other major programming languages are designed to be insensitive to whitespace, and therefore need special parentheses, typically **curly braces**, to denote the program nested structure.
- 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.

- (Statements can also be separated by **semicolons** in the style of Java and similar languages, which allows packing several small statements on the same line to save vertical space. In practice this is quite rare, and in fact is considered a rather bad style in Python.)
- A common oversimplification is to proclaim that Python language is not "compiled". More accurately, the programmer does not need to explicitly compile the Python **source code** script to a form executable by machine. Instead, the interpreter will compile the read in script into a more efficient internal representation before executing it, and while doing this transformation, check that all expressions in this script are **syntactically legal**.
- The result of this compilation is stored in a Python bytecode file with suffix `.pyc` for future use. Conventionally, Python source code files are named using suffix `.py`.
- So that other people (which includes all the near and far future versions of yourself) can understand what your code does, write **comments** to explain the purpose of the code. Python comments start with the octothorpe / pound sign / hashtag character (depending on which decade you grew up in) written as `#`, and continue to the end of the line.
- **Always comment on the why, never on the what.** The Python code already perfectly describes what it does, so comments should explain why you wrote that code. If your code is so complex that it needs additional explanation of what it does, you should redesign that code to be less clever. (In programming lingo, calling something "clever" is often done in a sarcastic fashion.)
- For those interested, originally written for Java and C++, the humorous document "How to Write Unmaintainable Code" applies just as well to Python programming.

Names and objects

- An important mechanism needed in every programming language is the ability to give **names** (also called **variables**) to objects so that they can be used in later expressions. This is done using an **assignment** such as `x = 42` or `bob = "Hello"`.
- Names can consist of arbitrary **letters** (in practice, you avoid a lot of trouble by using only the twenty-six letters of English), digits and underscore characters. However, a name may not start with a digit. (Names that start with double underscores have a special meaning explained later.)
- Python doesn't care one whit what these names "mean", but of course you should aim to refer to your objects using names that are meaningful and informative for human readers.
- In the Python interactive environment, the special name `_` that is merely 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.)
- Unlike in **statically typed languages**, names and their types don't need to be explicitly declared, but each name comes into existence into your current **namespace** at the first

assignment to it. The name will then refer to the object that was assigned to it. Trying to use a name that has not yet been created crashes the script with an error.

- A **namespace** is a data structure internal to a Python virtual machine that keeps track of all the names that exist inside it, and which objects those names refer to. Unless otherwise specified, declared names appear in the **global namespace**.
- **Each name is a separate thing from the 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 object still continues to exist in memory, and might be simultaneously referred to by other names (**aliasing**).
- The relationship between names and objects is **unidirectional**: the name grants access to the object associated with it, but there is no easy mechanism to go back from an object to all the names that point to that object.
- Names can be erased from the namespace with the `del` function. If the object is still pointed to by some other name, it will continue to exist as before.
- Once an object becomes **unreachable** from the names that currently exist in your reachable namespaces, the **garbage collection** mechanism of Python interpreter releases its memory for future use. This way you never (or, well... perhaps more in tune with Gilbert and Sullivan, "hardly ever") need to worry about releasing these objects yourself.
- To find out what names exist in the current namespace or inside some object, use the function `dir`. Despite the name, this function has nothing to do with filesystem or disk access, but lists the names that exist in the current **global namespace** in memory inside the Python interpreter process, or some particular object given as argument.
- Alternatively, the built-in function `globals()` returns the list of global variables.

Assignment operator

- The assignment operator `=` evaluates its **right hand side**, which can be an arbitrary expression, and then makes the **left hand side** name point to the object produced by the right hand side. The previous value of the name on the left hand side ceases to exist and cannot be recovered in any way.
- Note what a completely different thing it is to say `a = b` than to say `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 beginning programmers.
- **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**.
- After assigning `a = b`, later reassignment of `b` to something else does not change the value that `a` points to.
- Names allow us to talk about objects in memory, since objects themselves don't have any inherent names. (Some programs could create millions of objects during their run, and

you certainly don't want to think up a new distinct name for each and every one of these objects in your source code!)

- The document "Facts and myths about Python names and values" by Ned Batchelder clarifies the distinction between names and the objects that those names refer to. This difference is not really important in everyday programming where we can oversimplify and naively equate the object and the name that we have given to it (after all, this is how we basically deal with mathematical objects such as integers, for example, or even with other people), but understanding this distinction allows us to easily explain some situations that would otherwise be baffling.
- To find out the **memory address identity** of a Python object, use the function `id`.
- The function `input` reads in a line of text input from the user. Since your program has zero control over the user and his 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"` just to see if your program is prepared for that possibility.
- (As an aside, computer security 101: you cannot assume anything about data that is generated by and comes from the outside world. "Client is in the hands of the enemy.")
- (As another aside, **pretty please never use "input" as a verb**. Doing so will always only look stupid, especially when done in past tense. **And certainly you can't use the same verb "input" for both directions**, as in the two sentences "The program inputs a number" and "The user inputs a number". The correct verbs are "**read**" and "**enter**", as in "The program reads the input that the user enters.")

Types

- Contrary to the common misconception that for some reason is widely floating out there, Python is a **strongly typed** language so that every object in memory has a **type** that determines how that object behaves and what operations are possible to do with it.
- However, Python typing is **implicit** in that the language itself does not talk about the types of objects that it performs computations on. The Python interpreter will simply infer the type of the resulting object from the structure of the expression that was used to create that object.
- For example, the expression `2 + 2` would clearly produce a result object that is an integer, whereas `2 < 3` would produce a **truth value**.
- In reality, all data in computer memory consists of **bytes**, small "boxes" that contain a small integer number, but don't contain any inherent meaning about the type of the thing that they represent, or what they are interpreted to semantically mean in aggregate. In principle, the exact same bytes in the memory can be treated as integers, text, machine code instructions, or whatever else.

- This is the grand idea of **von Neumann architecture** that all real computers follow: there are only bytes, and the processor that performs simple copying, comparison and arithmetic operations on those bytes. Anything above that, such as "decimal numbers", "functions" or "conditions", is just as fictional as hobbits or vampires. However, as long as everyone agrees on the properties of these fictional objects, these useful fictions can be used as notational shorthands to express truths about phenomena that are very real!
- (Strictly speaking, even "bytes" and "processor" are useful fictions, just one level of abstraction below the previous one. In the actual reality, inside a computer there exist only cleverly arranged logic gates, through which electric current flows...)
- **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!
- To test whether some object is of the given type, use the operator `isinstance`.
- Python has many built-in functions to convert an object to a different 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 **semantically nonsensical** expressions such as `int("Hello")` ? When an expression cannot have any meaningful possible result, Python **raises an error** instead.
- Python does optionally allow **type annotations** that do not affect the compilation or execution of the code itself, but allow outside tools such as mypy to perform additional **static code analysis** and other computations, and also make it clear to the human readers that, for example, some function is supposed to receive a text string as an argument, but never an integer or a dictionary.
- Type annotations go against the Pythonic spirit of flexible **duck typing** that says that the actual type of the object is not important, but the capabilities of that object in what it can do. If it walks like a duck and so on, that object can be given to any function that expects to be given a duck, even though that object is not really a duck but something else.

Imports

- Python famously comes with "batteries included", so that in actual programming you don't want to be constantly reinventing the endless wheels spinning inside larger wheels, but **import** as much of existing stuff as possible.
- The `import` statement is used to execute the script (whose name is given without the filename extension) so that the names produced by the script are stored in a new namespace that becomes a module object in your current namespace.

- To access some name inside a module object, use the form `module.name`. The same syntax can be used to access a name inside any object, and modules are objects inside Python the same way that integers, strings and files are objects.
- Python is super flexible in that **every user-defined object contains an internal namespace** in which new names can be freely added and removed. Objects and their properties can be modified during runtime in near complete anarchy, so that "do what thou wilt" shall be the whole of the law here.
- (If some statically typed language weenie insists that all computational structures should be set in stone at compilation to save some microseconds of computation time and begins to air out any objections to this, just respond "Types, schmypes!")
- For example, complex number objects have internal names `real` and `imag` that allow you to access the real and imaginary parts of the complex number.
- Python does not have any kind of proper **encapsulation mechanism** to make some names **private** so that they could not be accessed from outside the objects. Almost all other programming languages created within the last three decades have such encapsulation, but Python is more relaxed in the spirit of "we are all friends and consenting adults here".
- For names defined inside module objects and other 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. The outside entity who chooses to access such a name takes the responsibility of whatever side effects and future incompatibilities this may cause.
- Using a **double underscore** is even more common, as this makes the name to behave properly in class inheritance that we will get on in the last lecture. In fact, you can use a double underscore to denote a name that is intended to be an implementation detail.
- If you need only some particular function from a given script, you can use the alternative form `from Foo import bar`, where `Foo` is the name of the script, and `bar` is the name that you wish to import. The script is still executed first in a separate namespace, but afterwards, the name bar is created to your current namespace and set to point to the same object that the name points to inside the module namespace.
- The **wildcard** version `from Foo import *` imports all names defined by the script. However, you should use this form only if you are certain that none of the names of the module clash with the names that you already use, since all the names inside that module are flooded into your current namespace. In practice, this form is used only for importing modules that define only one or two names inside them.
- The `import` mechanism is smart enough to remember which modules have already been previously imported. If some module is imported again the second time, the mechanism does nothing and simply reuses the previously generated namespace. (This is important in that the internal initialization logic of each module is executed only once.)

Some handy library modules

- Important library modules such as `math` provide functions needed in many programs. (For complex numbers, there is an equivalent library `cmath`.)
- Python stores decimal numbers internally using **floating point** in 64 bits per number. The floating point is an excellent way to represent a wide range of decimal values with dynamic precision that gets less granular the further away you get from zero.
- The floating point encoding also has special values for **positive and negative infinity**, and the special value **NaN** to indicate that the arithmetic operation does not have a meaningful result.
- However, the floating point encoding cannot represent exactly even some numbers that we humans would consider "simple". For example, the expression `0.1 + 0.2` does not equal `0.3`, but equals `0.30000000000000004` instead. This is because 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 exactly in **base ten**.
- When performing arithmetic calculations, the floating point arithmetic will always produce the representable result closest to the true result.
- This is not Python's fault, but inherent in the floating point encoding itself. Floating point arithmetic is done on the processor hardware anyway, with Python merely reporting the outcomes. Changing the encoding would not help either, since using 64 bits, there exist at most 2^{64} **representable values** out of the infinite continuum of real numbers, so the vast majority of real numbers cannot possibly have a fixed size floating point representation.
- (Yes, Virginia, there also exists an infinity of integers, but unlike real numbers, **integers are discrete**, so we can provide a unique encoding as bits to each one up to some arbitrary but fixed limit. The higher you make this upper limit, the more bits you need to encode each integer value.)
- No matter what base b you choose to encode your numbers, the fraction $1/n$ can be exactly represented in base b only if all prime factors of n are also prime factors of b. With computers, $b = 2$. (In the sixties, Russians reputedly experimented with **ternary** base $b = 3$. Fill in your own "In Soviet Russia..." joke.)
- The library module `decimal` defines the data type `Decimal` for arbitrary precision decimal numbers expressed in base 10. The `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 produce different precision results in arithmetic operations.
- Another 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.
- To generate random numbers from uniform distributions or to sample and choose random items from sequences, 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 real life applications.
- The modules `os` and `sys` define functions that allow your code to access the underlying computer and its file system.

Defining your own functions

- A **function** is a **named block of statements** that performs some operation that is intended to be performed more than once. Functions are objects in Python, and can therefore be assigned to names the same way as strings or integers.
- Function objects are defined using the special `def` statement that creates the name of that name to the current namespace, and makes this name refer to the object that contains the function body, converted to the bytecode form executable by Python.
- When a function is **invoked** (**called**), the statements in its body are executed in the order that they were written into the program source code. 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.
- As long as the caller receives the correct result produced within a reasonable efficiency of execution, they shouldn't even want to care how exactly the function produced that result anyway, but can treat the function as if it were a statement in the language itself.
- As you write more functions, you essentially extend the language to contain higher level operations that can then be used to define even higher level operations, thus building a complex program **bottom up** from **simpler pieces that you are able to understand in isolation**. This is important once the problem domain becomes so big that it is impossible to implement as a single function, but must somehow be sliced into smaller pieces.
- A function can expect to be given **parameters** whose values are given at the call as **arguments**. The caller is responsible for providing argument values that are meaningful for the function and its execution. Even if a function takes no parameters, the syntax for its call still requires an empty pair of parentheses, since 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** a result at the end. If no result is explicitly returned using the `return` statement, the special value `None` gets automatically returned.
- The `return` statement can also be used to conditionally terminate the execution of the function body as soon as the result becomes certain, without need to fool around until the execution reaches the end of the function.
- Any 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 from the call, assuming that there are no names in the caller's namespace that either directly or indirectly refer to the names of the local namespace. (If there are such names, perhaps inside the function object created in and returned from the function, the namespace continues to exist in memory as a **closure**.)

- The keyword `global` used inside a function causes its following name to be created and accessed in the global namespace.
- If the first statement inside a function is a triple quoted string, it is treated as a **docstring** that documents what the function is supposed to do. Many editors and other programming tools can properly handle and display function docstrings automatically.

Decisions

- Most functions need to be able to make **decisions** so that they behave differently in different situations. **Binary two-way decisions** are the fundamental building blocks that all computational operations are internally composed 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.
- A problem is computable if and only if it can be broken down into a tree structure of two-way decisions, possibly looping back to an earlier decision. **Flowcharts** are a human-friendly way to visualize such a decision structure, usually leading to more or less humorous conclusions.
- In higher level languages such as Python, same as in almost all other programming languages, binary two-way decisions can be made with the `if-else` statement.
- The `if-else` statement executes precisely one of its branches depending on whether its **condition** is `True` or `False` at the time that the execution reaches the statement.
- Often, you don't have anything to do when the condition is `False`. The `else`-branch is optional and can be left out completely in such a situation. (Since the syntax of Python does not allow an empty block of statements, the otherwise redundant do-nothing statement `pass` would otherwise have to be used for this purpose.)
- To choose from three or more possibilities, you have to break down this decision tree into some series of two-way decisions. As follows from the properties of **propositional logic**, this can always be done in many different ways, either nested or sequential, that are equivalent in the results that they produce. Your programming and thinking styles ultimately determine which of these equivalent ways to organize a **decision tree** you end up writing.
- As an additional shorthand to embed a small expression conditionally inside a larger statement, you can use `expr1 if cond else expr2` that evaluates to either `expr1` or

`expr2` depending on `cond`. (This roughly corresponds to the **ternary selection operator** found C or Java or similar languages.)

- Technically, the expression used as a condition does not have to evaluate to either `True` or `False`, but can be a value of any type. Other data types have predefined rules for which values are considered "truthy" and "falsey" depending on which result the conversion function `bool` produces from them.
- For numerical types, zero values are falsey, others are truthy. For sequences, sets and dictionaries, objects of zero length are falsey, and all others are truthy (yep, even the singleton list `[False]`).
- This convention allows writing more terse Python code, but in many situations, writing the comparison explicitly such as `x != 0` instead of `x` for an integer parameter `x` makes the code easier to read.
- On the other hand, if `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.

Combining simple conditions to create complex conditions

- Truth-valued conditions can be created with the comparison operators `<`, `>`, `<=`, `>=`, `==` and `!=` that compare the operands given to them. Note that the equality comparisons are done with `==` instead of `=`, since the latter operator already stands for assignment.
- This silliness remains in basically all programming languages today. If only `=` stood for equality like it does everywhere else in our lives, and assignments were denoted by some kind of left arrow character to make it plain for all what happens...
- The operator `==` compares its operands **for their content, not their identity**. The `is` operator checks whether its operands are the very same object in the same memory address. Therefore, the test `a is b` could equivalently be written as `id(a) == id(b)`.
- In Python, comparisons can easily be **chained**, in the style of `a < b < c > d`, instead of having to use the logical `and` operator to combine individual comparisons. In other programming languages, this condition would have to be broken down into a series of individual conditions `a < b and b < c and c > d`.
- As you can see in the previous example, more complex conditions can be built up using the propositional logic operators `and`, `or` and `not`. When building up these more complex conditions, it is often useful to clarify the logic with parentheses.

- Unless the logic of your code is redundant to begin with, the operators `and` and `or` are **never** interchangeable. Replacing one with the other will always change the behaviour of your function.
- The operator `and` has a higher precedence than `or`, and `not` has the highest precedence. Even so, **it is never wrong to add redundant parentheses to an expression to clarify your intent about what expression should do.** The Python compiler uses parentheses only to direct its **parsing** algorithm, and the parentheses do not appear anywhere in the generated low level executable.
- Both operators `and` and `or` are guaranteed to **short-circuit** so that if the first operand already determines the result, the second operand will not be evaluated at all. This is very important if the second operand has **side effects**, or could potentially crash the program.
- All the discrete math intro stuff about **propositional logic** is in full effect, should you happen to know any of that. Most importantly the **De Morgan's rules** for simplifying negations and double negations, which can be just as tricky in programming languages as they are in natural languages.

If-else ladders

- A multiway decision from a known fixed number of possibilities is often best written as an **if-else ladder**, a generalization of a single if-else, with the keyword `elif` denoting the individual steps from the second step onwards.
- Writing multiple if-statements separately in sequence makes them all to be executed independently of others, so that none of them, all of them, or anything in between, could be executed. But in an if-else ladder, **exactly one of the steps will be executed.**
- 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. The execution then skips the remaining branches even if their conditions also happened to be `True`.
- The order in which you write the steps of the if-else ladder is therefore extremely important, since rearranging these steps will generally change the behaviour of the ladder and the function that it computes.
- When designing an if-else ladder, you can write each step and its condition with the airtight guarantee that all previous conditions have been `False`. (Had any one of them been `True`, the execution would not have reached the current step!) Therefore, you don't need to test anything that you have already tested in the conditions of the preceding steps. Knowing this may allow you to simplify the later conditions, especially if you arrange all the steps in some clever way to begin with.
- An if-else ladder whose last step is not an unconditional `else` is perfectly legal, but it still seems hinky and is often somehow wrong. (This is not an airtight rule, though.) Such

a ladder will then behave as if there were an unconditional else in the end whose body simply does nothing at all.
- Avoid the temptation of writing an else-block that does nothing at all. The else-block is optional and can be left out altogether when there is nothing to do in the situation where the condition is false.

Text strings

- In reality, computers only store small integers inside memory **bytes**. Any other type of data must be internally expressed as an aggregate series of bytes. There are standardized ways of doing this for **floating point decimal numbers** and text strings.
- Python 3 stores characters and text strings properly in **Unicode** for perfect portability between different computer systems. The exact internal representation of these Unicode strings is hidden from the programmer who should not need to worry about it anyway.
- To create a **text string literal** inside an expression, just write characters inside either single or double quotes. **Escape sequences** starting with the backslash character are a standard way to include 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 is limited in the source code with that quote character.
- To embed an arbitrary Unicode character inside your string, use the extended escape sequence `"\uXXXX"` where `XXXX` is the Unicode codepoint of that particular character expressed in four digit **hexadecimal**, amply available on Unicode tables online.
- String literals that start with **triple quotes** can span multiple lines. Line break characters (yes, they are characters just like any other Unicode character) are taken as part of that string literal that continues until the closing triple quote.
- To disable the escape sequence mechanism that allows you to embed arbitrary characters inside the string literal, define that string literal as **raw string** by prefixing its first quote mark with letter `r`, sort of like the opposite of the f-strings below. (This will come in handy for **string pattern matching or manipulation** using **regular expressions** ("regexes"), since the regex syntax tends to be quite heavy on backslash characters.)
- The **byte array** data type of Python gives you access to the exact byte sequence that is used to encode a Unicode string or some other higher-level type. This is as close to data of raw memory bytes that we can get to in Python.
- Unicode strings can be converted back and forth to byte arrays with their methods `decode` and `encode`. This allows us to take a peek at how exactly the Unicode characters, each taking between one or four bytes of memory to store, are internally stored in a Python string.

Building up string literals from smaller pieces

- The easiest way to build up a long string from smaller pieces of strings is to use the operator `+` to concatenate these pieces together, in the style of Java programming language.
- Especially useful for console output, **formatted strings** with the `f` prefix replaces **format placeholders** denoted inside the string with curly braces with the evaluated value of the expression inside the braces. These are most commonly seen inside a `print` statement, but they can be used anywhere that you want to define a string literal.
- When printing out floating point numbers, always **round** them to some reasonable number of decimal places. For example, `print(f"{0.1 + 0.2:.1f}")` correctly prints `0.3`, even though the actual computed result is something slightly different.
- Alternative ways to achieve the same thing would be the old-time C-style string interpolation with the `%` operator, which for string objects means format substitution instead of integer division remainder, or using explicit `Template` instances.
- 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. Furthermore, the library modules `string` and `re` (for **regular expressions**) provide even more powerful operations.
- For purposes of output, non-string objects such as integers, truth values and higher level entity objects can be automatically converted to a string representation. For example, the integer object 42 is converted to a two-character string `"42"` when printed. Make sure that you understand that **the string representation is used only for output**, but the integer object 42 in memory does not consist of any characters at all, 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.
- Alternatively, the function `repr` produces a string that when evaluated as an expression using the built-in function `eval`, produces an object that is identical to the one whose representation was acquired with this function call.
- However, `repr` cannot always be meaningfully defined for all objects. Python convention to document this fact to the users of this function without throwing any kind of error is to **begin and end this representation string with angle brackets**, which is enough to guarantee that any attempt to evaluate this string will crash with a syntax error.

String slicing

- A part of the string, or any other **sequence** data type in Python, can be extracted by **slicing**, denoted with the **square bracket** operator.
- Inside the sequence, the character **offsets** from the beginning are numbered starting 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 offset is therefore always exactly one less than the length of the sequence. If the sequence is empty, it has no legal offsets at all to access its elements.
- To extract a single character from a string, use square brackets with one offset.
- To extract a longer substring than just once character, give the **start** and **end** offsets separated by a colon character, together creating a **slice**.
- Note that **the end offset of a slice is exclusive**, one step past the last character that you want to include in the slice. This initially perhaps a bit strange convention makes some mental arithmetic on slices easier, such as computing the length of the slice without the danger of **fencepost error** to create an **off by one error**.
- ("If a fence is 50 meters long and has a post at every 10 meters, how many posts does that fence contain altogether?" No, the answer is not five. Count them again, using your fingers if you need to.)
- Negative offsets would be impossible by definition, akin to asking "What letter comes before the letter A in the standard alphabet?" Therefore as a handy convenience, Python uses negative offsets to denote the **positions starting from the end** instead of the beginning. For example, the expression `"Hello"[-4:-1]` evaluates to `"ell"`. (Even with a negative offset, the end index of a slice is still exclusive.)
- **Step size** can be given as 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"`, for the canonical Pythonic way to reverse a text string.
- (To iterate through a reversed sequence without creating a separate copy of it, use the lazy sequence iterator `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".
- Since Python string objects are **immutable** (that is, 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 a whole.
- **Immutability of data has several surprising benefits in programming.** For example, string objects can share the underlying bytes instead of storing them redundantly in duplicate. This makes operations such as slice extraction more efficient in time and space.

- (Immutability has one huge general downside, which is why all data in programming is not automatically made immutable outside the beardy school of thought of **pure functional programming**. If some program needs to iterate through some large range of possible values, all data being immutable would require creating a separate object to memory for each value during the iteration. Using mutable data, the same memory bytes can be used to store each value one at a time, and the program does not run out of memory even if it were iterating through trillions of values.)

Lists

- If all names in Python could only ever refer to **scalar values**, it would be practically impossible for some program to handle millions or even billions of items of data that it reads from the outside world, since each data item would have to be talked about using a separate name that is distinct from all other names. Nobody would have the time or patience to type in all those millions of names in the program, and even if they did, there would be no point in doing that anyway.
- Every programming language needs to have some mechanism to store an arbitrary large number of objects under a single name. In the Python language, **lists** are special objects that contain an entire **sequence** of **object references** inside them.
- Python lists are **heterogeneous** so that the same list can simultaneously contain objects of different types. These objects can themselves be lists, resulting in a **nested list** or a **multidimensional list structure**, depending on how you wish to view it.
- Python lists work like **random access arrays** of other programming languages so that an element at any position can be read and written in the same constant time. However, unlike the immutable strings, Python lists are **mutable** in both length and content.
- (The term "random access" would be far more accurately called "arbitrary access" since there is nothing "random" about it in any sense of the word, but the term is far too established in its inaccurate form. I am *literally* foaming in my mouth at this insanity.)
- The internal implementation of Python lists is optimized for appending new elements to the end by maintaining an unused "slack space" as room for additional elements. Once the slack space runs out, the Python virtual machine allocates more room somewhere else in the memory and copies the list data there.
- For strings, trying to access substring out of bounds is not an error, but the operation simply treats the nonexistent part to be the empty string. For example, `"Hello"[2:100]` would produce `"llo"`.
- To find out the length of the given sequence, use the built-in function `len`. This function is able to operate on any sequence given to it.
- The canonical trick to create a separate but content identical copy of the list `a` is to say `a[:]`. This is mostly used in situations where you want to iterate through the elements of

the list while you are modifying it. As this would be logically impossible, you actually iterate through the elements of the original list while modifying its copy, so that the modification operations do not interfere with the iteration through the original elements.

- However, note that when a list is sliced and copied, the references inside the result will still point to the same objects as the references inside the original list. If an object is modified through one list, it has also changed when viewed through the other list.
- 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 answer list to be empty, and then `append` individual solution elements to that list whenever a new one is found.
- Note again that it must be possible for a list to be empty, if no solutions exist. There must exist some way for a function to effectively say that "Your question is perfectly legal and valid, but buddy, there are zero solutions to what you are looking for, so this empty list is what I must return as my answer to you now."

List comprehensions

- **List comprehensions** are a handy way to create lists in a concise way inspired by the mathematical notation used in set theory in math textbooks, by converting the values from an existing iterator or sequence into another sequence, possibly filtering out some elements along the way.
- In most other programming languages, programmers often have to do the work of the list comprehension with explicit **for-loop statements** that iterate through the elements one at a time, choose some of them inside the loop with a conditional statement, and `append` the results into an initially empty result list. List comprehension will achieve all that in one swoop.
- To use a list comprehension, there needs to already exist some list or other sequence to use as **source of elements** that the list comprehension will then **filter** and **modify** in a way that you can freely specify to create a whole new list for the result.
- The syntax of list comprehension consists of square brackets, followed by (1) the expression used to modify 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 integer iterator object `range` is often used as the source sequence inside the list comprehension. 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.
- As seen in the previous example, the if-block is optional if you do not wish to filter out any elements from the source sequence. The result of such list comprehension without an if-block will then necessarily have the exact same length as the original sequence.

- A list comprehension can even contain more than one consecutive for-blocks. The expression will then go through all possible combinations of the individual values produced by the individual for-blocks. If the first for-block produces 8 elements, and the second for-block produces 5 elements, the resulting list contains a total 8 * 5 = 40 elements for all the possible pairs that can be made up from these elements.

Nested comprehensions

- Same as all expressions in Python, list comprehensions can be **nested** to arbitrary depths to create **nested lists**, that is, lists whose elements are themselves lists. For example, the list comprehension `[ [x+a for x in range(3)] for a in range(4) ]` would produce a list whose four individual elements are themselves lists that contain three elements each, generated by the nested list comprehension.
- Note how very different the previous expression is when compared to the deceptively similar expression `[x+a for x in range(3) for a in range(4)]` that produces a list that contains twelve separate integers as its elements, as opposed to four lists that each contain three integers. Meditate on the difference between these two examples until you understand why exactly they are different.
- The second expression also demonstrates the order in which these for-blocks advance through their values. The second for-block goes through all its values before the first for-block advances one notch forward, at which point the second for-block goes back to beginning to go through all of its values once again from the beginning.
- To make this example more intuitive by using an analogy of an ordinary clock, the second for-block corresponds to the minute hand, whereas the first for-block corresponds to the hour hand. (If there were three for-blocks inside the same list comprehension, that third for-block would then correspond to the fastest of these three hands, the hand that ticks through the seconds.)
- The comprehension `[(h, m, s) for h in range(24) for m in range(60) for s in range(60)]` would therefore produce a list with a total of 86400 elements, one for every second that takes place during one entire day, each second represented as a three-element tuple (hour, minute, second) from `(0, 0, 0)` to `(23, 59, 59)`.
- The later loop of a nested list comprehension can also depend on the current value of the earlier loop, so that the comprehension `[x for y in range(2, 6) for x in range(1, y)]` produces the list `[1, 1, 2, 1, 2, 3, 1, 2, 3, 4]`.

For-looping through any given sequence

- 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 whatsoever, be it a list, tuple, string or whatever.
- In the syntax `for elem in items`, the term `items` is the arbitrary sequence to iterate through, and `elem` is a made-up name that refers to the element that is currently being processed inside the loop.
- The same nested body of statements will be executed for every element in the sequence. However, unlike in a list comprehension, this body can be arbitrarily complex and maintain additional **state** in other variables that will then affect the processing of the current element, whereas list comprehension assumes that the processing of the current element can be done without knowing anything about the other elements.
- Whole bunch of common idioms and techniques abound with for-loops, as illustrated in the example functions, and are hopefully applicable to the graded labs that you will be solving on your own.
- One common situation is that the processing of the current element depends on the value of both that current element and the element processed in the previous round. Simply maintain a local name (canonically named `prev`) that remembers the element from the previous round of the loop. At the end of the body of the loop, the current element is assigned to become this previous element for the next round.
- Sometimes processing either the first or the last element of the sequence has to be done in some special way. In computing, this situation is known as "a loop and a half", since we often end up duplicating some of the code of the loop body either before or after the loop. The more of your own code you end up duplicating, the more it should feel like you are not solving your problem the best way.
- Often, giving the variable `prev` some tactically chosen initial value allows the processing of every element to be done in uniform fashion. (To paraphrase the famous Zen riddle, "what was the value of the previous element when you are processing the first element?")
- Python offers plenty of sequence decorators for more advanced processing of the given sequence, such as `enumerate`, `zip`, `reversed` and `sorted`. These should be heartily used when possible, to make your code more concise, readable and Pythonic.
- Especially `enumerate` should be used whenever the processing of the current element depends on both the element value and its position in the sequence. Writing a for-loop of the form `for idx in range(len(items))` and then using `idx` inside the loop body to access the actual element has a non-Pythonic feel akin to C, Java and other such low-level languages where this was the only way to access the elements of a sequence.
- Since the body of the for-loop can contain any statements, other loops can be nested inside the loop body, same way as with nested list comprehensions.

- Among many others, the `itertools` module has a handy function `combinations` as a more succinct alternative for using two nested `for`-loops to iterate all possible ways to choose exactly `k` elements from the sequence, with typically `k = 2` for iterating through all possible pairs of elements.

Iterators

- Real programs have to be able to handle millions, billions or possibly even trillions of items of data. Python 3 unifies this logic for various types of data with the ubiquitous concept of **lazy iterators** that permeate the language in a highly useful form behind the scenes, the same way that everything in Python is an object even though we don't need to talk about this in the language itself.
- An **iterator** is a special kind of 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**. However, since the generation of these objects is **lazy** so that the next object of the sequence is generated only when it is explicitly requested, this process will not run out of memory even for infinitely long sequences.
- (In computer science and programming, being "lazy" is a good thing, doing exactly what is needed and never anything more. In fact, perhaps we ought to call such programs "elegant" or "optimal".)
- Every iterator object remembers its **internal state** 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 of objects produced by an iterator can even be empty. This is an important possibility in all programming that many non-programmers tend to have difficulty in grasping. Unlike in real life where everything is always designed for there being one or more items to process, program logic needs to correctly handle the edge case possibility that there really turns out to be nothing to do.
- **Whenever there is nothing to do, the correct thing is to do nothing.** (Think about this seemingly simple but very important principle, and meditate upon it a few times.)
- As a downside, iterators do not in general support **random access operations** such as **slicing** and **indexing** the elements of the sequence at an arbitrary position or position slice, but the elements must be generated and processed in strict sequential order. 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.
- For the same reason, iterators in general do not know how many more items they will be producing, and therefore cannot respond to the `len` function asking this.

- 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 in general do not support **rewinding** or restarting the sequence, but an entirely new iterator object has to be created.

Defining integer sequences with `range`

- Proper style of writing functions in Python is to have them take iterators as arguments, without having to care of exactly what type iterator the function receives. Such **parameter polymorphism** makes functions more powerful and flexible, instead of having to write separate functions for different types of data.
- Python's built-in function `range` creates and returns an object that produces an integer sequence one number at the time. The arguments for `range`, separated by commas, work exactly the same as for the string slicing operator. (Given just one argument `n`, `range` produces the sequence `0, 1, ..., n - 1`.)
- Additionally, you can give the sequence **step size** as the third argument to `range`. Using a negative step size 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 end value again being exclusive.
- Surprisingly in Python, `range` objects are not iterators, but can always be turned into one by calling the function `iter`.
- Any iterator can be explicitly asked for its next item of the sequence with the built-in function `next`. If the series is complete so that no more values remain, this function raises the `StopIteration` error.
- In practice, `for x in iter` is a handy way to execute a **body** of statements once for each value produced by some iterator `iter`. Inside the body, the name `x` will then contain the current value of the sequence that is being **processed** by the **loop**.
- The common idiom `for x in range(n)` will execute its body exactly `n` times. This is useful for problems where you know exactly how many steps you need to take to reach your intended goal, the same way a human would do if asked to repeat some operation for, say, exactly five times.
- The objects that `range` creates are even smart enough to know to use integer arithmetic to deduce in constant time that `-1 not in range(10**100)`, and that `len(range(10**10)) == 10**10`. In general, answering these questions for arbitrary iterators necessitates 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.
- The built-in function `list` can be used to build 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. (A couple of billion objects would surely suffice, even in our present petabyte age.)

File handling

- Names and objects inside the Python interpreter cease to exist when the Python interpreter terminates. Real programs need to be able to store data in some **persistent** fashion, typically into a file, so that the data outlives the process that created it.
- Following its "batteries included" design philosophy, Python makes reading and writing text files very easy. 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.
- When opening a file, 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** that uses only one byte per characters and punctuation that normally occur in English text, but allows other Unicode characters to be included freely with multibyte sequences where necessary.
- Conveniently for reading text files one line at the time, the file object can work as an iterator that produces the lines one at the time. However, since the line break character is included in the string that contains the current line, we often use the string method `strip` to eliminate the redundant whitespace characters from the edges of the string.
- 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.
- When a file is opened, the file system **locks** it so that other processes in the computer cannot access it. This prevents files from being corrupted by multiple processes trying to read and write them simultaneously. When you are done with the file, you should call the method `close` to give up the lock.
- Note that `close` is a function name inside `file` objects, instead of a Python built-in function in the same vein as the function `open`.
- As a superior alternative these days, Python's compound `with` statement that allows declaring a **context manager**, a local name that exists only for the duration of the nested block of code, is perhaps most often used with file handling. But this construct can be applied in any situation of the form "Acquire a resource, use that resource, then release that resource no matter how things went during the acquisition and use".
- The `with` statement guarantees that the file will be `close`d automatically after this statement no matter which way the execution leaves the body of the `with` statement, even if the nested body of this statement throws some error during its execution.

- (Interested students can check out the article "Improve Your Python: with and Context Managers" by Jeff Knupp that explains what context managers are in general and how you could even write your own context managers using generators and classes that you learn later in this course.)

Tuples

- Immutable lists of fixed size are often more memory-efficient to represent as **tuples**, which are syntactically denoted by separating the elements with commas.
- Most of the time ordinary parentheses are placed around the tuple elements, but in many cases this is optional. Using parentheses is necessary when creating a singleton tuple such as `(42,)`, or when a tuple is passed as an argument to a function, to distinguish between the commas that separate the tuple elements versus the commas that separate the function arguments. It never hurts to use parentheses around a tuple.
- Tuples are **random access sequences the same** way as lists, so all non-mutating operations work for them the same way as they do for lists and other sequences.
- Only the tuple object itself is immutable; the objects that it contains do not magically become immutable due to their containment inside some tuple.
- A handy quirk of the Python language is that multiple assignments can be performed in a logically simultaneous fashion by assigning to a tuple that contains the names to be assigned.
- For example, swapping the values associated with names `x` and `y` can be done in a single swoop with the tuple assignment `x,y = y,x` without having to use a temporary third name to store one of these values.
- Tuples and other sequences can be compared with each other for order. This ordering is done **lexicographically**, that is, in **dictionary order** so that the first position where the two sequences differ determines their ordering, regardless of the remaining elements. For example, `"aardvark" < "zebra"`, and `(9,) > (1, 2, 3, 10**100)`.
- Sometimes you want to perform some operation on sequence elements where the processing of a single element depends not only on its value, but its position in the sequence. In more primitive languages, this would be achieved with a for-loop that counts the positions up to the length of the sequence, and uses the position inside the loop to access the entire element. However, this technique is considered not Pythonic and should be replaced by higher-level constructs.
- Rather than keeping explicitly track of the position in a counter variable in the loop, the function `enumerate` turns a sequence into a sequence of tuples whose first element is the position offset (starting from zero) and the second element is the actual element itself.
- For example, enumerating the list `[42, 'hello', 7.5]` would produce a sequence that consists of tuples `(0, 42)`, `(1, 'hello')` and `(2, 7.5)`.

- The simple loop `for (idx, elem) in enumerate(seq)` gets the rest of the job done. Inside the loop, `idx` contains the position of the current element `elem`.

Sets

- Many programming tasks become much easier when aided with some cleverly chosen **data structure** to store and represent the data, instead of keeping all your data in one unsorted list and then writing functions to plough through this entire list every time that you need to do something on some data element.
- Especially useful are **sets** and **dictionaries** that exist as **first class citizens** inside Python.
- A dynamic set instance can be created either by calling `set()`, making it initially empty, 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` that determine whether the set currently contains the given element `x`.
- Checking whether an unsorted list contains the element `x` has to internally iterate through the entire list and compare these elements one by one. When `coll` is a set, it internally organizes its data in a far more efficient fashion so that these element query operations would still work in a snap of your fingers even if the set contained hundreds of millions of elements.
- The `set` data type has **methods** (that is, functions defined inside the datatype, as opposed to being used from the outside) `add` and `remove` to modify its contents dynamically, plus many others as documented in the Python library reference.
- (The data type `frozenset` is otherwise like `set`, but without the mutation operations, analogous to how the immutable Python tuples relate to the mutable lists.)
- Unlike lists and strings, sets are not sequences, so they cannot be sliced or indexed, since the question "What is the seventh element of this set?" is meaningless to begin with.
- A given set instance can be converted to a list with the function `list`. It is possible to directly iterate through the keys of a set using the `for x in coll` statement. However, this iteration will not generally go through these keys in any kind of meaningful (especially sorted) order, but in a seemingly random fashion in which these elements are internally stored in the underlying **hash table** data structure.

Dictionaries

- The **dictionary** data type `dict` is very much like the type `set`, but 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.
- This simplistic example of a phone book assumes that there do not exist two people with the same name, so that we can use the name as the unique key to identify the person. For

each such key, the dictionary can hold at most one value. Associating a new value to the same key is not an error, but merely erases the previous association for that same key.

- The operations of a dictionary data structure are **efficient on keys** so that given a key, its associated value can be found efficiently. (Same way as in an old-time printed phone book, the phone number for the given name can be found efficiently with **binary search**, but it is not so efficient to find the name if you just know the phone number!)
- 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. For example, counting how many times each word occurs in the given source text such as "*War and Peace*".
- For convenience, Python dictionaries define the square bracket indexing operation as shorthand for accessing individual keys and their associated values, and the for-loops work directly for dictionary keys right away.
- Like all other data types in Python, sets and dictionaries are **heterogeneous**, so that different types of keys can be freely added to the same set or dictionary. 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.
- Dictionaries are an immensely powerful tool that can be used to concisely solve various computational problems, often doing at least half of your work for you, compared to trying to solve that same problem without using any dictionaries.
- Sets and dictionaries allow each key to occur in the set only once. If you need a dictionary where each key can map to multiple values, map those keys to either a list or a tuple of values.
- In Python, every object actually contains the namespace dictionary `__dict__` that stores the names defined inside that object.
- To access the name `foo` inside object `x`, use the syntax `x.foo`. Alternatively, the square bracket indexing can be used to access a name inside an object when the name is given as a string, such as `x["foo"]`. This comes handy when the name is not known at the time the program is being written, but will be decided at runtime.
- Two special Python built-in functions `locals` and `globals` return dictionaries that contain the names of the current local and global namespaces, respectively.

Additional list, set and dictionary goodies

- Python already offers several handy built-in functions for common operations on sequences, such as `min`, `max` and `sum` that do not need to be reinvented all the time.
- Elements can be added to and removed from any position in a list. However, due to the way that the list elements are stored internally in memory, these operations are fastest when done to the end of the list, since the list object internally maintains some slack space in the end to better accommodate these appends.

- (Removing an element from the middle requires Python to move all further elements one step to the left, since no gaps may be left inside the sequence stored in the memory.)
- To check whether a list, set or dictionary contains a given key, just use the operator `in`. Lists also have the method named `find` that returns the position where the element given as argument is located.
- **Set and dictionary comprehensions** are analogous to list comprehensions, with syntax identical to list comprehensions but using curly braces instead of square brackets.
- In fact, Python 3 even allows **iterator comprehensions** to create lazy iterator objects on the fly, typically when an iterated sequence is passed as an argument to a function. These can be a life saver if the generated sequence would be too long to fit all at once into the heap memory of the Python virtual machine, just by changing the list comprehension square brackets to parentheses to turn the sequence into a lazy one.
- Strings and lists often need to be converted to each other. To break down a string into the list of the individual words that it contains, use the string method `split`, whose argument defines the **separator** between the words. By default, this separator is a whitespace character. For example, `"Hello there, world!".split()` would produce the result list `["Hello", "there,", "world!"]`
- To convert a list into a string, use the converse method `join`. Its optional argument is again the separator placed between the words, defaulting to a single whitespace.
- The `collections` library module defines other handy data structures that make many advanced programming tasks simpler, such as **double-ended queues** and **priority queue,** and a `Counter` data structure to simplify common **counting operations** on data. These behave from the outside as if they were sets and dictionaries, so they can be used freely wherever a set or a dictionary is required, but they organize their data internally in a different way to make some other useful operations more efficient than they would be in an ordinary set or dictionary.
- Python functions can be defined to accept arbitrary additional arguments. Special parameters `*args` and `**kwargs` must be placed at the end of the function parameter list. These two automatically collect the additional positional arguments and keyword arguments into a list and a dictionary that are available as local names inside the function.
- (Both these parameters can actually be named something else than their canonical names `args` and `kwargs`, since the preceding asterisk and double asterisk already syntactically denote these special arguments. But consistent **naming conventions** make code easier for other people to understand.)
- Two or more sequences can be iterated through in lockstep with the function `zip` that produces a sequence whose elements are tuples of the original elements.

General repetition

- In general, **you should always let the computer do your repeating for you**. Whenever you feel the urge to repeat yourself in code, even if "just for this one time, what could be the harm, I don't feel like thinking about this issue any deeper than that", that is a surefire symptom that you are doing something wrong.
- It is perfectly fine to copy-paste some useful chunk of code from **Stack Overflow** and similar places, but whenever you copy-paste some chunk of your own code inside the same project, stop what you are doing right there, go back to the proverbial drawing board, and try to think about what you are doing in a higher level of abstraction.
- Some functions need to be able to do something **a number of times that cannot possibly be known at the time the script is written**, and thus this number of repetitions cannot be hardcoded into the function.
- In some problems, this number of times cannot even be known until the function has been executed and we can only find out in retrospect what happened. The only way to proceed is to keep going for however long it takes to reach the desired goal, and keep hoping that you will soon get to your destination.
- Before structured programming in the seventies (and even today in low-level machine code), the control of execution consists of jumping back and forth within the code with the **goto statements** that were famously **considered harmful**.
- For nontrivial programs, using `goto` to control their execution produces unstructured **spaghetti code** whose logic, once grown to be large enough, could not be followed or modified in a controlled fashion. (For unintentionally comical examples of such an approach, see "Basic Computer Games", the classic homebrew computing book from the seventies that your instructor has fond memories of poring through as a kid.)
- Of course, things could always be worse: in the esoteric programming language INTERCAL, originally designed as a joke and parody of other programming languages of its era, has a `comefrom` statement instead of `goto`. (Think about this `for` a `while`.)

Reaching your goals with while-loops

- The idea of every while-loop ever written is to reach some **goal** that is some distance ahead, in most of the interesting problems this distance being unknown. We cannot directly jump to the goal, but can get at least a little bit closer to it by taking some simple steps that we know how to take, instead of trying to swallow the whole problem down in one big gulp.
- The infinite-way decision of how many steps we need to take to get there gets broken down into a series of small two-way decisions "Have we reached the goal yet?" that we can solve just by taking a look at where we are currently at, without needing to know how far ahead in the unknown darkness the goal resides.

- (All computations ultimately boil down to some series of two-way decisions. Those are the only things that a mechanical and mindless computing machine can do on its own. Anything higher than that, we need to tell the computer how to break it down to a series of two-way decisions.)
- The logic of 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 are needed to do so, 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 is `False` to begin with. (Again, when there is nothing to do, it is correct and essential to really do nothing!)
- In this course, **infinite loops** that never reach their goal but run forever are always a logic error. However, some loops work out the prettiest when they are nominally written as infinite loops, but a suitable conditional `break` is used inside the loop body to escape from the loop once you realize that you have reached your goal.
- Any loop in Python can be prematurely terminated with the `break` statement. A similar, somewhat strangely named statement `continue` will skip the rest of the loop body and go directly to the next iteration round, although unlike in the classic board game of Monopoly, we "pass Go in between" to update the loop variable accordingly.
- In fact, would it not be far more truthful if this keyword were named `donotcontinue`?
- Python loops also allow the curious construct of an `else` block after the loop. This 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`.
- Such `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.
- (Implemented in other languages, this type of loop typically maintains some truth valued local variable indicating whether any counterexamples have been found, and the else block is replaced by an `if`-statement testing for that boolean variable.)
- For more observations of how to write loops properly in Python, see the talk "Loop Like a Native" for the Pythonic constructs for various common iterative situations.

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 you either find it somewhere in a list or you have looked at every element.

- This will get slow once the lists start growing large, especially when a large number of searches are performed.
- To make searching more efficient, you can convert the list into a `set` with those same elements that is optimized for this very purpose, making search blazingly fast even for millions of elements. This will pay off if you know that you will perform multiple searches to offset the one-time investment of building the internal data structures of `set`.
- However, 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 is in sorted order, looking at any individual element tells you something not only about that one element, but 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 (in fact, has known it since they were little kids), but they just don't know that they already know it! We can see this from the fact when asked to look up something in a printed dictionary or phone book, nobody starts reading through the entries in order from A to Z until they find the particular entry that they are looking for.
- The formal binary search algorithm, invented back in the smoke-filled fifties by Top Men with crew cuts and horn-rimmed glasses, is rather tricky to get working right in all its edge and corner cases. Instead of the intuition of one finger jumping back and forth in the phonebook zeroing in on the name, we use two fingers that approach each other in a way that the name that we are looking for will always remain between these fingers.
- Binary search repeatedly compares the middle element between the two list indices `start` and `end` to the element being searched. Depending on the result of this comparison, eliminate either the left half or the right half by moving the corresponding index halfway towards the other index
- If the midpoint index `m` between `start` and `end` is computed with the formula `m = (start + end) // 2`, note the asymmetry `start <= m < end` between the sublist endpoints, which makes the correct steps to update these endpoints also asymmetric. The algorithm must be implemented correctly to advance at least one step no matter what, and never potentially jumping over the element being searched.
- Once these two indices `start` and `end` meet somewhere so that `start == end`, that is the only place where the element can be, and either it is in that place, or it is not. (If it is not, then adding that element to that location keeps the entire list sorted.)
- Python standard library module `bisect` contains good implementations of binary search that guarantee finding the leftmost or rightmost occurrence of the given element, or in the

absence of the said element from the list, the place where that element needs to be inserted to maintain the list in sorted order.

- We can use this idea to, for example, quickly find all the words in a sorted wordlist that begin with the given **prefix** that need not be a complete word by itself. For example, in our vocabulary data file `words_alpha.txt`, the list of words that start with the prefix `'jims'` turns out to be `['jimsedge', 'jimson', 'jimsonweed']`.
- To quickly find all the words that end with a given **suffix**, simply reverse all the words and sort the resulting wordlist, then reverse the results of the same search.

Repeated halving to cut a big problem down to size

- Binary search and other algorithms based on the idea of **repeated halving** are tremendously fast even for astronomically large lists, since each element comparison cuts the problem size in half. In linear search on an unsorted array, each element comparison eliminates only that particular element from consideration, thus requiring n comparisons to eliminate n elements.
- Binary search, on the other hand, needs roughly only log n comparisons. Logarithms grow very slowly, so log n is for all practical purposes a constant for any reasonable n.
- To estimate the number of halvings required to cut down a problem into one remaining possibility, for all intents and purposes, 2^{10} equals 10^3, and thus in general, 2^{10n} equals 10^{3n}. Good enough for government work.
- For example, binary search in a billion-element list would require roughly only thirty comparisons to cut it down, considering that 10^9 equals roughly 2^{30}.
- If you somehow had a "God's eye view" to the entire universe, this universe would have to be cut in half roughly only 250 times for one single atom to remain. This shows that within any realistic framework of computation, you simply cannot have a sorted list so big that binary search could take any noticeable amount of time for that list.
- 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 proceed accordingly from there depending on whether the secret number is smaller or greater than your guess.
- (Integers and their arithmetic may seem simple and silly, but isn't it great how integers, no matter which philosophical view you subscribe with respect to the sense that they and other mathematical objects finger-quotes "exist", come in sorted order right out of the box? Integers sure would not be anywhere as useful, were this not the case!)
- To find a **root** of an arbitrary but **continuous** real-valued function $f(x)$ given as a **black box** that allows you to evaluate it at any point x of your choosing, use binary search to repeatedly cut down the interval [*start*, *end*] initially chosen to be wide enough so that

f(*start*) and *f*(*end*) have different signs, which guarantees that the continuous function *f* has at least one root between *start* and *end*.
- Based on the sign of the middle point value *f*((*start* + *end*) / 2), move the endpoint that has that same sign to the middle, thus cutting the remaining interval in half. Keep going until the distance between the start and *end* is small enough for your needs.

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 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.
- (This principle would more accurately be called **zero-one-arbitrary principle**, but the term is already too established in its incorrect form. Why can't people understand that words actually mean things?)
- Even when a programming language allows an unlimited number of something, its interpreter and the finite physical machine that in turn executes this interpreter must necessarily place some upper limit to how many of that something there can be. But the language itself, residing in the timeless Platonic plane of ideas and mathematical abstractions, must not impose any such limitations from our imperfect physical world.
- The behaviour of the nested loops might not initially be that intuitive to grasp. You should especially note that the inner loop is executed in its entirety from the beginning every time the outer loop moves forward one notch.
- To get an intuitive idea of this, again 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".)
- An often occurring need is to iterate through all `n*(n-1)//2` possible pairs of elements of the given `n`-element sequence so that each pair `(x, y)` is processed only once. This is

normally achieved in more primitive languages with the use of two nested for-loops, but a more Pythonic style is to apply the function `itertools.combinations` to produce the lazy sequence of such pairs of elements.

- The technique of `itertools.combinations` also immediately generalizes to iterating through all triples, quadruples and so on of elements, allowing us to, for example, to easily iterate through all 2,598,960 different poker hands of five cards chosen out of the ordinary deck of 52 cards.
- Theoretically, we could similarly iterate through all possible 13-card bridge hands, but since there are 635,013,559,600 of those, the computation might take all day. However, since the combinations are produced in a lazy manner instead of creating them all into an explicit list, the computation will not run out of memory since only one combination at the time is kept in memory.

Generators

- Those of you who are familiar with writing **classes** to define your own high level data types, could write your own iterator types to produce arbitrary sequences simply by defining certain **magic methods** to those classes.
- That feat is not actually too difficult once you have learned how classes and **methods** work, but since lazy iterators thoroughly permeate the spirit of Python 3, the language also offers a simple way to define lazy iterator types as **generators** using the familiar function definition syntax.
- A generator is a function that uses the keyword `yield` instead of `return` to return the result. This fact alone tells the Python compiler that when called, this function actually creates and returns an **iterator** object. The body of the function is not executed at the call.
- When this iterator object is actually asked to produce the next element of the sequence, the function body is executed. When the execution arrives at some `yield` statement, the execution pauses and the yielded result becomes the next object of the sequence that is produced by this generator.
- The local namespace and the execution state 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.
- To end the production of the sequence inside a generator function, you can either explicitly raise the `StopIteration` error, or far simpler, simply return from the generator function in the ordinary fashion.

- As with all iterators, multiple instances of the same generator can coexist in memory, each with its own local namespace and execution state.
- As a handy little piece of **syntactic sugar**, `yield from` can be used to `yield` all items produced by another generator as part of the sequence produced by the current one, instead of having to write the explicit `for`-loop for such common and trivial purposes.

Different levels of programming errors

- **Syntax error**: the code does not conform to the syntax rules of the Python language. These are detected during the compilation and prevent any execution of the script, even from the parts that precede the syntax error.
- **Runtime error**: the program crashes during execution because it tries to do something that is logically impossible, such as divide the string `"Hello"` by the integer 7. (Multiplying a string by an integer is OK, though, as that one is shorthand for repetition.)
- Even though types are not expressed explicitly in Python language, Python execution is strongly typed so that every Python object knows its own type. If some operation is not meaningfully defined for some types of object, the Python engine has no choice but to give up and throw an exception.
- Explicitly typed languages such as Java require the programmer to explicitly give the type of each variable, and that variable may be assigned to point only to objects of that type. This creates more hassles in writing code, but is enough to guarantee that a successfully compiled Java program cannot crash at runtime due to a type error.
- **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. From the pool of infinitely many possible programs, you just happened to pick and write the wrong one.
- Logic errors are by far the most difficult level of these programming errors, in fact the bane of our existence as programmers. Logic errors can only be detected by you, or if you miss them, eventually by your customers and end users. No interpreter or other execution tool can possibly read your mind to determine what your mind intended to do, as opposed to what your stubby fingers actually did type in.
- Above the logic errors there exists even one higher level of **specification error** where you have understood and defined the problem that you wanted to solve incorrectly, thus by definition making it impossible for you or anybody else to write the correct program to solve your problem.
- Sometimes even well understood problems cause us to write specifications that contain a logical contradiction and are thus impossible to solve. These contradictions are revealed when we actually try to write the program and realize their existence, and go back to the proverbial drawing board and update the specification.

- Since the Python language is so concise and performs essentially zero type checking at compile time, **unit testing** shoulders all responsibility in revealing errors. Other languages such as Java are "heavier" than Python, with their design goal of wanting to reveal as many errors as possible at compile time by forcing the programmers to be verbose and explicit about what they want. (Testing is still needed in those languages, since no compile time checking can catch but a small range of all possible errors.)
- 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.

Software testing

- It is important to test your functions with many different **test cases** to root out the errors that are lurking inside them. Each test case consists of the function arguments and the expected result. If the actual result differs from the expectation, the test case has revealed the existence of a **bug** in the code, an error in the logic that causes the computed results to be incorrect.
- Automated tools exist to run through the entire **test suite** with one command, preferably with just one mouse click. (Or maybe even zero clicks.) This allows the entire test suite to be rerun every time the code is somehow modified.
- Standard practice in professional programming when building large systems of millions of lines of code is to rebuild the entire system from the clean slate of nothing but source code every night, and rerun the entire test suite on the current state of the codebase. In the morning, the programmers get the test report and should not write any new code until they have fixed all of the errors revealed to be lurking in the current code.
- When testing your code, you need to change your mindset from constructive to destructive so that you are honestly trying to reveal all the weaknesses and cracks inside the logic of your program, instead of just trying to breeze through the boring testing stage by giving your code simple test cases that you know it will be able to handle correctly so that you can move on to the next interesting task.
- **The purpose of testing is to reveal the existence of errors**, not to give you a false sense of security of your code correctly handling a dozen possible argument values from the infinite number of potential argument values that it could receive.

- The fact that a function correctly handles some small number of example test cases does not establish that that function will correctly handle all possible arguments that somebody might give it.
- **Testing that did not reveal any errors was a failure, not a success!** (Read this sentence many times over and over until you have come to accept its fundamental message.)
- For a multitude of reasons, people tend not to be good at seeing their own mistakes and blind spots. In all serious organizations, testing some piece of code must always be performed by a different person than the person who designed and wrote that code.
- However, **test driven design** where **no code is allowed to be written until you have written a set of test cases for it** is also a perfectly valid software engineering principle. Being required to first write a set of test cases that the existing code does not pass forces the programmer to prove the need for the new code (which is important in the reality of limited time and other resources versus potentially unlimited needs of expansion), and also to think about the various aspects and details of that problem, which then gets him off to good start in typing in the code to solve that problem.

Designing good test cases

- Since you cannot possible try out all possible inputs for any function that is sufficiently complex for somebody in the real world to pay you actual money to write it, you need to maximize your chances to hit some bug with the limited "shots" that you get to take with your test cases, in the spirit of the game of *Battleships*, except that this version of the game is played on an infinite board where the size and shape of the enemy ships and their total numbers are initially unknown.
- Once a program crosses a certain threshold of size and complexity, it will always have one more bug remaining in it, no matter what you do. **Always**. No matter what you do. There is always some unexpected possibility lurking that you did not think of.
- Software engineering techniques used by critical entities such as NASA (who have no option of pressing the reset button or calling in a technician to fix a system that has crashed) can achieve bug rates close to zero, but their techniques are not feasible for software projects run under practical amounts of time and money.
- A simple but good strategy to improve your chances of hitting some bug is to try out all combinations of **edge cases** and **corner cases** of your function inputs, especially the **smallest possible value** and the **largest possible value** of some argument, and all combinations thereof.
- Most importantly, remember that **zero is always a possibility in programming**. When testing functions that take a list or a string as an argument, remember to test that these functions work correctly when given an empty list or an empty string. After that, try out the function with a **singleton** list or string that has exactly one element.

- If you have access to the source code of the system being tested, you can think up new test cases to cause the code to go through different execution paths through it, to improve the **test coverage** metrics of your test suite. Modern **fuzz testing** techniques that feed the system random inputs trying to make it crash can automate this process and reveal bugs overnight while the programmers are sleeping.
- Testing can only reveal the existence of some bug. Testing is followed by **debugging** where the test case that revealed the bug is used as a guide to find the logic error in that function. The bug fix must work for all possible arguments in the equivalence class of the test case that make that particular bug manifest its ugly little head, not merely for that one particular test case.
- The automated test suite for the graded labs of this course 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.
- (Exercise in Python programming: write a function that is given two functions and the test case generator, and finds and returns the first discrepancy, a test case where the two functions produce a different result. Or count all discrepancies. Or return the shortest and simplest discrepancy, instead of just the first one.)
- (That one exercise shows the power of dynamic languages such as Python compared to statically compiled, explicitly typed languages such as Java or C, where this task would be several orders of magnitude more cumbersome.)

JSON

- **JSON** (JavaScript Object Notation) is a widespread modern standard for expressing **arbitrary structured data** as linear Unicode text in an unambiguous and portable fashion. This standard is language independent so that the same JSON files can be read and written by different programs written in different programming languages.
- Unlike many other file formats, JSON is human readable and easy to modify with any text editor, especially when whitespace and indentation are used. The format is otherwise insensitive to whitespace, so that large JSON documents that are intended to be read only by computers can be made smaller by erasing their redundant whitespace.

- 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. In fact, every JSON file is already a legal Python expression that could theoretically be evaluated as Python code as it is out of the box!
- However, for security reasons, the **atomic elements** of the encoded structure may only be strings, truth values, integers and floating point numbers, so that no executable code is allowed, again to prevent 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 deep.
- The `json` library module makes reading and writing JSON files trivial. The function `load` reads the given file and returns the object structure encoded in it. Furthermore, the parser inside the module treats the JSON data as pure text, and is guaranteed never to execute any part of it as code. This makes it safe for the programmer to use it on JSON files regardless of how seedy their sources are, even if they were dug up from the darkest swamps of the Internet.
- (Of course, the data can still be semantically misleading in that behaving as if the data were true results in suboptimal actions. As one wag pointed out, the distinction between data and code is a question of semantics, since all data turns into a decision at some point of execution, since otherwise that data was redundant in the first place.)
- The function `dumps` turns the structure into a properly formatted JSON string that can then be written into a file.

Recursive functions

- A thing is said to be **self-similar** if it contains a strictly smaller version of itself inside it as a proper part. (As an aside, in mathematics an entity is **infinite** if it fully contains something as its proper part that is **isomorphic** to the entire entity.)
- **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 $n! = 1 * 2 * \ldots * (n - 1) * n$ can be converted into an equivalent recursive definition $n! = (n - 1)! * n$.
- 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 $0! = 1$.

- Recursive definitions can be a surprisingly powerful way to solve complex problems. The more complex and powerful the function $F(n)$ is, the harder it also works for you when placed to the right hand side of the definition $F(n) = s(F(n - 1))$ where s is some simple function, since both sides of the equation balance by definition.
- From a technical point of view, there is no difference whatsoever in what happens in the execution of a function call, whether that function is recursive or not.
- Just like with every function call, recursive invocations of the same function each have their own local namespaces. This way the same parameter and local names can exist independently of each other in different levels of recursion.
- In theory, both **recursion and iteration are computationally universal** (both being **syntactic sugar** for the deeper concept of **mathematical induction**), so any computational problem that can be solved one way, could **always** somehow also be solved the other way. However, some types of problems are better suited for recursion than iteration.
- 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, so recursion as a technique does not gain anything in such problems compared to iteration.
- 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.

Practical application of recursion

- To solve a recursively defined problem, first solve a smaller subproblem, a smaller instance of that very same problem, by having the function call itself. Just like with the cliched factorial example, then use the result of that subproblem to solve the original problem in a much easier way than computing that problem from scratch.
- The subproblem is solved by first solving an even smaller subproblem, which is in turn solved by first solving an even smaller subproblem, and so on.
- Every recursive function must have at least one **base case**, when the subproblem is sufficiently simple so that you can solve it on the spot without any more recursive calls. Every recursive function should always begin by testing for the base case.
- A recursive function can have more than one base case. In fact, there could even exist infinitely many base cases, especially if the recursive function takes two or more parameters.
- **Fractals** are pleasant and aesthetic geometric shapes that are self-similar and have a non-integer dimensionality in their infinitely deep nesting. **Subdivision fractals** such as

the **Sierpinski triangle** or **quadric cross** that consist of smaller copies of themselves become natural and quite easy to compute and render with recursion.
- In mathematics, every fractal is infinitely deep, but when rendering its picture onto a finite pixel raster on the screen, the base case is when the shape becomes "small enough", however you choose to define that. (For example, being smaller than the area of a single pixel.)

Downsides of recursion

- Among those who **program by superstition**, recursion tends to have a reputation of being inefficient and bad, existing in some abstract fancy-nancy higher plane that is never to be used in actual programs.
- Few people seem to be able to coherently explain why, especially since the strengths of recursion don't really show up until third year or thereabouts in a decent computer science undergrad program with problems where the recursive ability to **branch** to multiple directions makes it a superior alternative to iteration with loops.
- That said, recursion really does have **three downsides to be understood and avoided**.
- First, every function call, recursive or not, creates a new namespace of local names. A deep recursion simulating a loop (for example, recursing through the list indices) can thus easily cause a **stack overflow**.
- Even with languages such as Python where namespaces are allocated from object stack and therefore in principle limited only by your process memory, recursion depth has been artificially limited to prevent an erroneous **infinite regress** from consuming all of the heap memory.
- The recursion limit can be changed with the function `setrecursionlimit` in the `sys` module, but any reasonable recursive function should be able to complete its execution within that limit. Deep **linear recursions** should be implemented with loops within the same stack frame anyway to begin with,
- Second, even without a stack overflow, the internal bookkeeping of the stack makes the recursive method a constant factor slower than the equivalent iterative solution.
- Third, a recursive method where each recursive call can generate more than one further recursive call can cause an **exponential chain reaction** of function calls. Sometimes this is inherent in problems whose search space truly is exponentially large. However, this is horrendously inefficient when the same subproblems are computed over and over from scratch an exponential number of times.
- Easiest way to fix this is **memoization**: use some 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.

- This is 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.

Applying functions to other functions

- In many ways, Python sits snugly halfway between the **imperative** and **functional** programming paradigms. Especially in Python there is no real distinction between code and data, but functions and code are objects exactly the same way as strings or integers are objects in memory.
- The spirit of functional programming is to pass back and forth functions that are combined to create new functions. Functions can even be passed themselves as arguments for truly mind-blowing results, as with the famous **Y-combinator**. (If you can follow how that one works once you have googled its explanation, you pretty much understand all of functional programming.)
- A more down to earth application is passing **a strategy function** as an argument to another function to be used as part of its execution. Whenever the function needs to make some particular decision internally, it consults the strategy function to ask its opinion, so to speak, and uses that result.
- For example, a function that sorts a list can receive a **key function** that is used to convert mutually incomparable elements to integers or some other comparable values that are then used in element comparisons during the sorting. This way, the sorting algorithm itself needs to be written only once, and it can be used to sort data according to arbitrary different criteria that are not even known when the sorting function itself is being written.
- The key function can be used to implement **sorting by multiple criteria** by having this key function return a tuple of these criteria. This conveniently produces the desired effect, since the Python tuple order comparison is lexicographic.
- A **function decorator** is a function that receives the underlying function as a parameter, and then creates and returns a function object that calls the underlying function, possibly modifying its arguments or result in some way, and possibly performs some other computation about the usage of the underlying function.
- The original underlying function can then be called through the decorator without knowing that there is a decorator that does its work invisibly in between the caller and the original underlying function.
- As an example of the usefulness of function decorators, you could decorate a key function with a decorator that keeps a count of how many times it has been called, to measure the number of operations performed internally by some sorting algorithm.
- As a powerful testament to the dynamic and higher-order nature of the language, Python `functools` module offers the handy function decorator `@lru_cache` that can be

applied to any function (typically recursive ones, but this is not a law of nature or man) to automatically **memoize** it to remember the results that it has previously computed for quick lookup.
- (The abbreviation prefix `lru` stands for "least recently used", so that if the memoization data structure becomes 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 spirit of Marie Kondo.)

Creating small functions on the fly with 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, which are named `x` and `y` inside the function, and returns the sum of their squares. This object is typically passed as an argument to some function that uses it internally to make the decisions that it needs.
- For syntactic reasons, Python lambdas are essentially restricted to being one-liners. If you need a more complicated function, you have to actually define it as a proper function. Other languages with their required delimiters have an edge here over Python, since their lambdas can be arbitrary code blocks.
- The Python style guide advises that you should only use lambda expressions inside some larger statement, and never use lambdas as a substitute for `def` for a function declaration, to make function declaration look like object assignments to names.
- 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 end, you have to first wrap that operator inside a lambda expression, such as `lambda x, y: x + y` for addition. For convenience, the library module `operator` contains named lambda expressions for all built-in operators in the Python language.

Dynamic evaluation of arbitrary text as code

- The dynamic nature of Python is illustrated by the infamous `eval` statement that can evaluate an arbitrary string as a single expression, compiled on the fly before evaluation.

The program could therefore build up a string that contains an expression, and then evaluate that string as code.

- In all **dynamic languages** where executable code is not set in stone at compile time but can be generated at runtime, `eval` can be a very powerful tool to do various cool things, but its inherent danger is when it is used to evaluate strings that originate from the potentially hostile outside world, thus opening the system up for **code injection** attacks. This is an especially serious danger in **web services** written in Python for back end.
- (See the Internet-famous **xkcd** comic of "Bobby Tables" about this phenomenon with the **SQL** database query language. Never forget: "**The client is in the hands of the enemy!**")
- (One pithy comment in a Hacker News thread about this issue put it the following way: *There's no such thing as "data". Every piece of "data" eventually gets interpreted, thus becoming a "command" in some level of abstraction. The core of every security vulnerability is the belief that "this is just a piece of data"*.)
- As a rule, remember the humorous maxim that the word `eval` is a common misspelling of "evil", to be used only when no other safer techniques can possibly do the job. Usually, there are plenty of other techniques to do the same job in perfect safety.
- For some additional safety, `eval` can be given dictionaries as named arguments that are used as local and global namespaces during the evaluation, so that the malicious expression is restricted to this virtual **sandbox** inside the Python environment and does not get to modify the actual local and global namespaces of the Python interpreter.
- However, even this does not prevent all the damage that could potentially be done by direct access to the filesystem, and many other known (and more seriously, yet unknown and perhaps defined in future versions of Python and its standard libraries) forms of mischief and mayhem.
- The interactive Python interpreter is an example of a **REPL**, **"read-eval-print-loop".** This small program, written in Python itself, repeatedly reads in an input string, evaluates it, and prints the result on the console. In fact, once you understand the core Python, you should by now be able to write an entire small Python REPL (including **command history** and other bells and whistles) in Python!
- A more powerful function `exec` allows arbitrary Python code to be executed. Statements given as the string must be separated with either line break characters or semicolons to make the language syntax unambiguous.

Functional programming tools

- As an alternative to loops and conditions, the functional programming tools `map`, `filter` and `reduce` can often be used to express computation logic in a far more clear and succinct (and in fact, inside the Python interpreter, more efficient) fashion.

- List and iterator comprehensions can these days do the work of `map` and `filter`.
- The built-in function `map` takes a one-parameter function `f` and an iterator `it` as parameters, and returns an iterator that produces a sequence of values that you get by applying the function `f` to all elements of the sequence from the iterator `it`.
- For example, `map(lambda x: x*x, [1, 2, 3, 4, 5])` would produce the iterator for the series `1, 4, 9, 16, 25`. As usual, use the function `list` to get all these values in one swoop as a list, instead of stepping through them one at the time.
- The function `filter` cherrypicks those elements from the sequence for which the given function produces a `True` result, discarding those that are considered `False`.
- The function `reduce` (in some other functional programming languages, this function is more vividly called `fold`) turns a sequence back into a single element by repeatedly combining two consecutive elements into one element, using the two-parameter function given to it as the first parameter.
- Iterators can be a surprisingly powerful model of computation. Using the functions defined in the `itertools` 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 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 library module `functools` has the function `partial` that can be used to **curry** some function that takes *n* parameters into another function that takes *n* - *k* parameters, by fixing the values of the *k* missing parameters.

import numpy as np

- The dynamic flexibility of Python data structures such as heterogeneous lists and dictionaries makes for fun and concise programming, but can be inefficient in both time and memory when dealing with large datasets, especially in our current **big data** era.
- To represent a heterogeneous list in computer memory using only simple little bytes (the only thing that really exists inside computer memory), plenty of additional information needs to be stored for each element.
- In science and engineering, all this dynamic flexibility is often not needed, since all data is known to be some simple uniform types such as integers or decimal numbers. Furthermore, these numbers are often known to be inherently restricted to some limited range of possible values (for example, in the population data of the cities of the world, no city has more than about thirty million people), making the internal unlimited integer representation of Python massive overkill for such data.

- **numpy** is an important Python extension that offers a data structure `ndarray` (short for "n-dimensional array") to represent large 1-D **vectors**, 2-D **matrices** and even higher-dimensional **tensors** of **homogeneous** elements.
- Since all elements inside a numpy array are of the exact same numerical type, they can be stored in memory consecutively without any extra information in between, and processed directly in efficient machine code functions that can even be **parallelized** for even more speed with today's **multicore** processors.
- Inside the `ndarray`, integer elements now have a fixed maximum value they can possess, instead of being able to contain arbitrarily large numbers such as `1234 ** 5678` the same way as the Python built in integer type can easily do.
- If some computation produces a larger value than would fit into a fixed number of bytes allocated to it, the result **overflows** and is truncated to fit in those bytes in a cruel Procrustean fashion, making the result essentially useless nonsense.
- Each `ndarray` knows its dimensions and element data type, and uses efficient and low level code written in C and assembler to perform various mathematical operations orders of magnitude faster than they would be for heterogeneous Python lists where the Python virtual machine has to check the type of each element to choose the appropriate Python function to execute, and the same thing happens inside the statements of that function.
- Integer elements can be defined to be of type `intX` (**signed**) or `uintX` (**unsigned**) where `X` stands for the number of bits used, by necessity a multiple of eight. 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 reach for one more negative value than positive values, due to zero being the special "odd man out" in the middle of the range.)
- All these computations are done by the processor machine code instructions that the numpy module operates on directly. As it is with many other things in life, you don't need to know their exact details to be able to use those things. However, you do need to be aware of the external consequences that those details impose on you!
- (More generally, in many places in life, you get to pretty freely choose what the rules are. However, you never get to choose the **consequences** of your chosen rules, since those will not be determined by your wishes and intentions, but by the external reality inside which your rules continue to operate even after you "have left the room.")
- The functions defined in numpy modules operate **element-wise** on the entire `ndarray` at once. You should use these rather than Python built-in functions, even when Python already has that function such as `sum` or `max`.

- Note that the multiplication operator * between two numpy arrays performs element-wise multiplication. If you want to perform proper **matrix multiplication** the way it is defined in linear algebra textbooks, use the function `np.dot` that computes the **dot product** of the individual rows and columns.
- 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. For example, adding a two-dimensional matrix of shape `(10, 20)` and a one-dimensional vector of shape `(20,)` broadcasts the second array into the same `(10, 20)` shape as the first array by duplicating that row vector ten times, so that the matrix addition can be performed elementwise.
- Of course, no actual duplication of data in memory occurs during such broadcast, since the numpy engine is smart enough to access the data from the correct memory location.
- As a special case of broadcasting, any scalar value can be thought to be a 1-element numpy vector that can be broadcast into any number of dimensions. For this reason, we only need one function `np.ones` to create an array whose all elements equal 1, since we can always multiply that array by a suitable scalar value `c` that gets automatically broadcast to the same shape, resulting in an array whose elements are all equal to `c`.
- Matrix multiplication in the sense of linear algebra using the dot product of individual vectors is another special case of broadcasting.
- The `ndarray` data type is an iterable and looks and feels like a Python object, with all the usual operators such as list slicing working the way a reasonable person would expect them to. However, note that slicing produces another **view** to the same underlying data, instead of creating a new `ndarray` and copying the data there. This is the opposite of how Python list slicing operation works.
- Many 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 and **matplotlib** for rendering pretty graphs based on numeric data, on top of which many other extensions (such as **pandas** for data analysis) have been written.
- **Scikits** are third party modules built on top of scipy, specific to some particular field of science. These days, the biggest buzz is surely about scikit.learn that offers a host of **machine learning algorithms** that can find and extract the hidden underlying regularities that lie behind the given dataset of observations, to allow extrapolation from that regularity to unseen values from the problem domain.
- All these fancy machine learning algorithms are available for laymen with only a couple of simple lines of Python code to execute the appropriate machine learning function on the given numpy dataset.

Manipulating images and sounds with numpy

- A **raster image** is a three-dimensional **cuboid** of numbers, with two dimensions for the pixel and the third dimension encoding the colour as sum of (usually three) **colour components**, most important of which are **RGB** (red, green and blue) and HSB (hue, saturation and brightness). RGB is used directly in the computer 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 one-byte **unsigned integers** from 0 to 255. With colour transformations, proper care should be taken to ensure that the computed result values stay within these limits.
- In principle, any image operation could be written as a Python function that operates on these numbers, and this way theoretically build up all the capabilities 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 started, a whole bunch of **filters** and other common image operations have already been implemented in the `scipy.ndimage` module.
- **Image convolution** with the given **kernel matrix** is a powerful operation that can do a multitude of cool things, and many other 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.
- An uncompressed audio recording given as a **wav** file is a one-dimensional vector of intensity numbers for each sound channel, one channel for mono and two channels for stereo sound.
- (The more famous **mp3** format compresses this raw data in a clever fashion that slightly corrupts the sound but in a way that the human ear cannot detect. In graphics, the **jpeg** image encoding scheme works on the same principle. Increasing the compression rate makes the resulting file correspondingly smaller, but introduces disturbing artifacts that the human eye and ear can notice.)
- The intensity of the sound has to vibrate and fluctuate for the human ear to detect a sound. Simple **pure notes** are produced with **sine waves** of the desired frequency. Other waveform shapes produce different sounds that consist of multiple frequencies and therefore sound more natural and thus far more pleasant to the ear.
- (All human senses in general dislike signals that are too pure and monotonous to have emanated from natural sources. On the other hand, there needs to be some underlying sense of order for the signal not to degenerate into **noise** that is equally unpleasant for humans. **Aesthetics** is the art of striking the perfect balance between these two.)

- (As is known in mathematics, arbitrary curves can be broken down into sums of sine waves of different frequencies and intensities with **Fourier analysis**.)
- The intensity of any sound given as a vector of numbers can be modified by multiplying the numerical values (again being careful to avoid overflow), and adding and averaging simple sounds produces a sound that contains them both in desired portions.
- Completely random sound intensities produce **noise** whose type depends on the properties of the random distribution used to produce these random intensities. Different types of noise with different properties can be produced with statistical and mathematical methods.

Thinking in higher level data types and their operations

- Python mixes imperative, functional and object oriented programming paradigms in a concise and relaxed fashion that allows programmers not only to freely choose the extent they wish to follow these paradigms, but mix and match these paradigms within the same program depending on what is most convenient for their needs at the moment.
- So far, we have expressed all our concepts using the data types that already exist inside the language or its standard library modules. For example, we decided earlier that an ordinary **playing card** is represented using a tuple that must contain exactly two strings that are the rank and suit descriptions, such as `('deuce', 'diamonds')`.
- This decision could have been made differently in many other ways, depending on our needs on the operations and behaviour of these playing cards. But we had to choose one of these possible representations and then stick with it consistently.
- The rest of the code now has to know that the implementation of the concept of a playing card is specifically this one to be able to handle the playing cards correctly. Once a critical mass of people has written code that depends on the assumption that playing cards are represented specifically this way, we can no longer change the internal representation of these playing cards even if we later think up some representation more suitable for the needs that we did not realize we have during the original design.
- There is nothing inherently wrong with such a low-level approach to data, it just feels like programming way back in the dinosaur era of the sixties and seventies. The modern **object oriented** paradigm allows programmers to define brand new data types of their own that can be used to create new **instances** as objects in the program, so that these objects are not fundamentally any different from other objects such as lists, strings or `Fraction` objects from types defined in Python language and its standard library modules.
- Instead of thinking at the level of "To analyze a hand of cards, my code operates on the list whose elements are tuples that contain two strings for rank and suit", you get to think

"To analyze a hand of cards, my function operates on the list whose elements are `Card` objects".

- In fact, once you define the concept of "hand of cards" as a separate data type `Hand`, you get to think at an even higher level "My function operates on a `Hand` of `Card` objects", without knowing or caring how those cards are internally represented inside that `Hand` object!
- In good **object oriented design**, the **nouns** of the problem domain become classes in the program, and **verbs** become methods in those classes. Especially when the programs grow larger, this approach makes the program design more **stable and robust** compared to a design where the classes and functions stem from the internal organization of the program functionality, instead of the structure of the problem domain.
- **The larger and more complicated a system is, the higher level concepts we ought to be thinking about it!** Within the confines of our small human brains and the slow conscious mind, a large system cannot be designed and understood trying to think about it as the sum of its nitty gritty low level details and operations.
- This is pretty much the same principle of why we don't write our programs directly in binary machine code, the true and honest underlying reality of all computation, but prefer higher level programming languages to express our ideas in terms of higher level concepts.
- (The obligatory quote in this context is always *"Civilization advances by extending the number of important operations which we can perform without thinking of them"*, as noted by Alfred North Whitehead back in 1911.)
- The ultimate ideal that the object oriented design and programming strives towards is that **no problem should ever have to be solved more than once**, after which that problem should never need to be solved again by anybody, unless they perhaps feel like getting some frisk programming exercise, or if they believe that they could come up with a more efficient solution to that particular problem.
- Useful and powerful concepts are turned into modules and released into the world so that other people can also use them. Really useful concepts will eventually become part of the language standard library, and the most useful ones such as "dictionary" or "set" eventually get absorbed into the very language itself.
- For example, try to imagine Python as an otherwise identical language, but without the `dict` data type in the standard library. How much more difficult would our various examples and lab problems have been to implement without this immensely powerful concept of the dictionary at our fingertips to do half our work for us?
- It has been said that if you are unable to solve some problem by programming, that merely proves that the concepts available in your language are insufficiently advanced. Any problem becomes trivial to solve once you have sufficiently high level concepts at

your fingertips. (The real problem is for somebody to first invent and implement those concepts in a useful way!)

Defining your own classes with `class` statements

- To define your own data type as a **class**, use the keyword `class`. Executing this nested statement creates and defines a **class object**, from which an arbitrary number of **objects** can later be **constructed**.
- Classes are objects in Python like everything else. What makes classes special is that some other object can be an **instance** of that object, such as `"hello"` being an instance of `str`.
- 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.
- 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 is always a reference to the object for which the method is being called. This parameter can be named anything, but is canonically named `self` so that other people reading your code can tell at a glance what is what.
- A class can also have **class attributes** that are defined for the class as a whole, instead of being separate attributes inside individual objects. These are accessed through the class object just like all other object names, of course. Class attributes represent data that is the same for everybody throughout the class, instead of being stored separately for each individual object.
- Class attributes are usually "plain old data", but like anything else in Python, they can be any object, including functions, even entire **nested classes**.
- For example, inside a hypothetical `BankAccount` class, `balance` would be an instance attribute so that every instance could have its own separate `balance`, whereas the total `count` of how many bank account instances exist would be a class attribute.
- Unlike most other object oriented programming languages, 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 Python convention of starting an attribute name with **a single or double underscore** informs the outside users that that name is an implementation detail that might work today when accessed from the outside, but that fact might change at any time in the future versions of this class.
- To keep your own code **future compatible**, it should never assume anything, let alone rely on such private parts of the classes that it uses, but **depend only on the public interface** that is promised not to change in backwards incompatible ways.

- (Of course, it is up to the class designers and maintainers to keep their end of this promise. For any serious and well-established class libraries that have thousands or even millions of users around the world, in practice they always will. The alternative is simply too revolting to realistically worry about, akin to the owner of some successful factory suddenly going cuckoo bananas and intentionally setting the whole thing on fire.)

Making your objects first class citizens of Python

- To create a new instance of the class `Foo`, simply 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.
- To call a method that exists inside an object whose `type` is the class where that method was defined, use the syntax `object.method(arguments)`, that should already be familiar to us in data types that we have imported from various modules. The Python virtual machine finds the method in the class that the object was constructed from, and executes it using the object as the additional argument `self`.
- The **parametric polymorphism** of the important built-in functions such as `str`, so that these functions smoothly work even for objects of types that will be written decades after these functions were written, is made possible in Python by having these functions call certain special methods for the object given to them as argument. This makes the Python core language **future compatible** with any classes and objects that anybody may happen to create in the future.
- For example, the Python function `str(x)` internally calls the method `x.__str__()` and returns whatever that method returned. (In your own classes, `__str__` is a good method to define right away to make testing and debugging printouts more readable.)
- Another method `__repr__` is supposed to return a representation that satisfies the equality `x == eval(repr(x))`. This is not possible even in principle for some data types. To denote such impossibility, the returned representation can be enclosed in **angle brackets** so guarantee that any attempt to `eval` it will crash with a syntax error.
- Other special **dunder methods** (complete list, also known as "magic methods") can be used to define the behaviour of Python's arithmetic operators, comparisons, indexing, ...
- Whenever the Python compiler sees the code using some built-in operator of the language, such as `a + b`, it just silently turns it into the expression `a.__add__(b)`. This is the little wizard man hiding behind the great curtain who makes all this wonderful magic just work.
- Syntax rules in all programming languages are basically just made up. Of course, since the idea is to get other people to also use your language, your syntactic constructs must

be familiar to other programmers and people who want to learn to program, for example the **infix notation** of arithmetic operators such as `+` and `*`. It is the job of the compiler to parse the syntax and turn it into something that the machine can actually execute.

- When defining their own data types, it is the programmer's responsibility to ensure that `__add__` and similar methods do the right thing as is appropriate for that particular data type. Consequently, if some operator does not even make any sense for a particular data type, that dunder method should be left unimplemented.
- 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 if `a < b` and `b < c`, then also `a < c`.
- As proven by the work of **Alan Turing** way back in the thirties, way before even before the first room-sized mechanical computer was built, all semantic properties of programs expressed in any language capable of universal computation are algorithmically undecidable. No compiler can therefore enforce the contract and reject a function that violates it.
- However, if these implied contracts are violated, many 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, which would be a pretty bad programming practice anyway outside some rather special narrow domains.

Patching new and old properties

- Since everything in Python is (still) an object, classes and the methods defined inside them are still objects, and therefore everything that has been taught earlier about Python objects still stands as firm as ever.
- If only some particular object needs to do something special for which we do not wish to create a whole new subclass, **monkey patching** simply means reassigning the name of some method to refer to some other object than which that name was originally bound in the class object.
- Alternatively, one 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 an 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. ("Which one? Only her hairdresser knows for sure.")
- 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`. Were these defined as ordinary attributes instead of managed properties, nothing would stop anybody from assigning `bob.height = -42`. (Even worse, since any name can always refer to any object whatsoever regardless of our intentions about types, assign `bob.height = 'Hello, world!'`)
- The **property getter function** can be used to 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 the concrete attributes stored inside the object.
- Virtual properties can be defined upon other virtual properties. However, make sure that your program does not accidentally get stuck in an infinite recursion where these virtual properties are circularly defined based on each other, instead of these definitions leading down to something concrete, in effect thus becoming the famous infinite stack of "turtles all the way down".

Inheritance relationships between classes

- A significant chunk of the power of object oriented programming comes from the possibility of creating **inheritance** relationships between classes to enforce certain consistency within all objects constructed from the same class hierarchy.
- Statically typed languages such as Java allow the compiler to enforce this consistency to catch various logic errors at compile time, whereas Python has the loose spirit of "anything goes" in what the **subclasses** can do in relation to their **superclasses**.
- To express that your class is a subclass of another existing 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*.
- For example, it would be perfectly reasonable for `Car` to be a subclass of `Vehicle`, but it would be nonsense for `Car` to be a subclass of `Engine`. Every car **has an engine**, but the **car itself is not an engine** from the point of view of its users, nor is it an engine or a car.
- No programming language or its compiler can possibly enforce any kind of **semantic consistency** within the program. The compiler and the virtual machine can only ever

enforce the **syntax** and **typing** rules, but that's as high as they can go. (As the famous pithy saying in programming puts it, you can always write nonsense in any language.)

- 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 can define only the functionality that is new in that subclass, and **inherit** everything else from its superclass.
- In subclasses, you can **override** method names to provide a different implementation for some method. This way instances of different subclasses can do the same thing in different ways, depending on their particular types.
- To prevent the programmer from accidentally overriding some methods that the subclasses ought not to override because the rest of the class assumes that those particular methods perform some exact particular thing, Python silently performs **name mangling** on all the names inside class that begin with a double underscore, so that these mangled names are guaranteed to be distinct from those same names defined in any of its subclasses.
- To call the superclass version of some method from a subclass method, you can prefix the call with the call `super()`. This is perhaps most commonly seen inside the `__init__` method. For that and many other methods, sometimes the existing superclass method implementation does some good stuff that we wish to also be available as a useful part of the subclass method implementation, to avoid falling into the cardinal sin of repeating that exact same code in the subclass method.
- **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 chicken makes, but we cannot similarly describe what kind of sound an animal makes, without knowing what specific subtype of animal that 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 body, typically just `pass`.

Multiple inheritance

- Python also supports **multiple inheritance**, so that some subclasses can have more than one **immediate superclass**. In general, multiple inheritance opens up a whole can of complications that do not arise when only the normal single inheritance is used, so we need to be a bit more careful here.

- The **method resolution order (MRO)** algorithm built in the Python virtual machine determines which method gets executed in a situation where multiple superclasses define the same method name, and usually does the right thing as the programmer intended.
- Multiple inheritance is perhaps most commonly seen in the wild 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 these classes might otherwise 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.
- That, by the way, is the grand theme that permeates throughout the entire CCPS 209: **DO NOT REPEAT YOURSELF.** See you there.