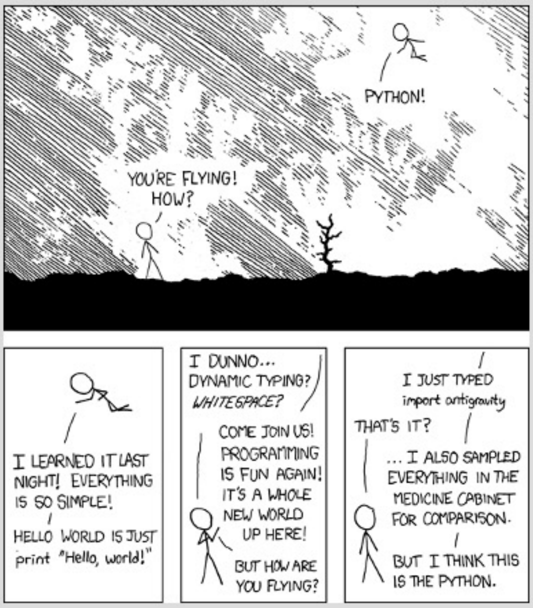
PYTHON PRIMER



AIT-BUDAPEST

András Aszódi

These slides are intended to introduce basic Python features that we will need during the “Biocybernetics” lectures.

Python's origins

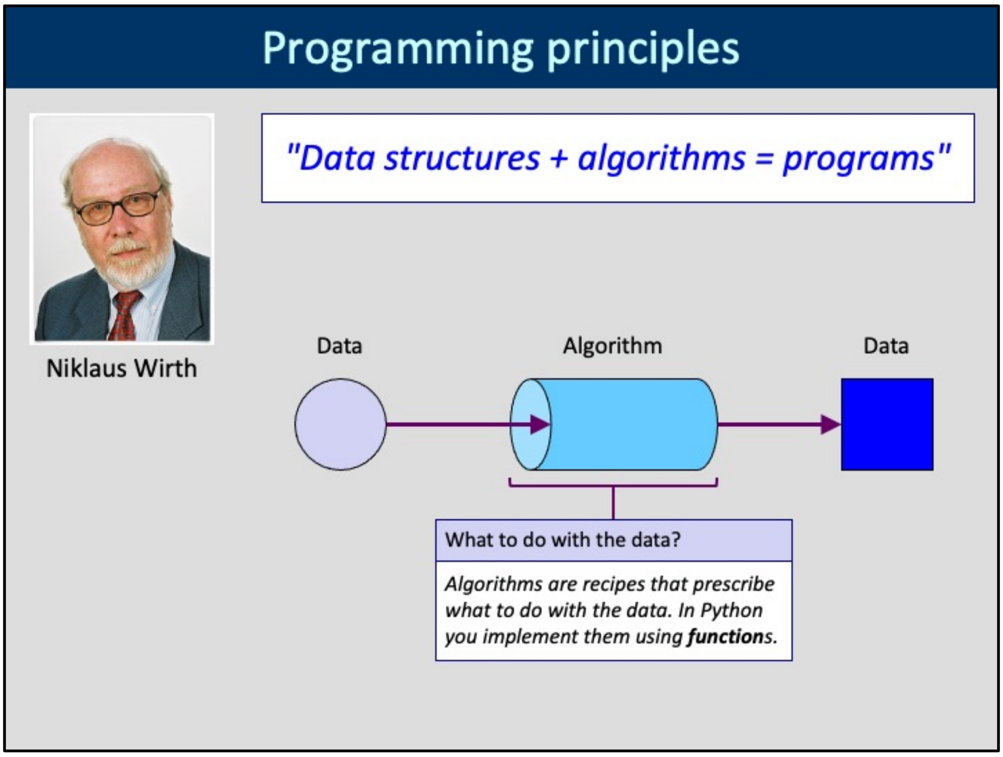
No, Python has nothing to do with snakes. The language is named in honour of the immortal Monty Python group.

reference

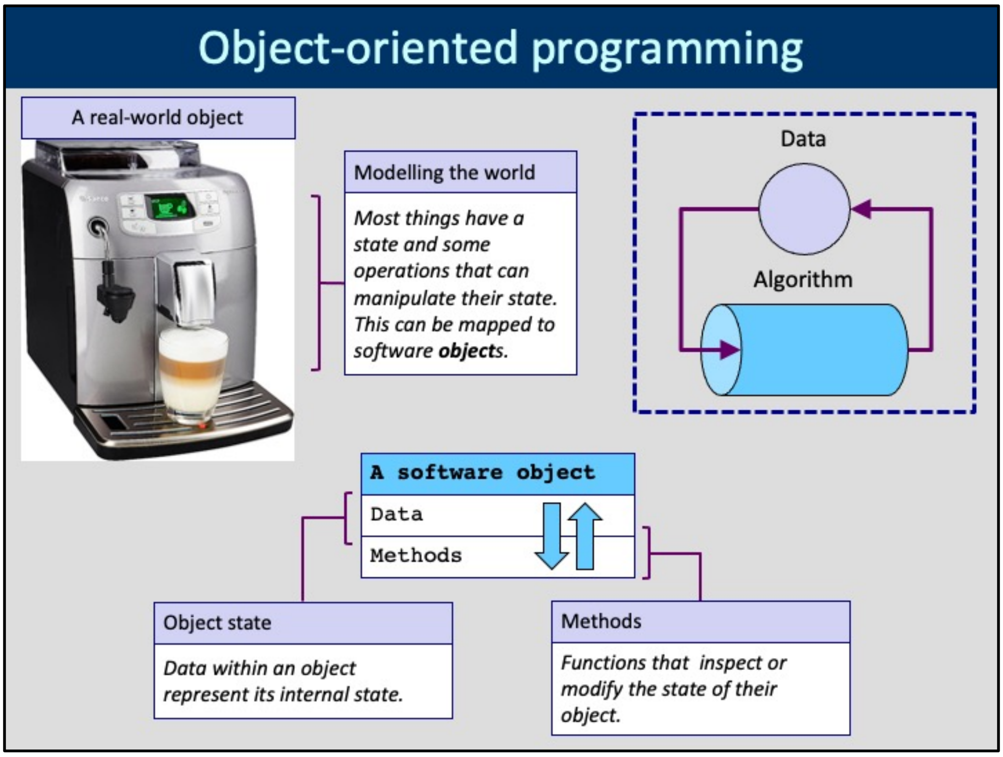
The Zen of Python

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one-- and preferably only one --obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea -- let's do more of those!

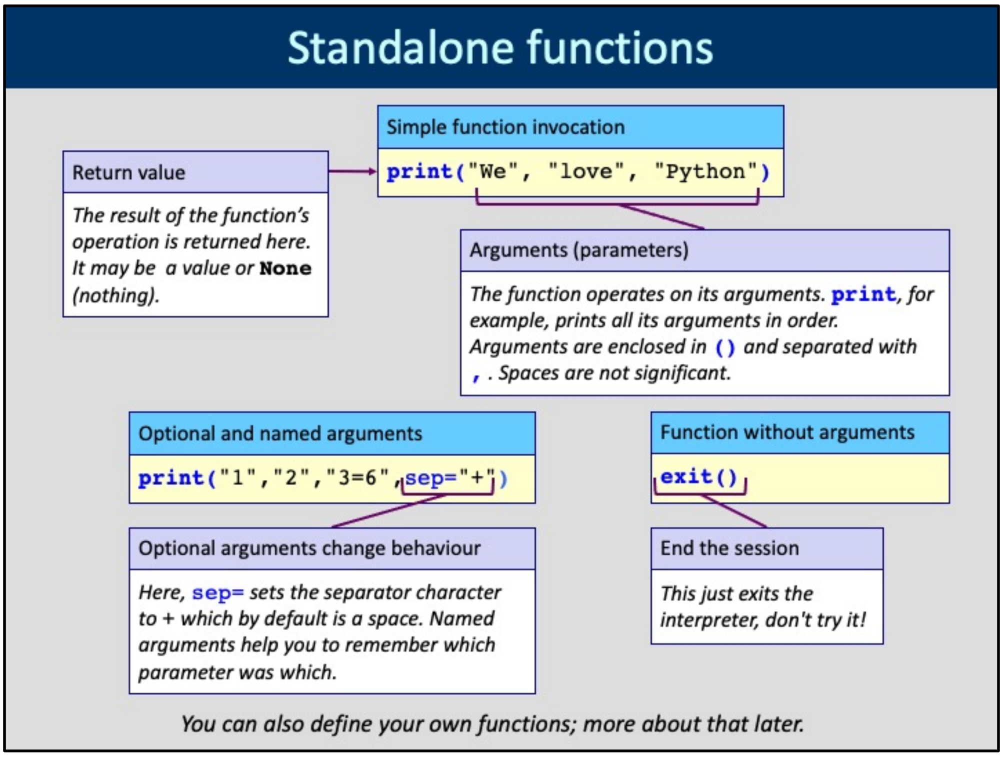
This is the summary of the "Python philosophy", the design principles of the language. Some of them are universal in the sense that they should apply to any human creative effort.



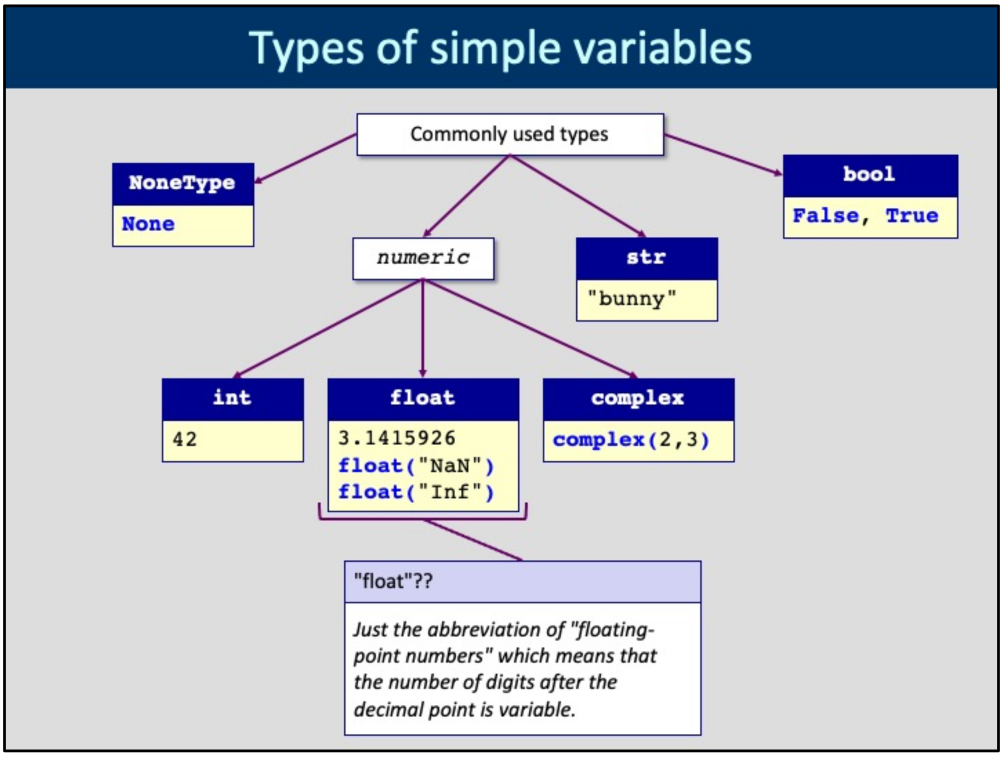
Programming is like cooking. In the kitchen the recipe describes how to convert the ingredients (vegetables, meat etc.) into a delicious dish. In computers, algorithms describe how to convert the input data into a desired result. In Python, the "recipe" (i.e. the algorithm) is implemented as a function, software constructs that take pieces of data, do something to them and then return results.



Python is an "object-oriented" programming language: this means that data and the algorithms operating on the data are conceptually "bundled together" in "software objects".

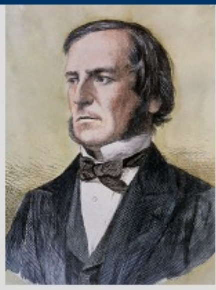


We pass data into a function via its parameters. Optional parameters have default values, which you may override if you wish. Some functions take no parameters at all, they are invoked with an empty argument list ().
Functions can also return a value, or maybe no value at all which is called "None" in Python.



This slide shows only the most often used Python types. There are some other special predefined types which you can look up in the documentation.
The Python data structures (lists, tuples, dictionaries etc.) are discussed in a separate training unit.

Boolean operations



George Boole (1815-1864)

NOT

x	not x
False	True
True	False

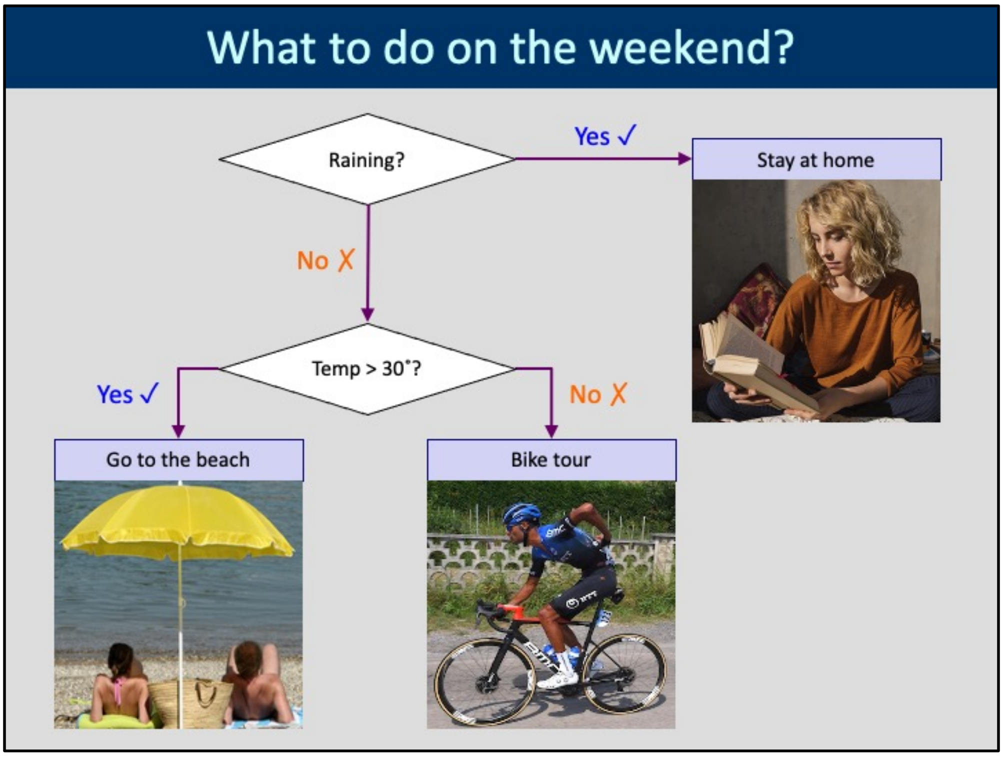
AND

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

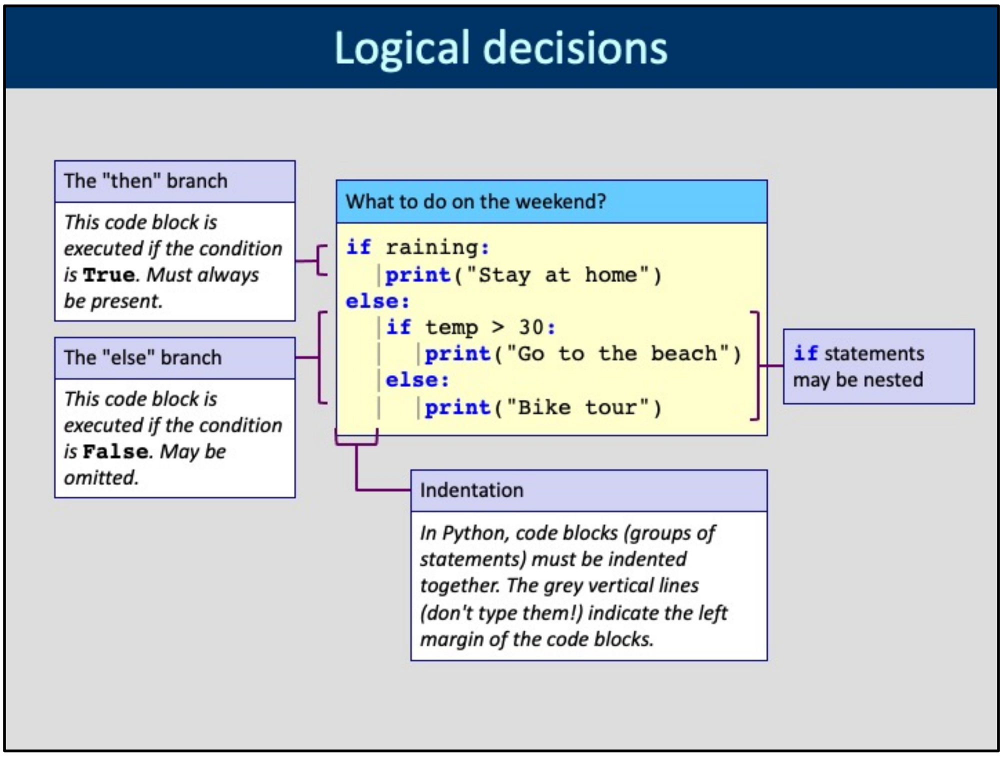
OR

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

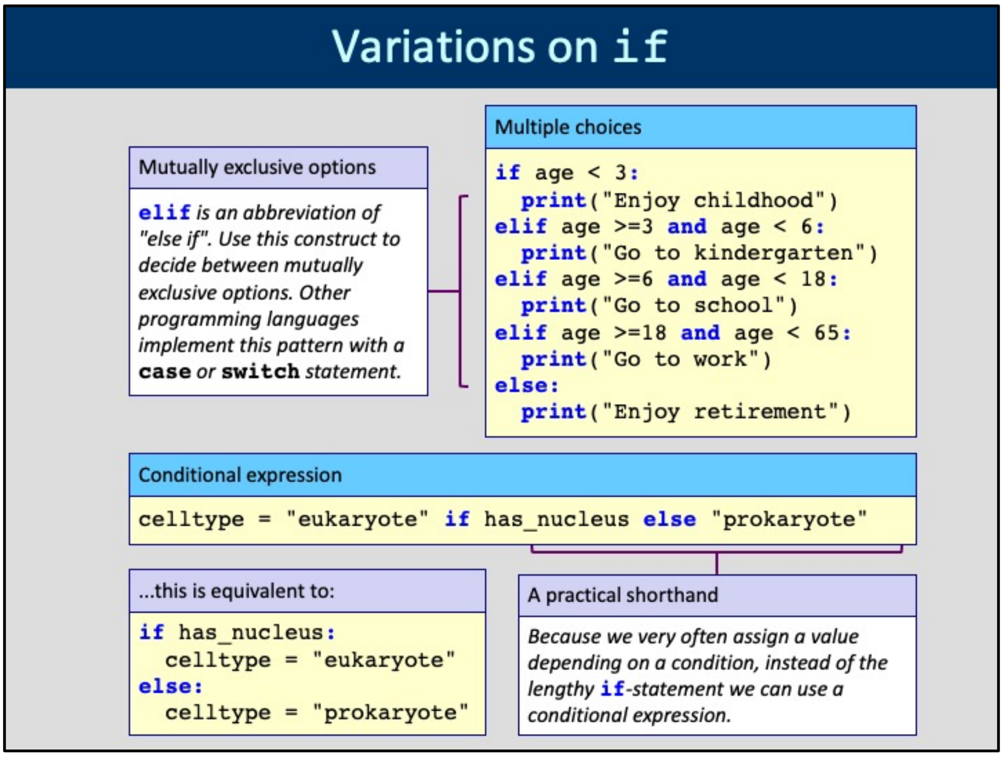
George Boole devised the algebra of simple two-valued logic named after him. Statements can have only "True" and "False" values and these can be combined using the 3 standard operators NOT, AND, OR as shown on the slide.



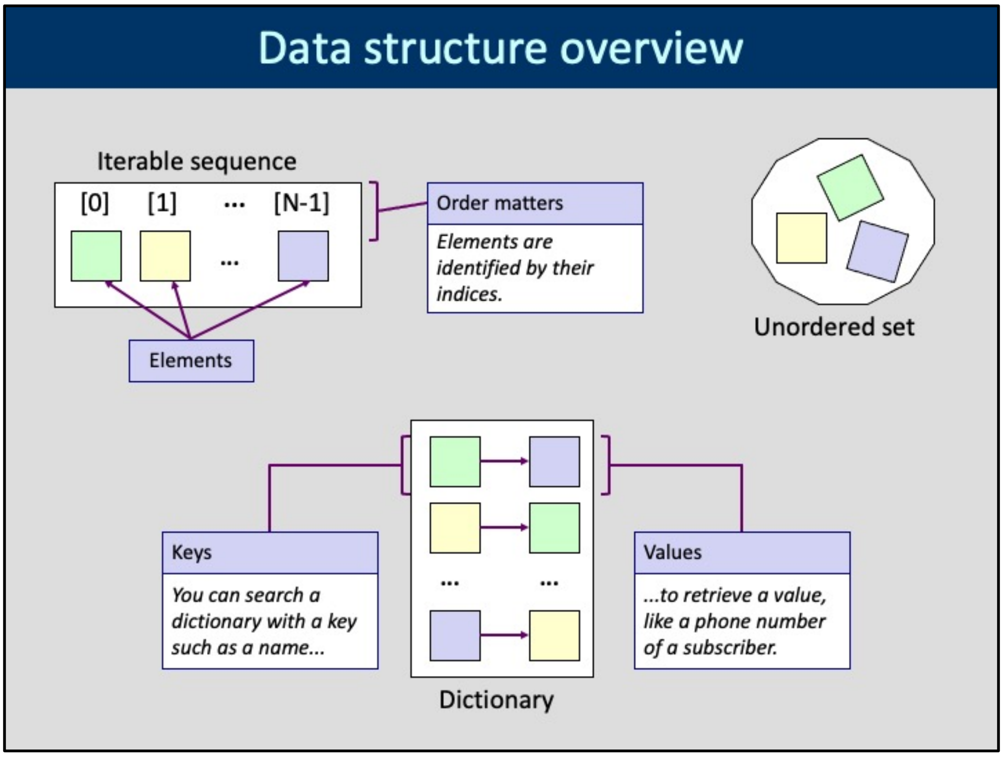
In practically every program we need to take decisions and execute instructions depending on logical conditions. Flowcharts, such as the one shown on the slide, visualise the logical flow of the algorithm.



The "if" statement, which in one form or another is part of any programming language, tells Python to execute different parts of code depending on whether a logical condition is True or False.
The branches of the "if" statement are so-called code blocks that group statements together. Code blocks must be indented by indentation in Python (this is a general feature of the language). Modern IDEs help you getting the indentation right.



Sometimes you need to decide between several, mutually exclusive options. Python offers the if... elif ... elif ... else construct for this purpose.
Another variation is the `x = y if cond else z` construct which is a syntactic shortcut to assign two different values to a variable based on a logical condition. The right-hand side of the assignment `y if cond else z` is called a "conditional expression".

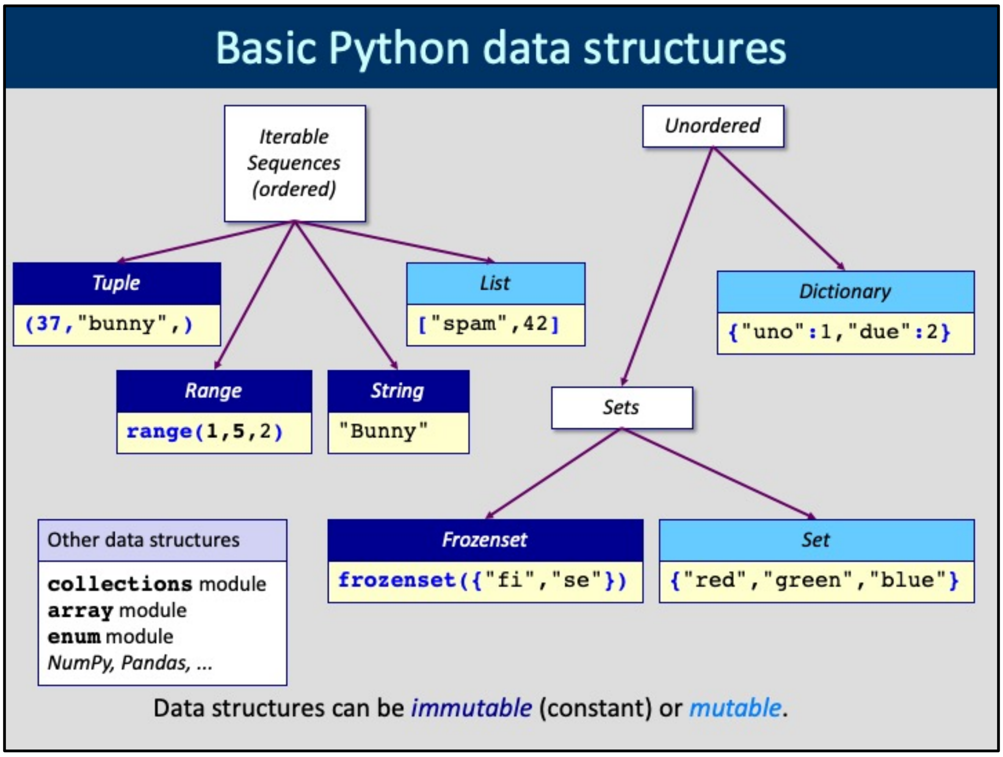


We have learnt how to store single data points in simple variables. Data structures are software concepts for representing more than one data item that logically belong together.

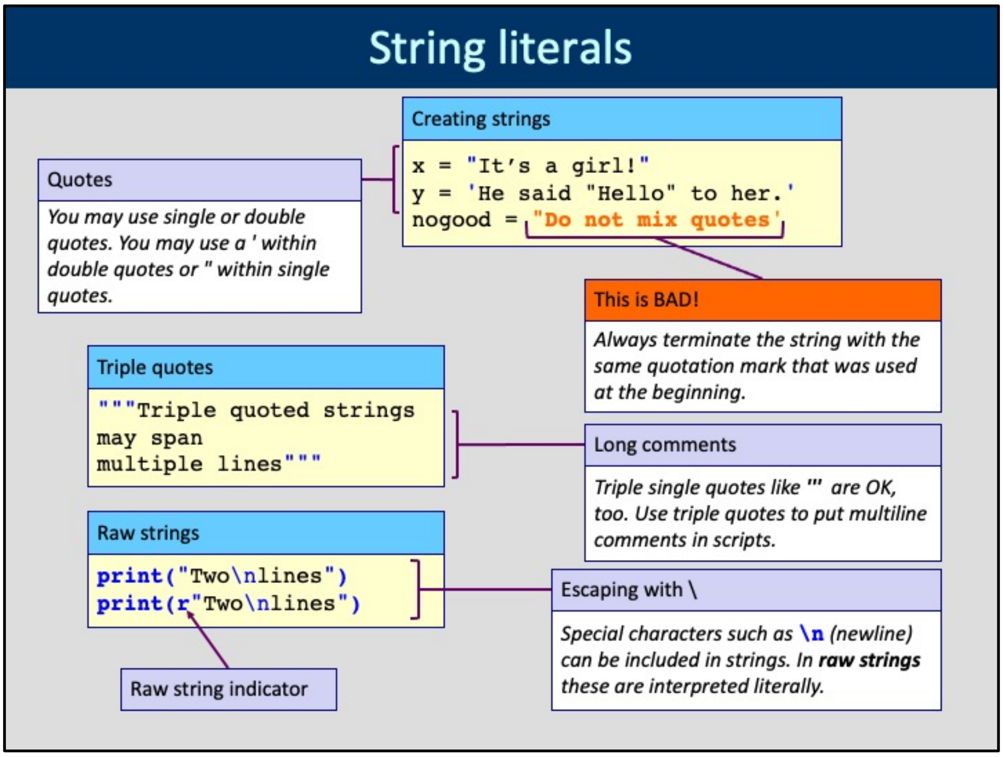
This somewhat abstract overview illustrates the most important data structure types:

1. Iterable sequences are made of a linear list of data items where the order of the items matter. You look up individual data items through an index, which is an integer >= 0.
2. Dictionaries work like a phone book. If you know the name of a subscriber (the "key"), then you can look up his/her number (the "value") easily. The order of the key-value pairs does not matter.
3. Unordered sets just store a bunch of data items. Only the fact that they belong to the same group matters.

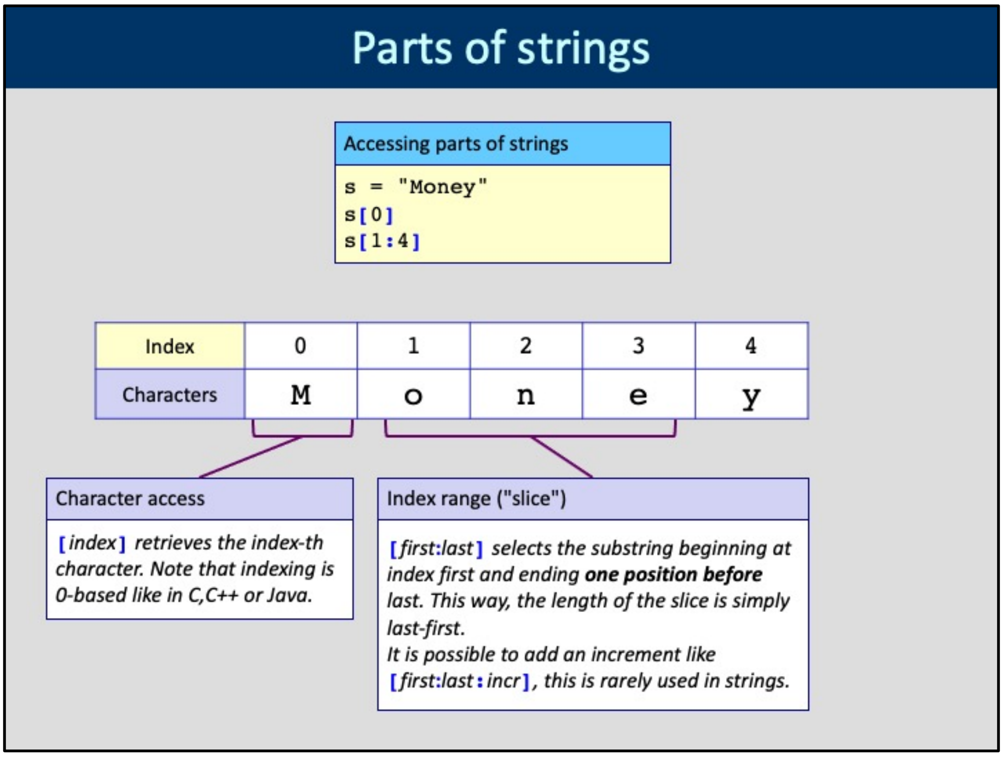
Most Python data structures are heterogeneous, i.e. the types of the individual items within a data structure may be different. This is in contrast to e.g. R where all vector elements must have the same type. There is actually a Python datatype called `array` that stores numeric values which all must be of the same type but this is a special case.



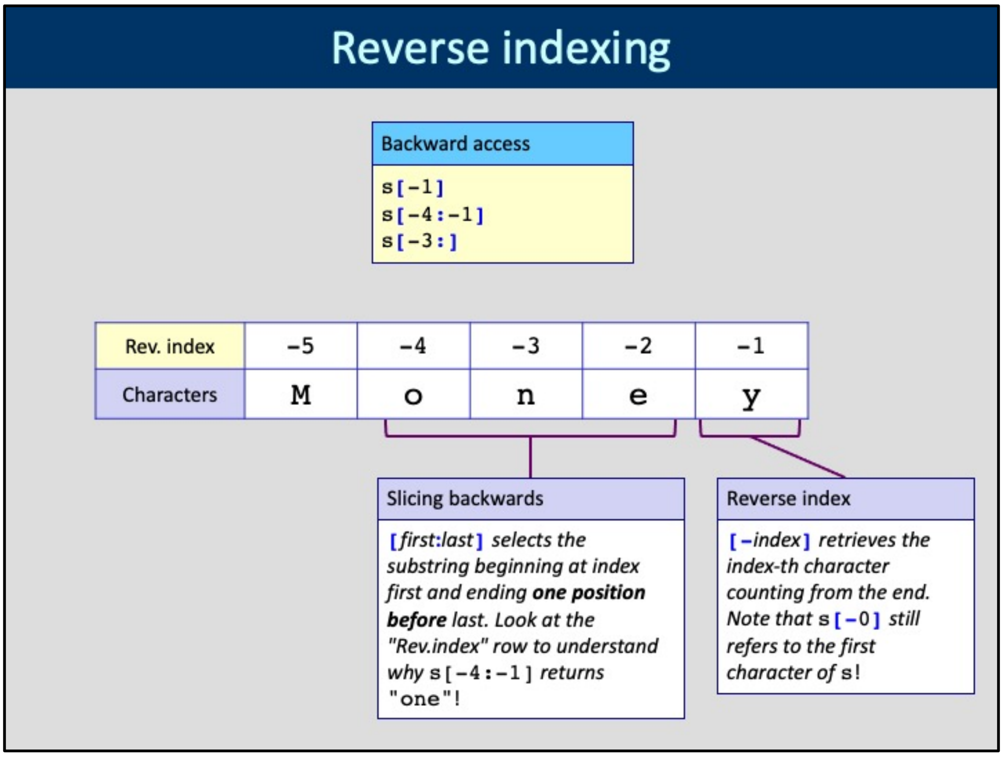
The slide shows only those data structures that we will discuss in detail. There are some other data types that are used only in specialized circumstances, such as `bytes` or `memoryview`, these are not covered in this basic training.



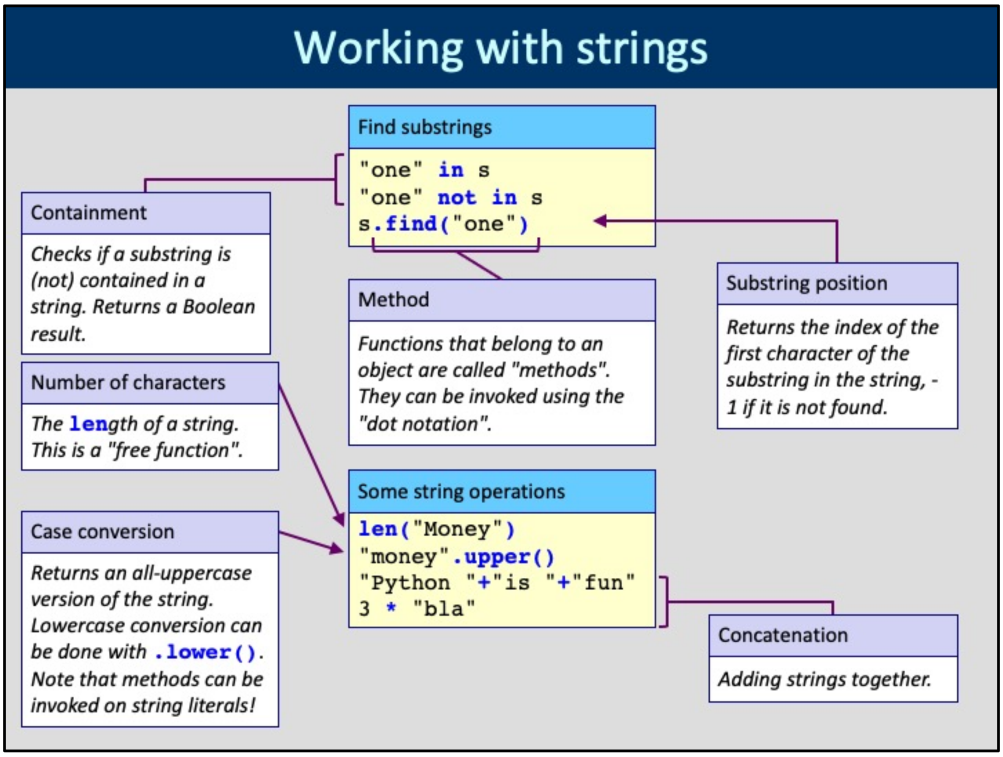
There is no difference between single-quoted and double-quoted strings in Python. It is a matter of taste which one you use, although officially the single quotes are preferred. I rather use the double quotes because in C, C++ and Java they indicate strings (single quotes enclose single characters). Python, however, has no separate character type.



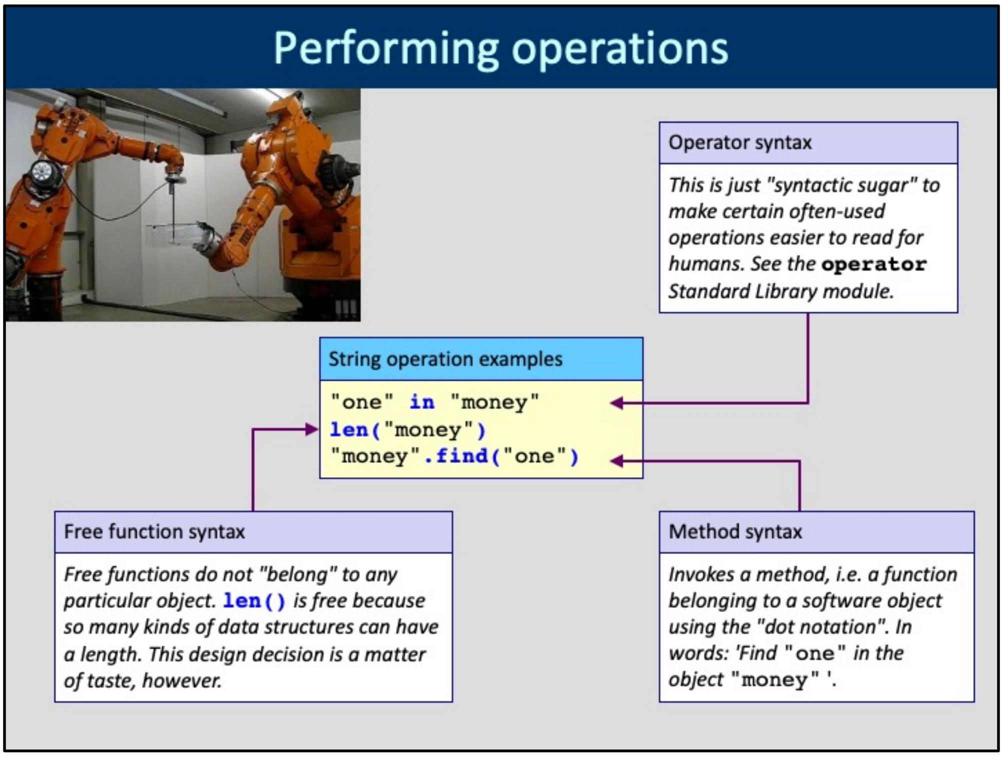
Individual characters and substrings can be accessed using the indexing operator [...]. Indexing is used the same way in all iterable sequences as we will see later.
The general form of the slice index is [first:last:incr] which selects the characters from first to last with increments corresponding to incr. Try s[0:5:2] to see what happens!



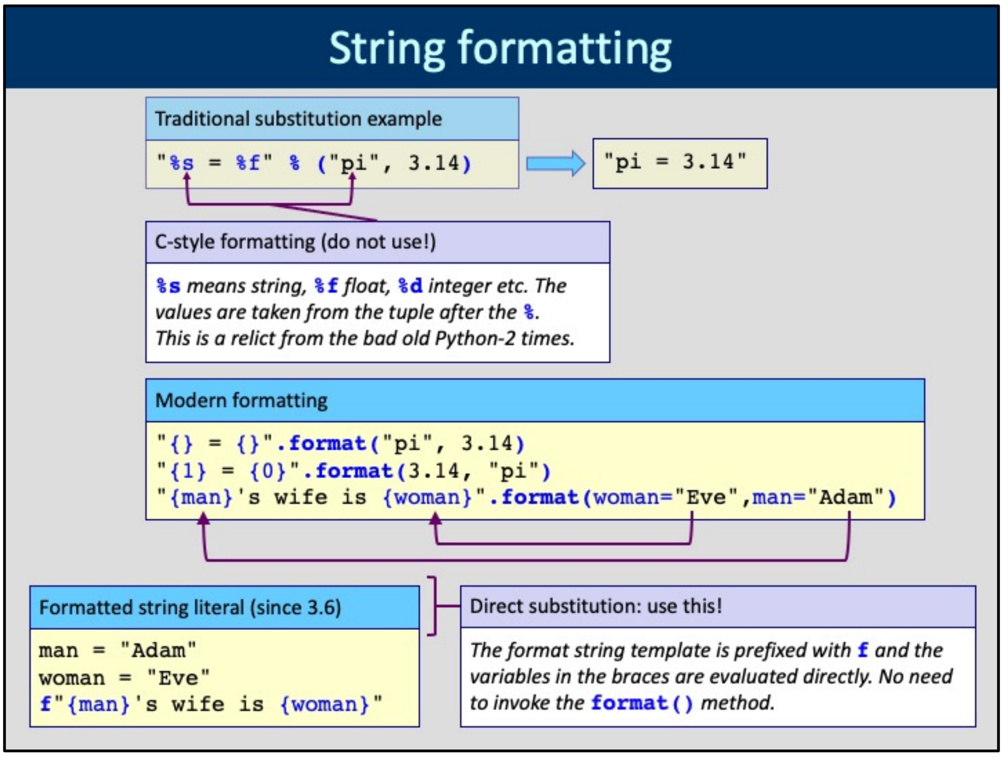
Reverse indexing uses negative indices, with -1 corresponding to the last position of the string. This could be useful to process the string ends ("suffixes"). Slicing is analogous to the normal "forward indexing", see example on the slide.
To get the last N characters of a string, use the index [-N:].
Note that in R a negative index means "not this element"; for instance s[-3] would mean "every element except the third".



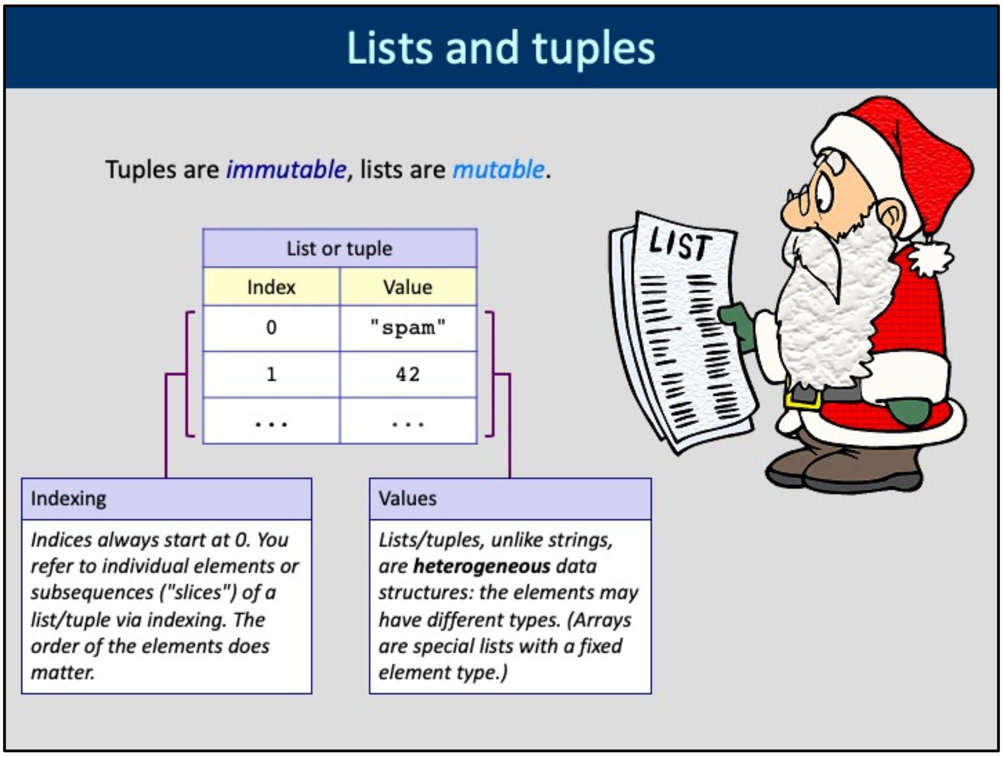
Only a few often-used string methods are shown here.



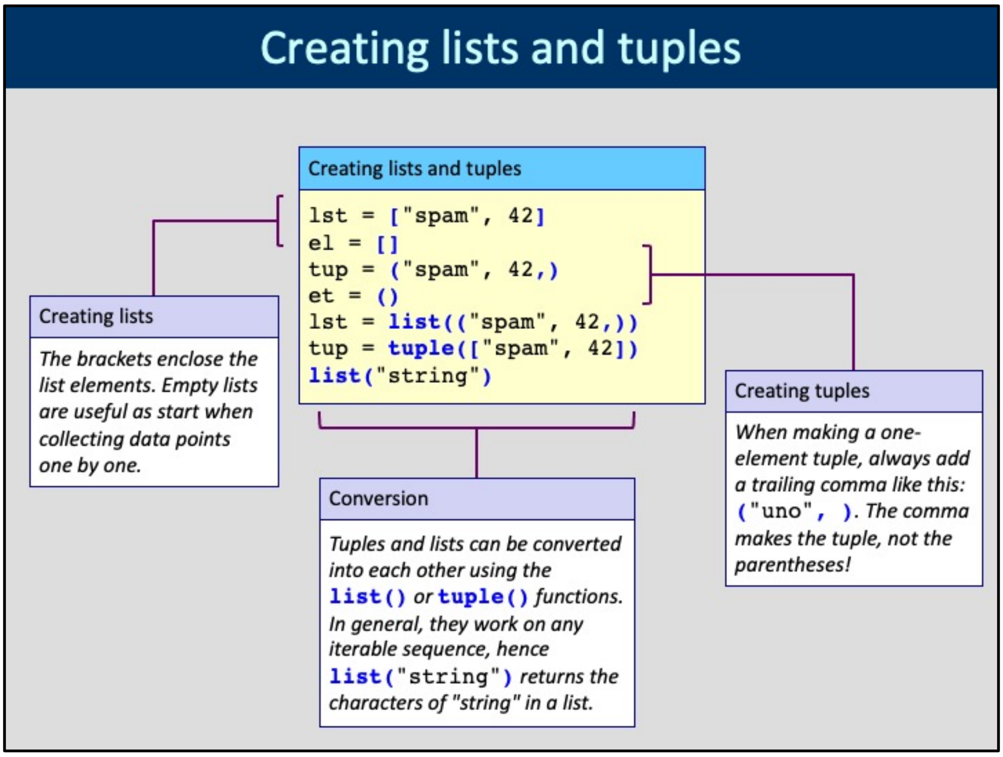
There are several ways of performing operations on a data structure (object) in Python.



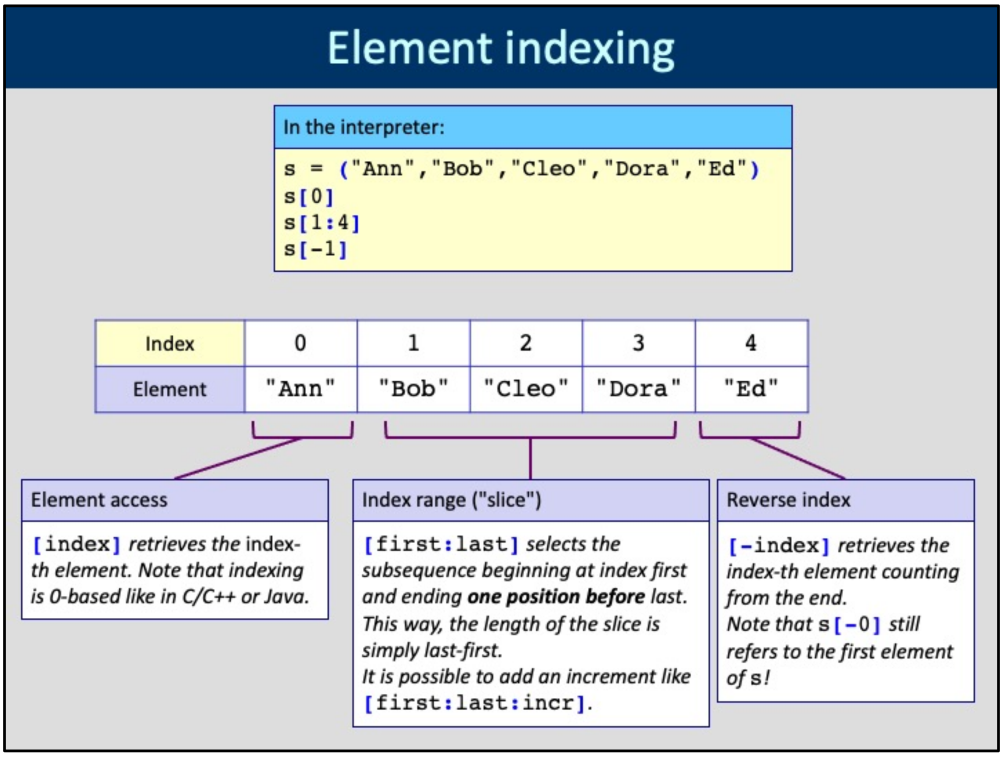
The traditional C-style formatting is mentioned only so that you can recognise it if you see it in older code. Please do not use it when writing new scripts.
The new-style formatting offers lots of options which we cannot all cover in this training. The slide demonstrates just the basic usage. Please refer to the documentation: https://docs.python.org/3/library/string.html#formatstrings



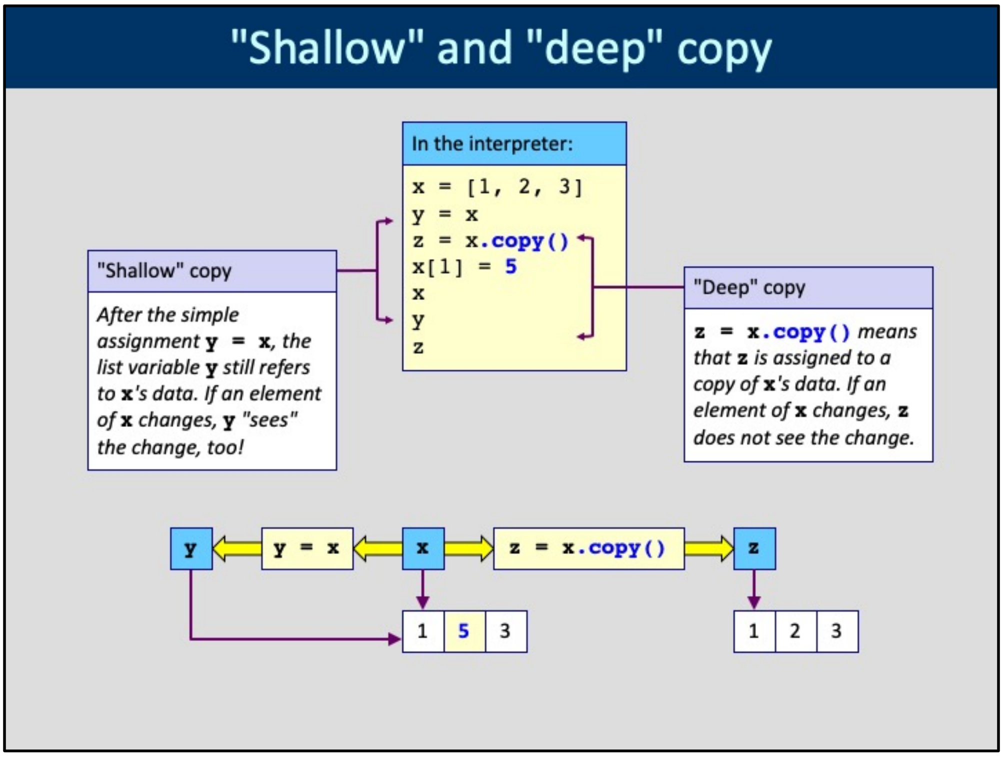
Lists and tuples are both linear sequences which means that the order of the elements does matter. The elements are identified by their indices. Indexing starts at 0, just like in C, C++ or Java.



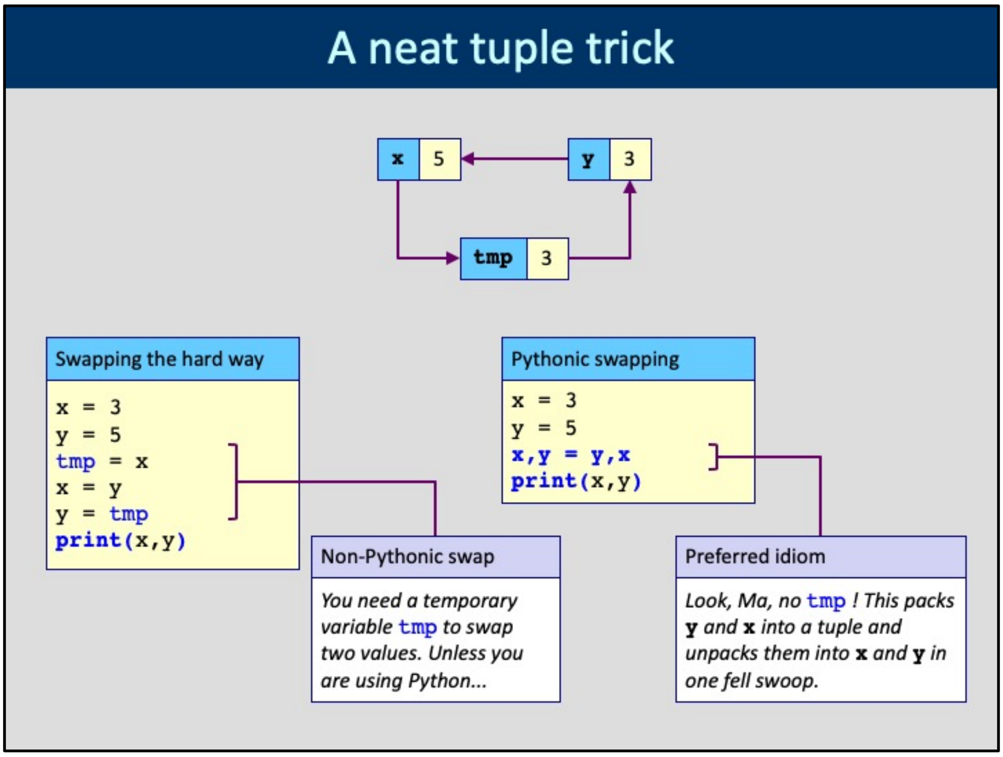
The brackets [] or the parentheses () indicate to Python that you want to create lists or tuples, respectively. Explicit conversion using the list() or tuple() functions are rarely needed.



List (or tuple) indexing works exactly as accessing the characters of a string. In this respect strings are also ordered sequences with the restriction that all the "elements" of a string must be characters.



If you assign a list to another variable, then Python "shallow copies" the list, which means that the underlying data elements are not copied. This improves performance (copying a big list with a million elements can take loooong!), but the price we pay is that the new list variable "sees" all the changes you make to the original list. The solution is to invoke the copy() list method which performs a "deep copy". This means that really all elements of the list are copied, giving you a completely independent new list. In our example, changes to `x` are not seen by `z` and vice versa.



Python often lets you express a programming pattern in a compact way. Swapping two values are a good example.

reference

Common sequence operations

Preparation
`s = ("Joe","Ann","Bob",)`

Operation	Example	Result	Comments
Containment	`"Ann" in s`	`True`	Use `not in` for the opposite effect
	`s.count("Bob")`	`1`	Number of occurrences
Concatenation	`(7,3) + (2,8)`	`(7,3,2,8)`	
	`2 * (7,3)`	`(7,3,7,3)`	`(7,3)*2` also works
Indexing	`s[2]`	`"Bob"`	Indexing starts at 0, negative indices start from last position.
	`s[0:2]`	`("Joe","Ann")`	Slice length is *last-first*
	`s.index("Bob")`	`2`	Index of first occurrence or `ValueError` if not found
Length	`len(s)`	`3`	Separate function, not a method
Minimum/maximum	`min(s)` `max(s)`	`"Ann"` `"Joe"`	Works only if elements are comparable.

These operations are supported by strings, tuples, lists etc.

reference

Mutable sequence operations [1]

Preparation
`t = ["Joe","Ann","Bob",]`

Get back to original state
`t = list(s)`

Operation	Example	New value	Comments
Indexed assignment	`t[2]="Eve"`	`["Joe","Ann","Eve"]`	The indexed member is replaced
	`t[0:2]=["Eve"]`	`["Eve","Bob"]`	Slice will be replaced with a ***sequence***
Copy	`t.copy()`	Returns "deep" copy, the elements are duplicated	
Ordering	`t.reverse()`	`["Bob","Ann","Joe"]`	In-place reversal
	`t.sort()`	`["Ann","Bob","Joe"]`	Elements must be comparable
	`t.sort(reverse=True)`	`["Joe","Bob","Ann"]`	Sort in reverse order (descending instead of ascending)
	`t.sort(key=cmp)`	depends on *cmp*	Use your own element comparison function *cmp* (advanced)

These operations are supported by lists only.

reference

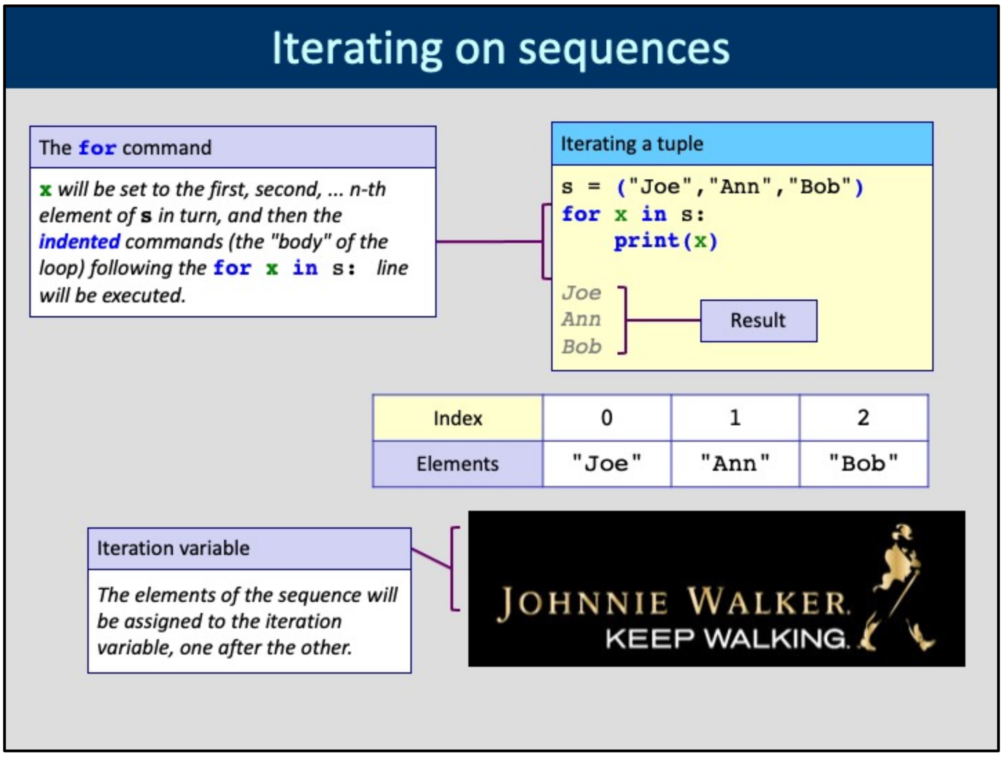
Mutable sequence operations [2]

Preparation
`t = ["Joe","Ann","Bob",]`

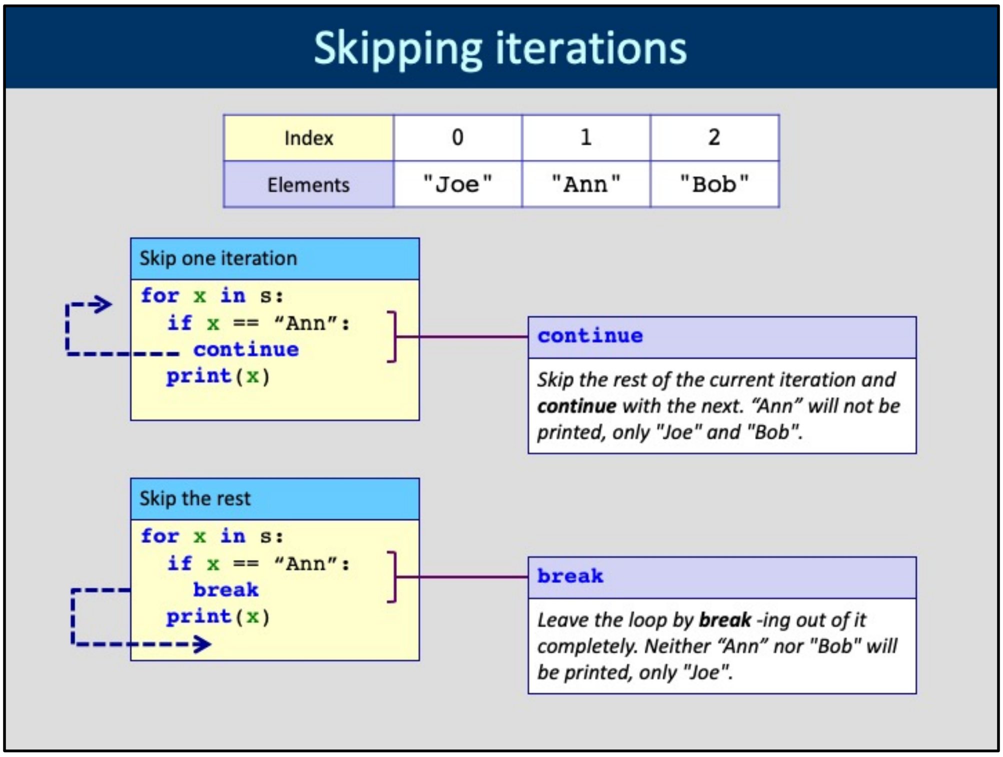
Get back to original state
`t = list(s)`

Operation	Example	New value	Comments
Grow	`t.append("Eve")`	`["Joe","Ann","Bob","Eve"]`	Appends a single element
	`t.extend(( "Lia","Zoe"))`	`["Joe","Ann","Bob", "Lia","Zoe"]`	Adds a sequence to the end
	`t.insert(1,"Guy")`	`["Joe","Guy","Ann","Bob"]`	Inserts element at the position indicated
Shrink	`del t[1]` `del t[1:3]`	`["Joe","Bob"]` `["Joe"]`	Deletes an element or a slice. **del** is a command!
	`t.clear()`	`[]`	Deletes all elements
	`t.remove("Ann")`	`["Joe","Bob"]`	Finds and deletes element
	`t.pop(1)`	`["Joe","Bob"]`, returns `"Ann"`	Returns and deletes *i*-th element, by default the last one if index is omitted

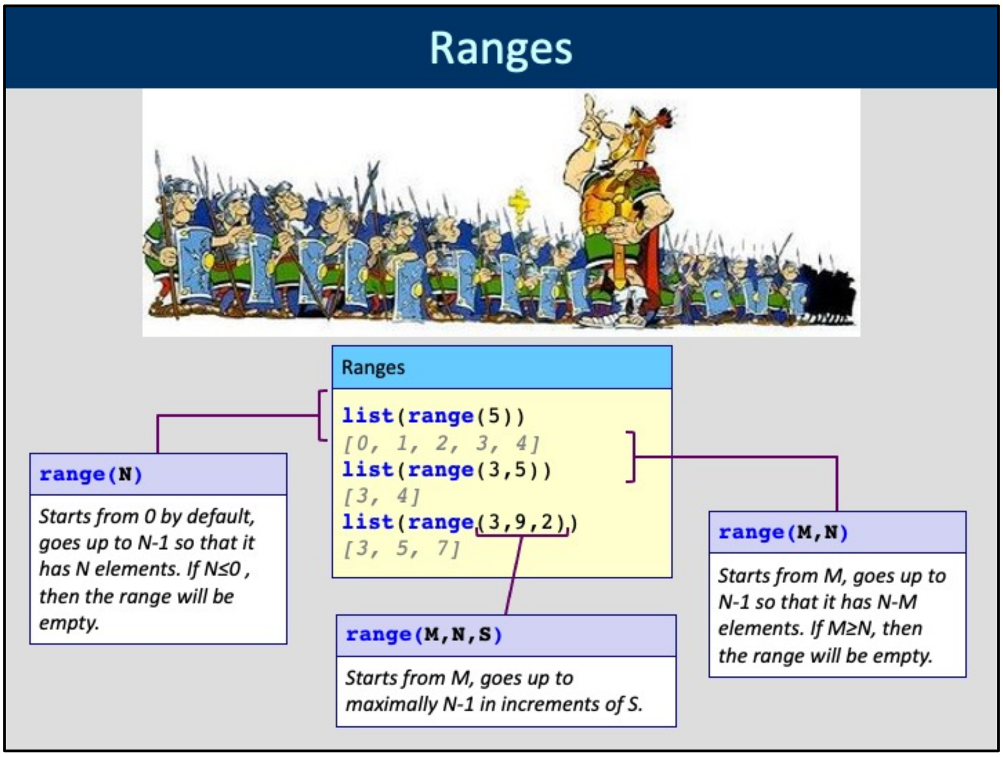
These operations are supported by lists only.



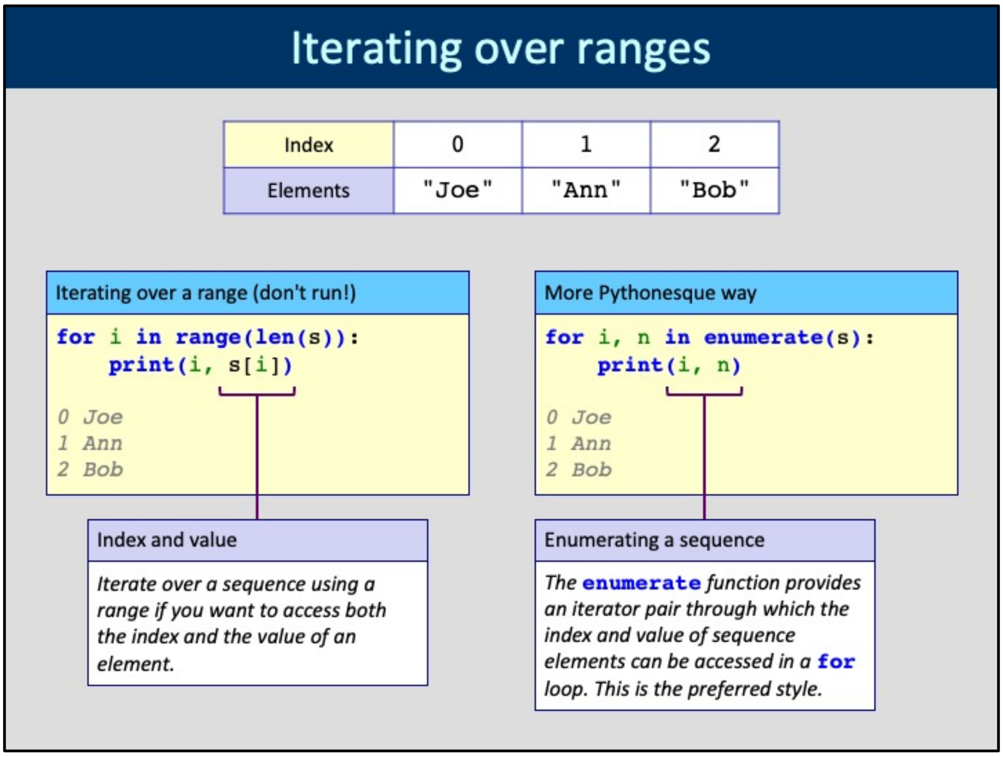
Iteration is a central concept in most programming languages. Python's for statement is best understood on the examples of sequence iteration. There is another iteration construct, the `while` which works differently.
The hands-on example shows iteration over a tuple. Note that other sequences (lists, strings etc.) can be iterated over exactly the same way.



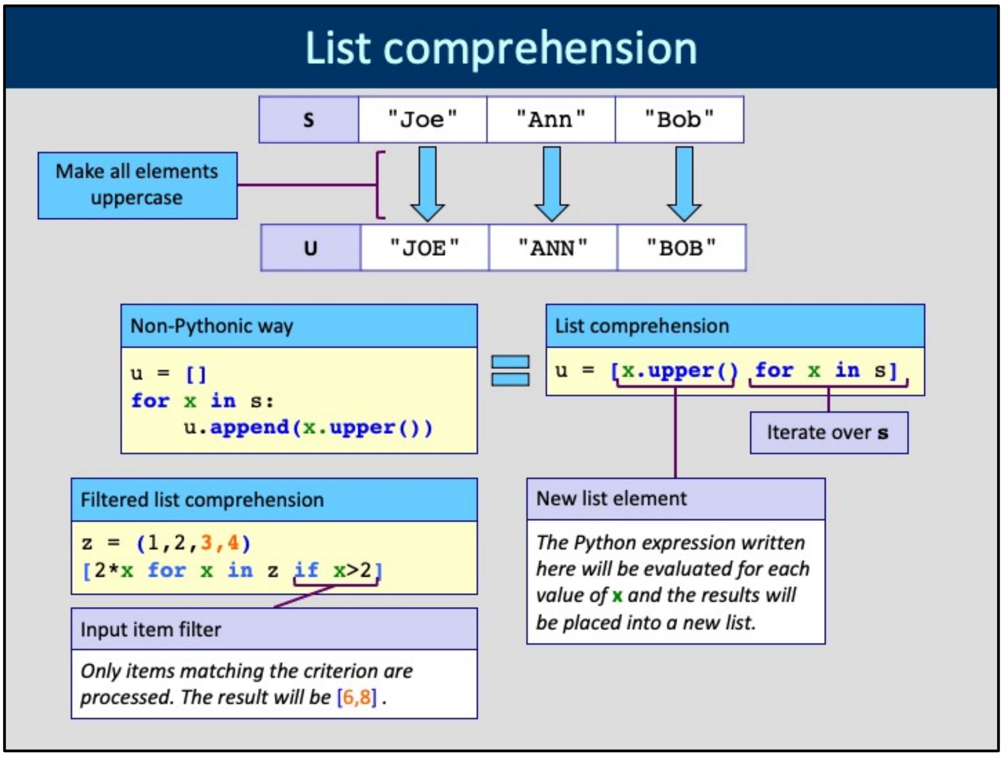
Sometimes we want to skip an iteration or leave the loop earlier than expected. The `continue` and `break` statements are used for these situations. Other programming languages usually offer similar constructs.



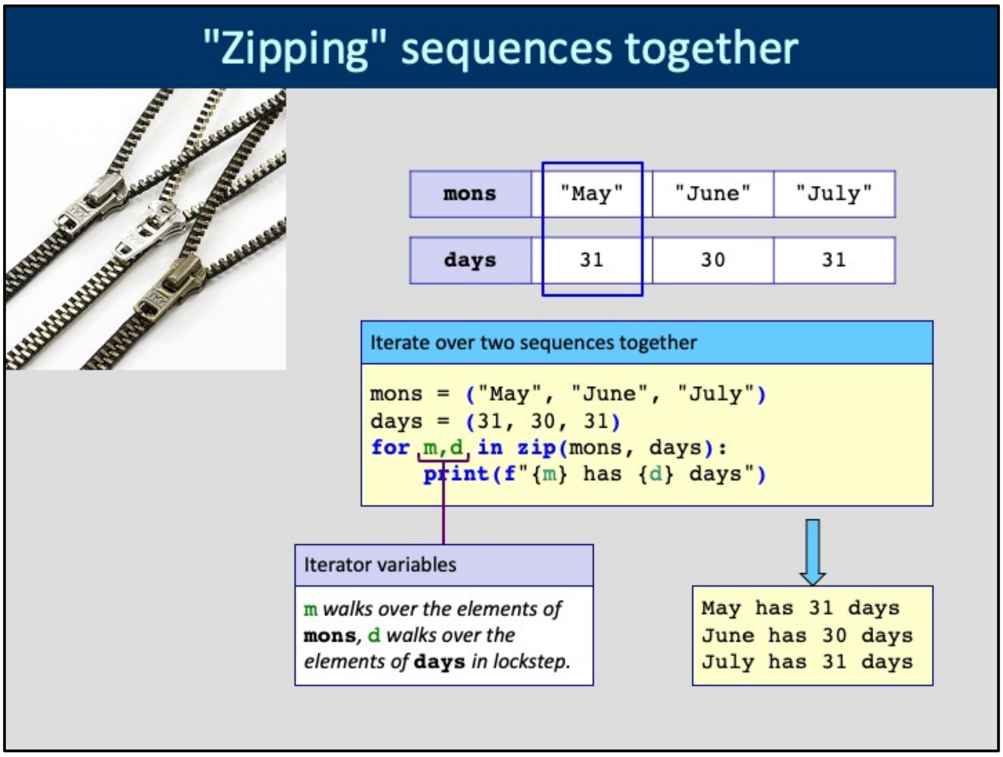
Ranges are objects representing regular integer sequences which are used quite often to iterate over other sequences as we will see on the next slide. Because they are objects, `print(range(3))` will actually print "range(3)". To see the elements of a range, convert it to a `tuple` or a `list` first.



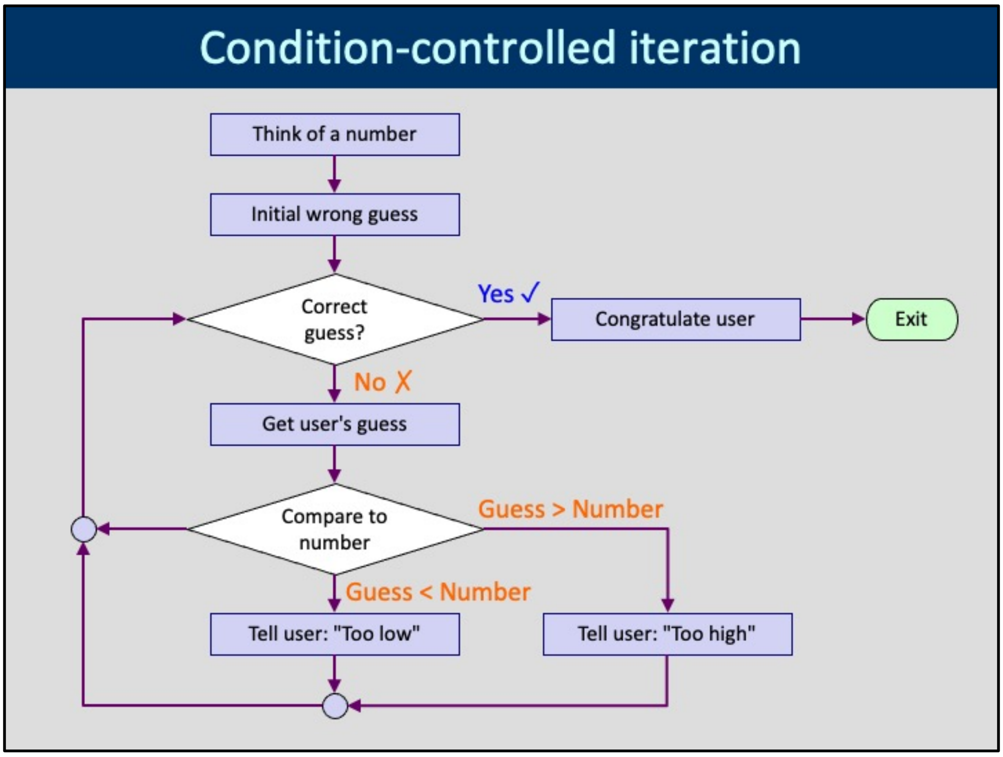
Often we need to iterate over a sequence so that both the index and the value of the elements are required in the loop body. You can do this "analytically" by iterating over the range defined by the length of the sequence and then looking up the value belonging to the i-th index in the loop. The more "elegant" way of doing it is shown on the right hand side. This idiom makes use of the `enumerate` function which returns both the index and the value of the elements of its argument in turn, and then the for loop can refer to both.



List comprehension converts a sequence into a list by applying a transformation to each element of the input. Such operations are very common in practice.

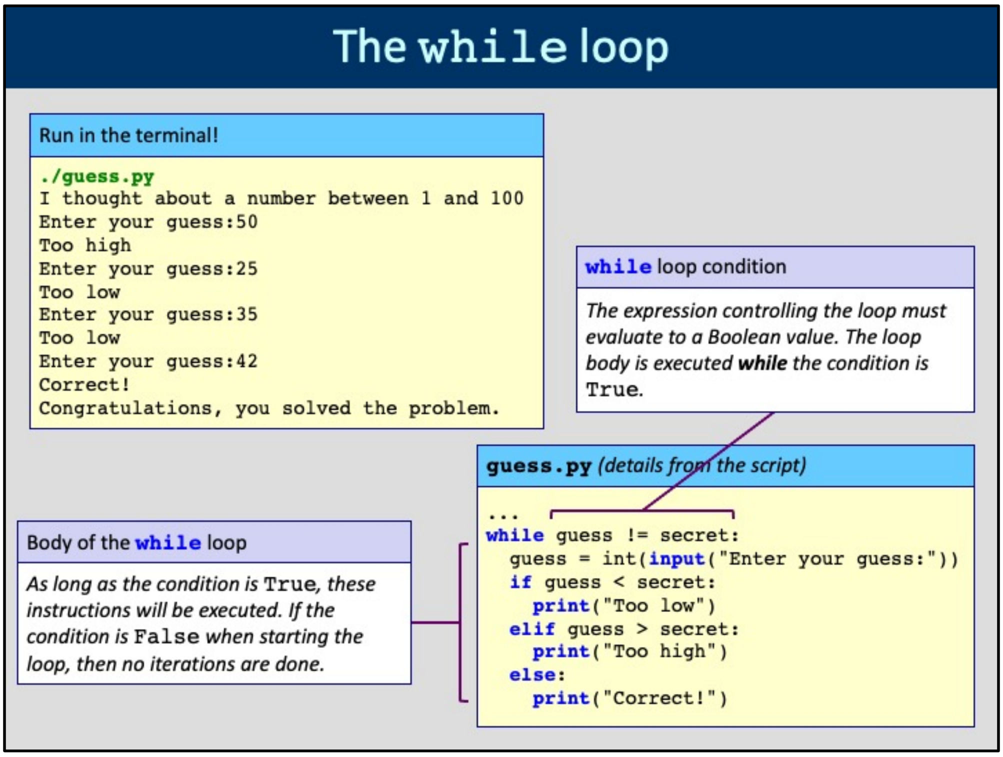


The idiom shown on the slide is very useful if you want to process the elements of two sequences "together". The `zip` function really "zips" them! Formally, `zip` returns an iterator tuple which will be used in the `for` loop. You can refer to the iterator tuple's elements by name as shown here.
Unlike physical zippers, Python's `zip` can zip together even 3 or more sequences. This is rarely used.

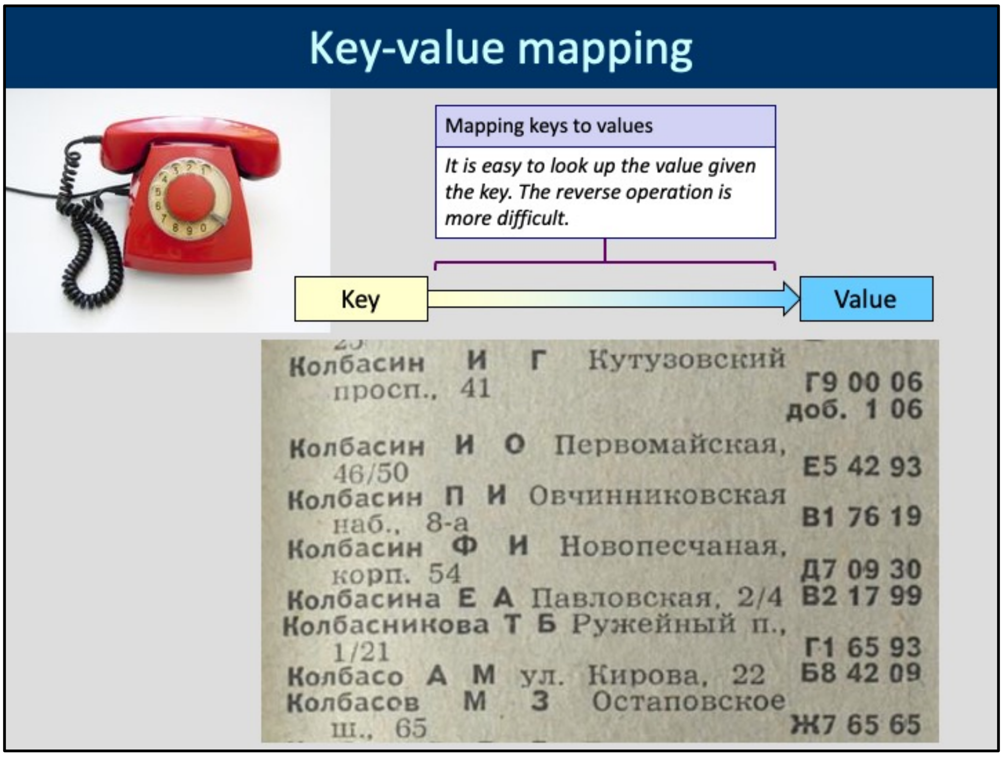


In addition to iterating over sequences, Python supports condition-controlled iteration which means that we execute a list of commands while a certain logical condition is true. This is very useful if we do not know the necessary number of iterations in advance.

We will play a game. Python thinks about an integer number and we have to guess it. The script tells us if our guess is too high, too low or correct.

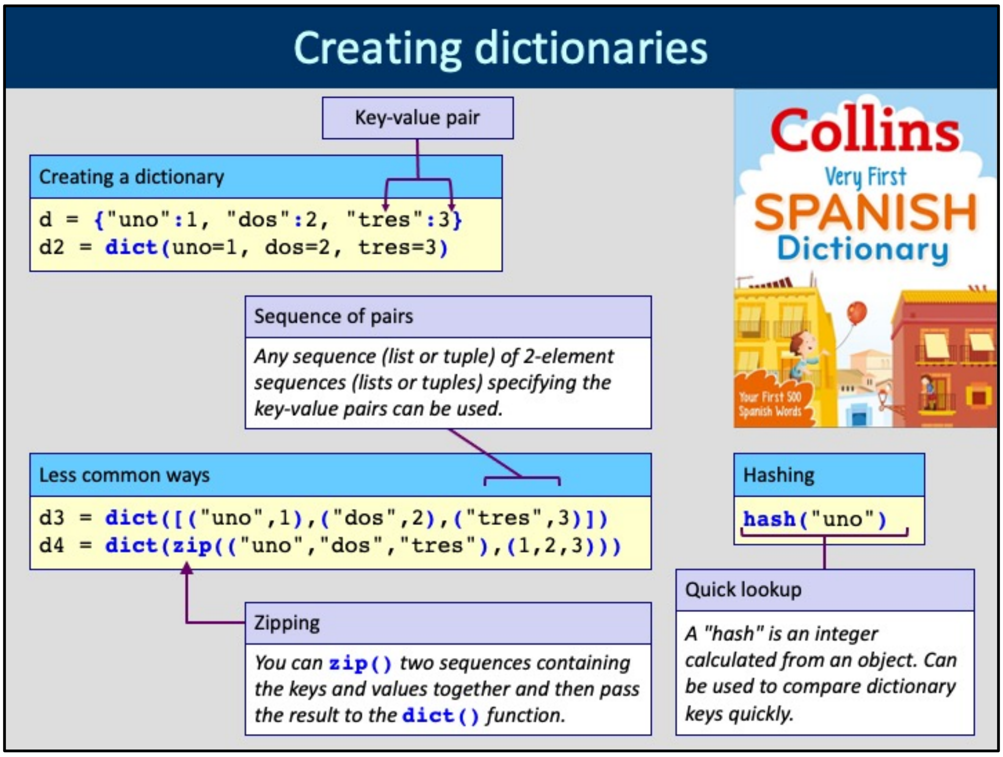


I wrote a script that plays the number guessing game with you. The essential parts are shown on the slide.
What is the best strategy, i.e. how can you guess the secret number in as few steps as possible?



Often we need a data structure that knows about "associations" between items, much in the same way as a phone book associates people with their numbers. People's names are the "keys" and their phone numbers are the corresponding "values". If you know a person's name, it's easy to look up his/her number in the phone book: the relationship between the key and its corresponding value is unidirectional.

In Python such a data structure is called a "dictionary". Other programming languages may call it an "associative map" or a "lookup table".



Dictionary keys must be "hashable". A hash function makes an integer number out of an object (how this is done would take us too far). Python uses the key hash values to speed up dictionary lookup.
Not all data types are "hashable". For instance, tuples can be dictionary keys, but lists can't. Most often we use strings as dictionary keys.
The values of a dictionary, on the other hand, can be anything, including lists, lists of lists, other dictionaries, ... etc.

reference

Read-only dictionary operations

Preparation
`d = {"uno":1, "dos":2, "tres":3}`

Operation	Example	Result	Comments
Containment	`"tres" in d`	`True`	Check if a ***key*** is present. Use `not in` for the opposite effect
Lookup	`d["dos"]`	2	Raises `KeyError` if key is not found
	`d.get("uno")`	1	Returns `None` if key is not found
	`d.get("cinco", 99)`	99	Returns the 2nd parameter (the default value) if key is not found
Length	`len(d)`	3	Separate function, not a method
Copy	`d.copy()`	Returns "deep" copy, the elements are duplicated	
Minimum/maximum	`min(d)` `max(d)`	`"dos"` `"uno"`	Works on the keys, not on the values
Views and conversions to sequences. Insertion order is preserved since Python 3.8	`list(d.keys())`	`["uno","dos","tres"]`	Sequence from the key view (works with `tuple()` as well)
	`list(d.values())`	`[1,2,3]`	Sequence from the value view
	`list(d.items())`	`[("uno",1),("dos",2),("tres",3)]`	Sequence from the key/value pairs as tuples

In principle dictionaries are not sequences, i.e. the order of the key-value pairs is not well-defined. That was the case until Python 3.7. Since Version 3.8, Python dictionaries preserve item insertion order. However, you are well advised not to rely on this feature.

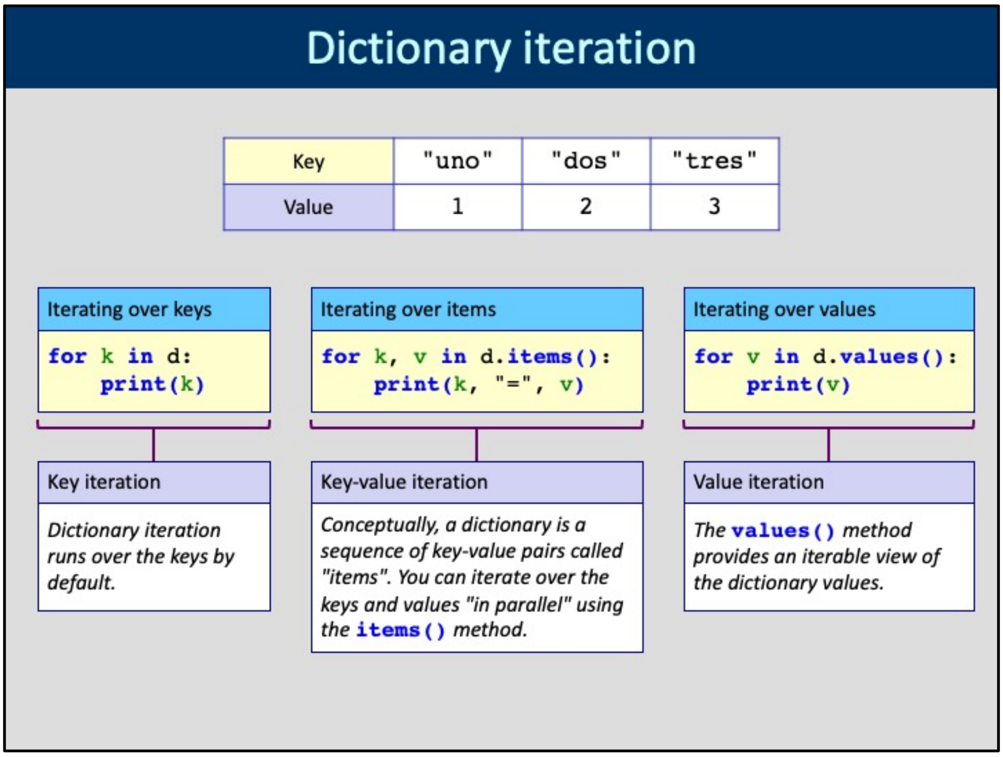
reference

Mutable dictionary operations

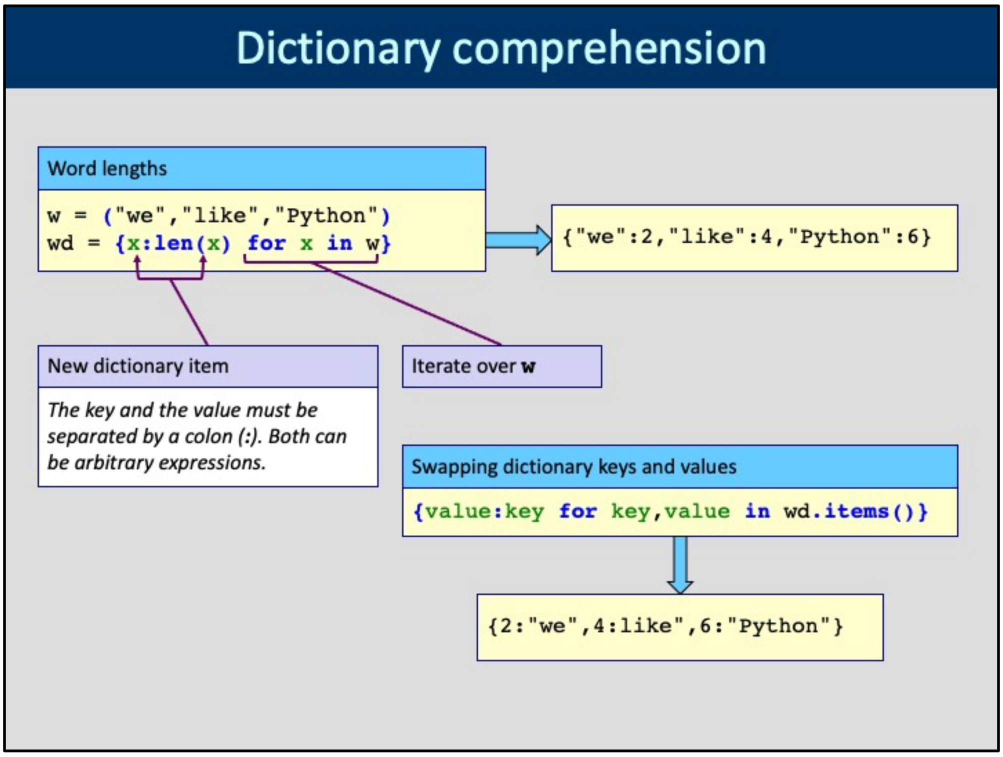
Preparation
`d0 = d`

Reset to original
`d = d0`

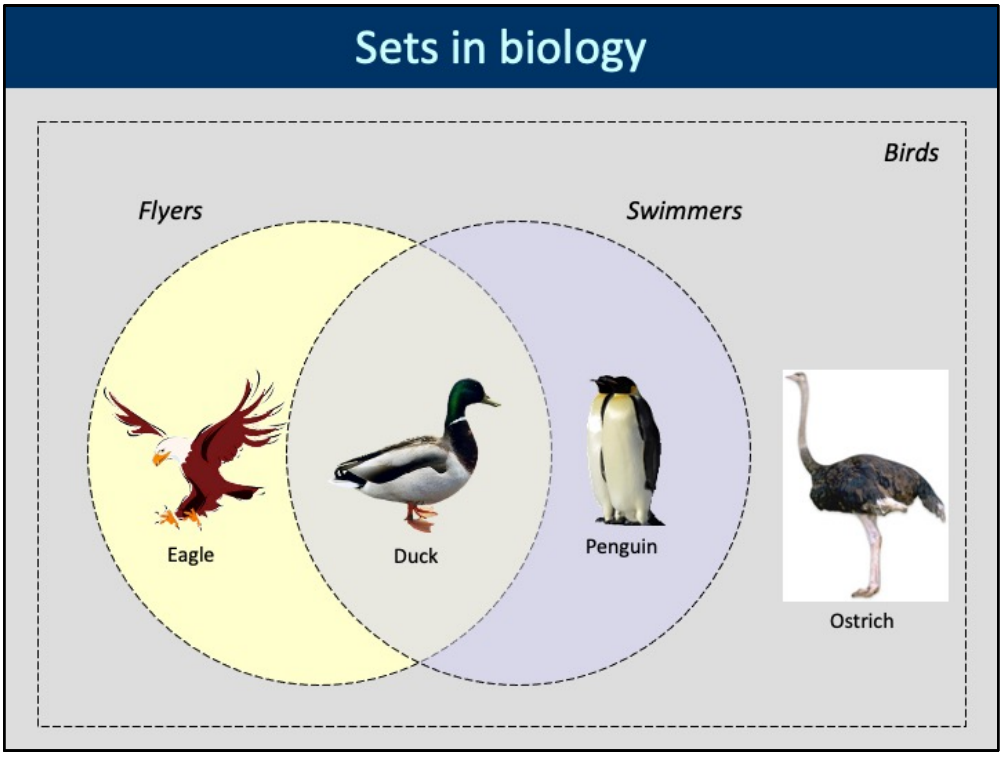
Operation	Example	New value*	Comments
Indexed assignment	`d["dos"] = 22`	`{"uno":1, "dos":22, "tres":3}`	The value is replaced
Grow	`d["cinco"] = 5`	`{"uno":1, "dos":2, "tres":3, "cinco":5}`	A new key/value pair is added
	`d.update( {"seis":6,"ocho":8})`	`{"uno":1, "dos":2, "tres":3, "seis":6, "ocho":8}`	The contents of the argument is merged into the calling object. Values for matching keys will be overwritten
Shrink	`del d["dos"]`	`{"uno":1, "tres":3}`	Deletes by key. Raises **KeyError** if key is not found!
	`d.pop("dos")`	`{"uno":1,"tres":3}` *Returns* 2	Returns value by key and then deletes it. Raises **KeyError** if key is not found!
	`d.clear()`	`{}`	Deletes all elements



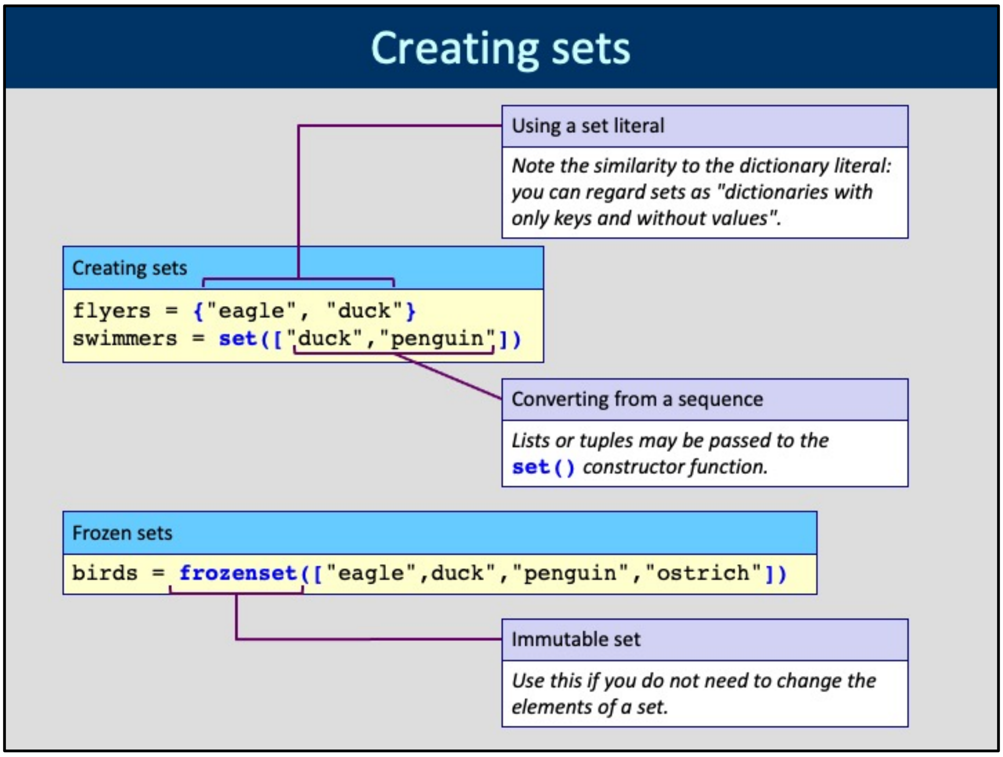
Because you can look up the values via their keys, iterating a directory over its keys should be sufficient. This is why by default directory iteration runs over the keys, although you can get an iterable object by invoking the `keys()` method, which is rarely used. Iterating over the values or "in parallel" over the items can be convenient.
Remember that dictionaries are not sequences, and before Python 3.8 item order was unspecified. Since Version 3.8 dictionaries preserve item insertion order.



Dictionary comprehension is analogous to list comprehension and offers an elegant way of swapping keys and values. Note that the values must be hashable in order to serve as keys.



Sets are mathematical objects that group "things" together without any particular order. Among birds we may define the set of those that can fly ("flyers") and those that can swim ("swimmers"). A duck belongs to both sets. Eagles can fly but not swim, and penguins can swim but not fly. The poor ostrich belongs to neither set.



Set construction is quite similar to how dictionaries are built.

reference

Read-only set operations

<table>
<tr><th>Operation</th><th>Example</th><th>Result</th><th>Comments</th></tr>
<tr><td>Membership</td><td>"eagle" in flyers</td><td>True</td><td>Use not in for the opposite effect</td></tr>
<tr><td rowspan="4">Containment</td><td>flyers.issubset(birds)
flyers <= birds</td><td>True</td><td>Subset (incl. equality)</td></tr>
<tr><td>flyers.issuperset(birds)
flyers >= birds</td><td>False</td><td>Superset (incl. equality)</td></tr>
<tr><td>flyers < birds</td><td>True</td><td>Proper subset (full containment)</td></tr>
<tr><td>flyers > birds</td><td>False</td><td>Proper superset</td></tr>
<tr><td rowspan="4">Set operations</td><td>flyers | swimmers</td><td>{"eagle",
"duck",
"penguin"}</td><td>Union</td></tr>
<tr><td>flyers & swimmers</td><td>{"duck"}</td><td>Intersection</td></tr>
<tr><td>flyers - swimmers</td><td>{"eagle"}</td><td>Difference</td></tr>
<tr><td>flyers ^ swimmers</td><td>{"eagle",
"penguin"}</td><td>Symmetric difference</td></tr>
<tr><td>Cardinality</td><td>len(birds)</td><td>4</td><td>Number of elements in the set</td></tr>
<tr><td>Copy</td><td>birds.copy()</td><td colspan="2">Returns "deep" copy, the elements are duplicated</td></tr>
</table>

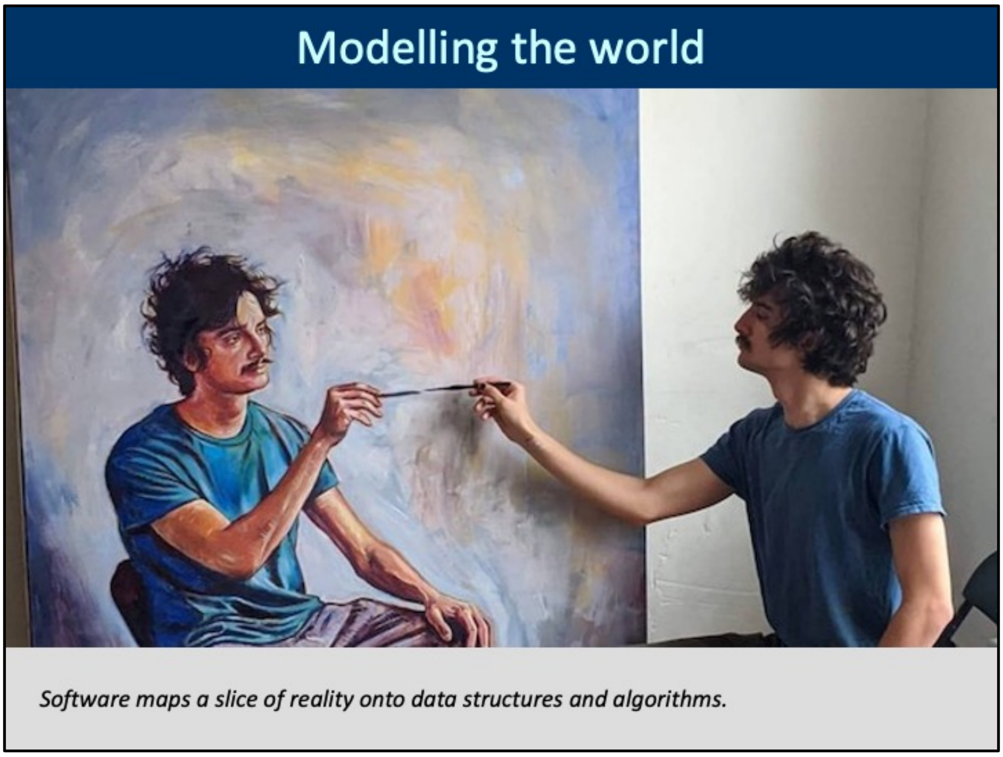
These methods are supported both by `set` and `frozenset`.
The containment operations set<=other and set>=other are available as methods in the form of set.issubset(other) and set.issuperset(other), respectively. The argument `other` can be an iterable sequence, not just a set.
The set operations |, &, -, ^ can be invoked as the methods set.union(other), set.intersection(other), set.difference(other), set.symmetric_difference(other) as well. In these "non-operator" methods the parameter `other` can be any iterable sequence, not just a set.

reference

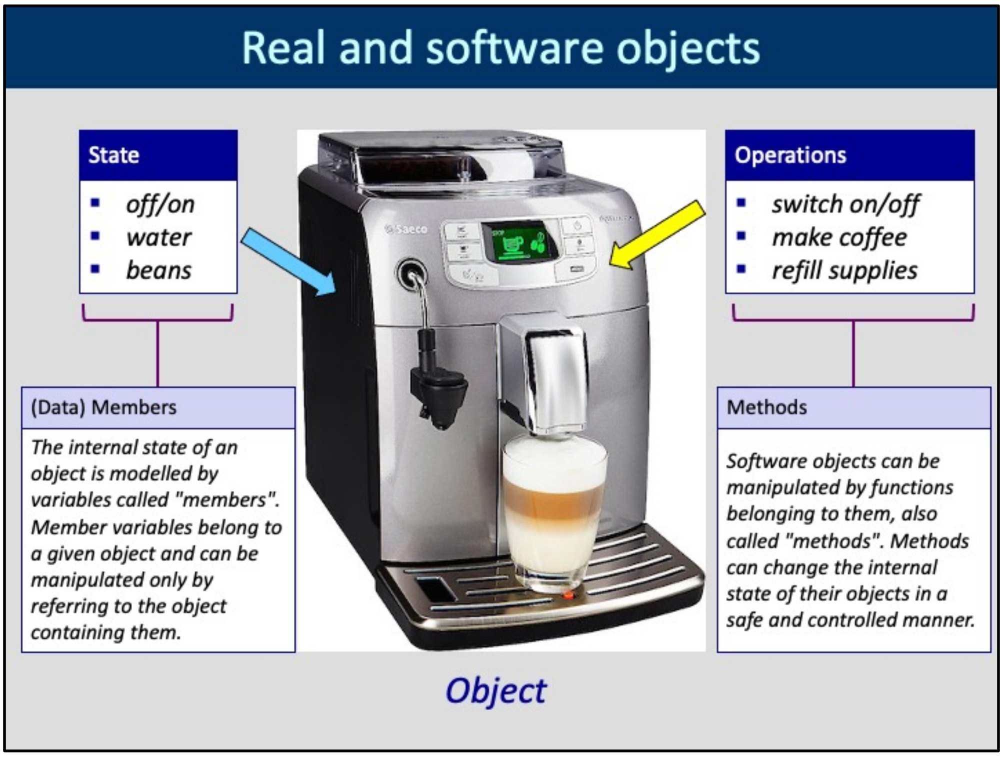
Mutable set operations

Operation	Example	New value*	Comments
In-place set operations	`flyers |= swimmers`	`{"eagle","duck", "penguin"}`	In-place union
	`flyers &= swimmers`	`{"duck"}`	In-place intersection
	`flyers -= swimmers`	`{"eagle"}`	In-place difference
	`flyers ^= swimmers`	`{"eagle","penguin"}`	In-place symmetric difference
Grow	`flyers.add("swift")`	`{"eagle","duck", "swift"}`	A new element is added
	`flyers.update( {"magpie","swift"})`	`{"eagle","duck", "magpie","swift"}`	The contents of the argument is merged into the calling object.
Shrink	`flyers.remove("duck")` `flyers.discard("duck")`	`{"eagle"}`	Deletes an element. `remove()` raises `KeyError` if the element is not found!
	`flyers.pop()`	Deletes an element *randomly* and returns it. Use for "consuming" a set in an iteration loop.	
	`flyers.clear()`	`{}`	Deletes all elements

The in-place set operations |=, &=, -=, ^= can be invoked as the methods set.union_update(other), set.intersection_update(other), set.difference_update(other), set.symmetric_difference_update(other) as well. In these "non-operator" methods the parameter `other` can be any iterable sequence, not just a set.

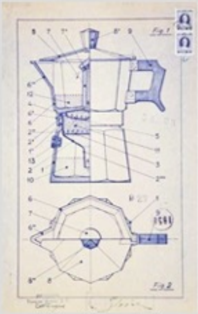


When we write a program, we always model "real" things in software, where objects of mathematical reality (e.g. numbers) also count as "real". There are more than one ways to model the same entity.



In this training we will model my espresso machine.
Real objects have internal states. For instance, a coffee machine can be in an OFF or an ON state. The amount of water and beans also belong to its internal state.
Software objects model the internal state by appropriately chosen variables. We say that these are "member" variables because they belong to ("are the members of") a given object. This is in contrast to the "free-standing" variables we have used until now; they were not "owned by" any particular object.
Real objects can be manipulated by us. For instance, a coffee machine can be switched on or off, you can press a button to make coffee, you can fill up the water tank or the beans holder. Similarly we can manipulate software objects by invoking "member functions", also known as "methods". Methods belong to objects much in the same way as member variables. They may change the internal state of the object they belong to. Free-standing functions had no such special relationship with any of the free variables: they just take arguments.

Type, class, instance



The blueprint ("type") of a coffee machine

Instances of coffee machines (yours, mine...)

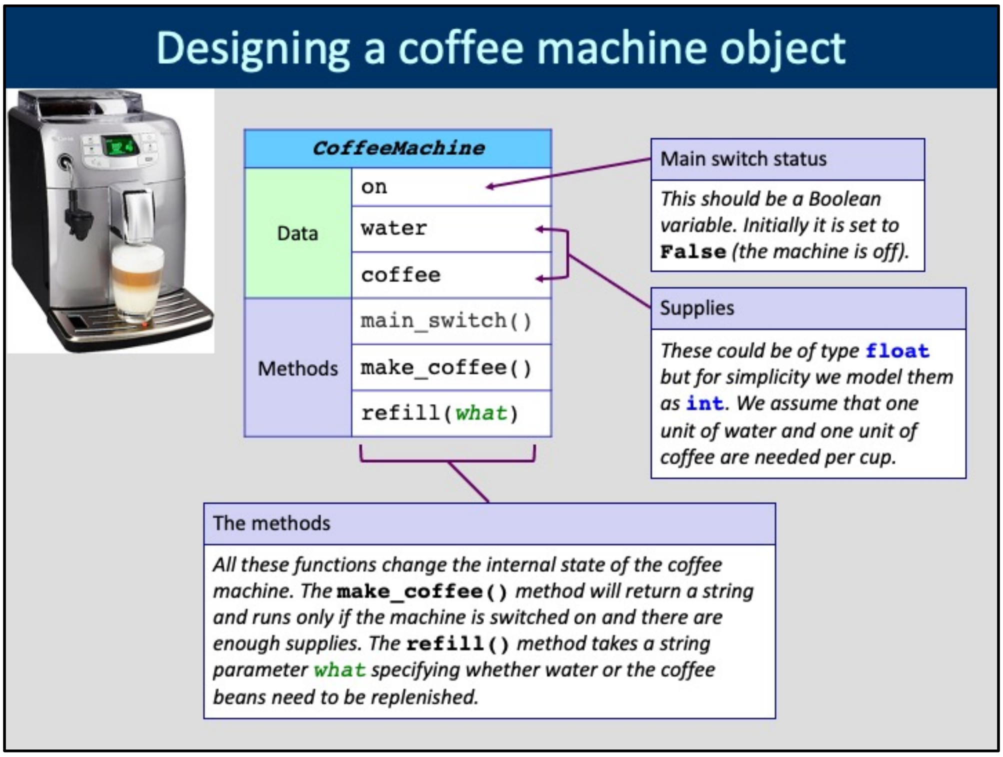
	Simple types	Objects
Concept modelled	*"An integer number"*	*"A coffee machine"*
The ***type***	`int`	`class CoffeeMachine`
Variables storing ***instances***	`x = 42` `y = int(3.1)`	`cm1 = CoffeeMachine()` `cm2 = CoffeeMachine()`

Is there a difference?

Actually, no. In Python, really and truly ***everything is an object****. The built-in type* `int` *is a class, a blueprint for storing and handling integer numbers. Each integer variable stores an instance of an integer object. Only the syntax is different when you define your own classes.*

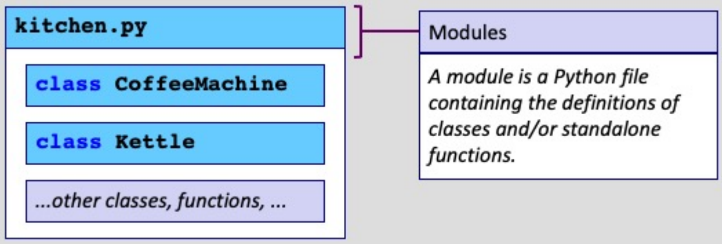
We have seen in the Introduction that types represent the properties of data. Objects also have a type that specifies their properties, it is called the "class" of the object. A class can be regarded as the "Platonic ideal" of the objects being modelled, or a "blueprint" or "recipe" that defines the objects. We say that an individual object is the instance of its class.

Because in Python everything is an object, the built-in types we have seen so far are also classes. Their instances store data (an int object stores an integer number, a str object stores a sequence of characters, etc...) and they have methods associated with them that define what you can do with the data.



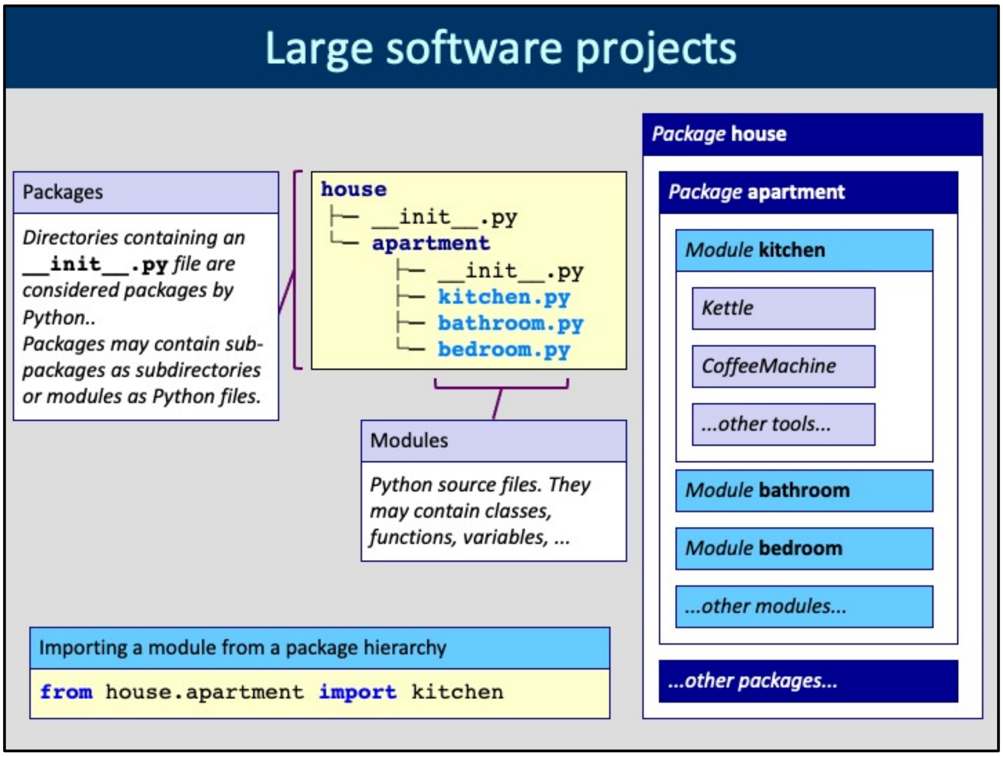
This is how a CoffeeMachine object would look like. It is essentially a data structure that has methods operating on its data. Together they define the internal state and the behaviour of the object. The programmer must think very carefully about which features s/he wishes to model, this design phase can take quite long in more complex cases.

Where to put the class definition?

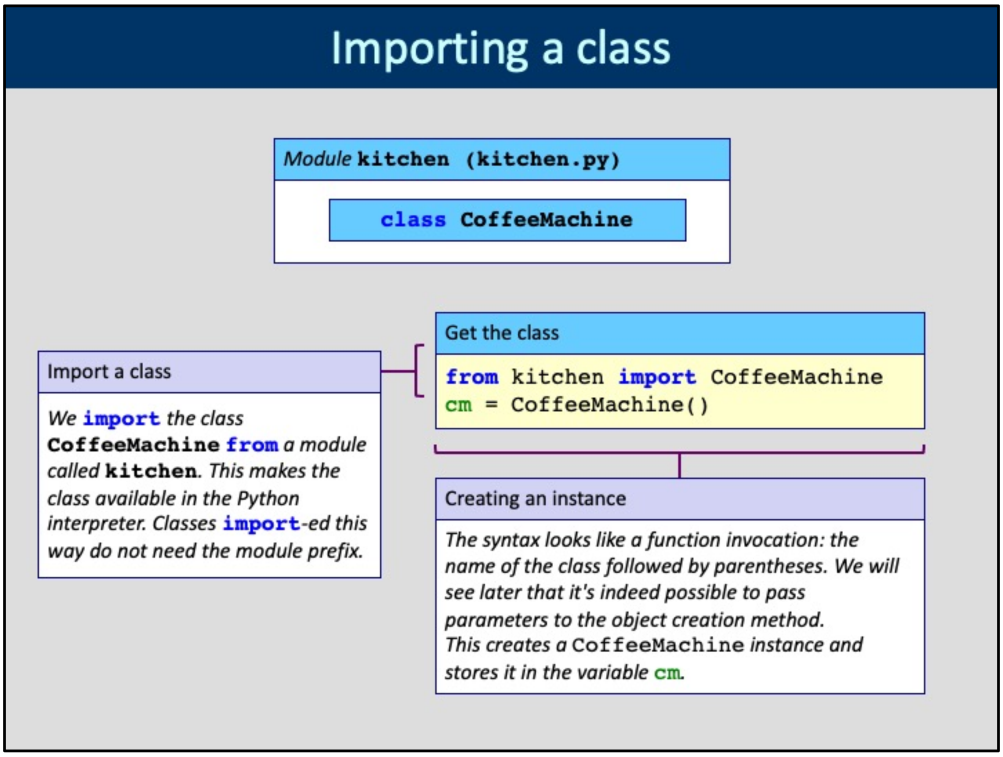


Import command	Access an item	explanation
`import kitchen`	`kitchen.Kettle`	*Everything is imported, module name prefix **is required***
`from kitchen import *`	`Kettle`	*Everything is imported, module name prefix **not required***
`from kitchen import Kettle`	`Kettle`	*Only the named item(s) are imported, module name prefix **not required***

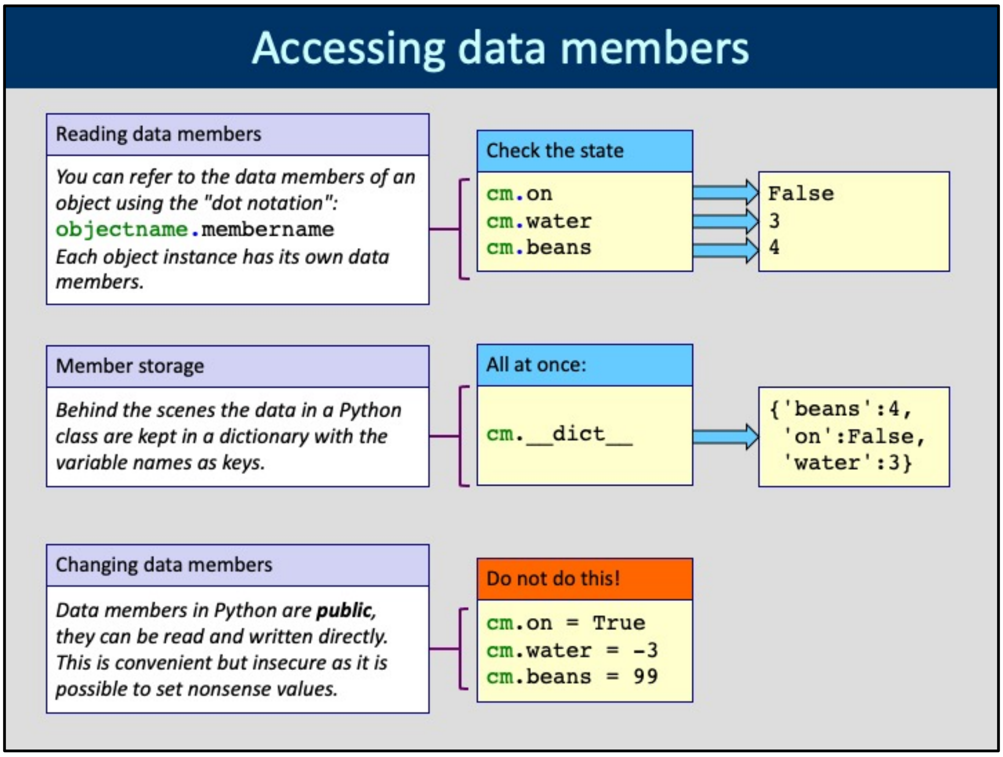
Class definitions are usually kept in separate source files called "modules". Modules help organise the source for larger projects. They may contain stand-alone function definitions and data as well.
To use the entities in a module, they have to be imported first.



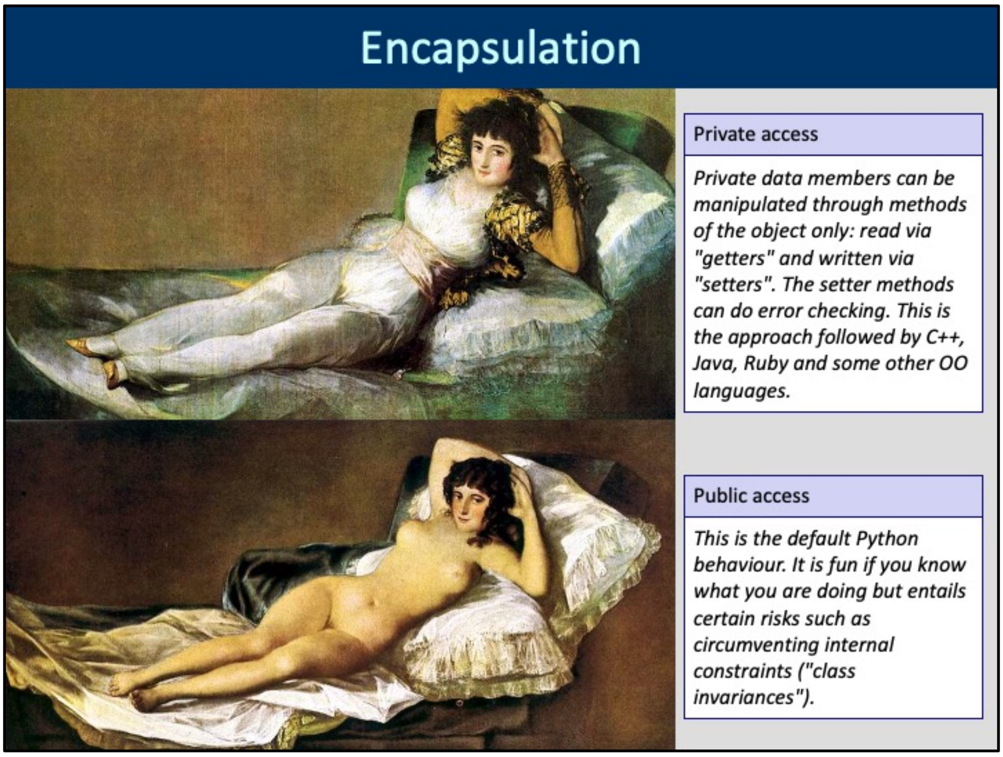
Python uses *packages* and *modules* to organise large software projects. Modules are Python source files which can contain classes, functions or pieces of data. Modules can be bundled together in packages that are represented by directories in the file system. Packages may also contain other packages. A package directory is labelled by a (usually empty) file with the name "__init__.py".



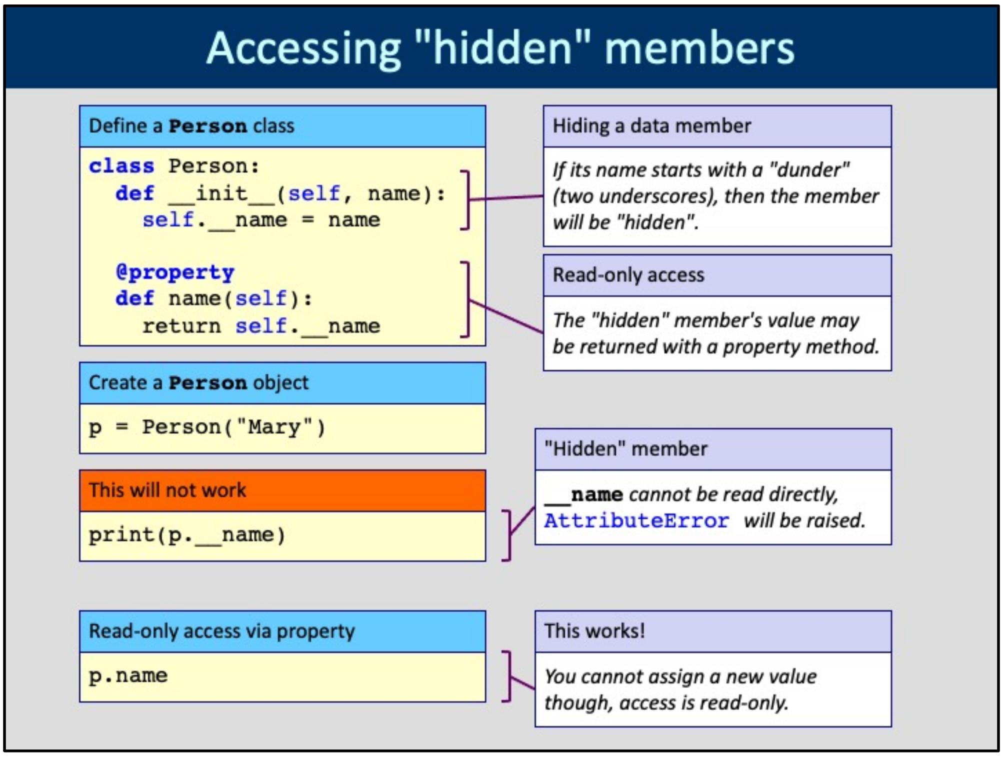
We use here the from ... import ... syntax to get access to the CoffeeMachine class. It is also possible to import everything from a module, but in that case we need to prefix each class name with the module name using the dot notation which is quite cumbersome.
To create an instance of a class, we invoke its name as if it were a function. What really happens in the background is that two methods, __new__() and __init__() are invoked. When you write your own class, you can define the __init__() method yourself: here you can initialise the members of the object and prepare it for first use. This is called a "constructor" in other object-oriented languages.



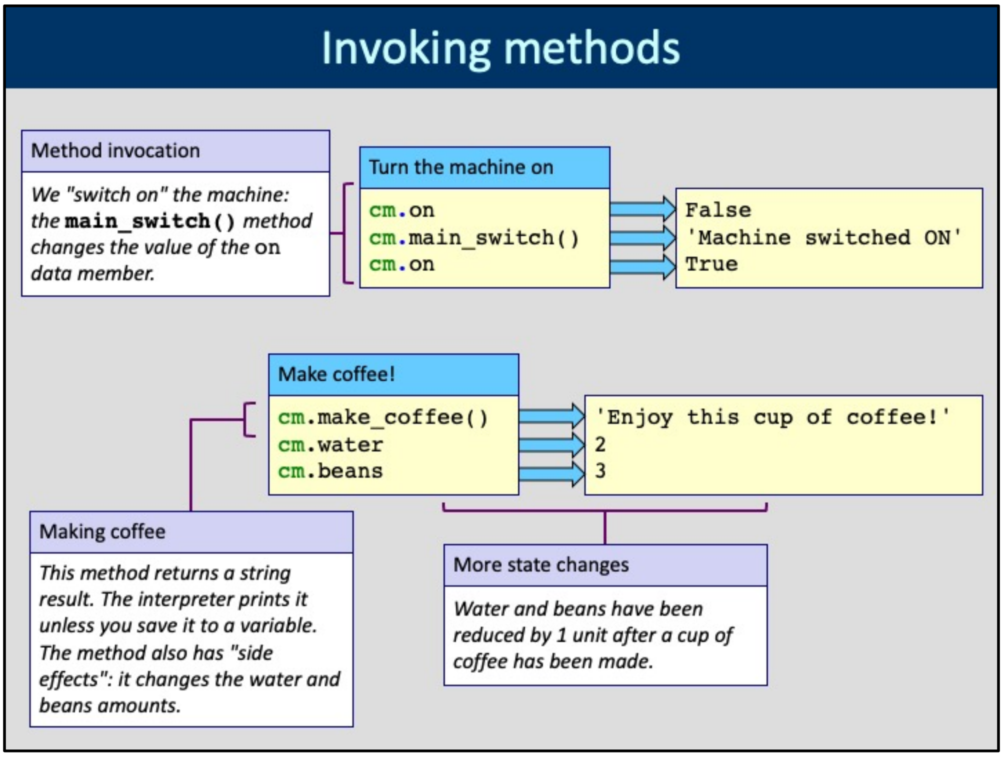
As we have seen already when invoking member functions (methods), the "dot notation" in Python expresses a "belongs-to" relationship. For instance, `cm.on` means that the data member `on` belongs to the object stored in `cm`. Since each instance is different, the dot notation is needed to distinguish between the on/off status of my CoffeeMachine from yours.
Python allows direct manipulation of the data members because it can be convenient. This convenience, unfortunately, also allows the user to set member variables to some nonsense values, like in the example on the slide where the amount of water is set to -3 units.



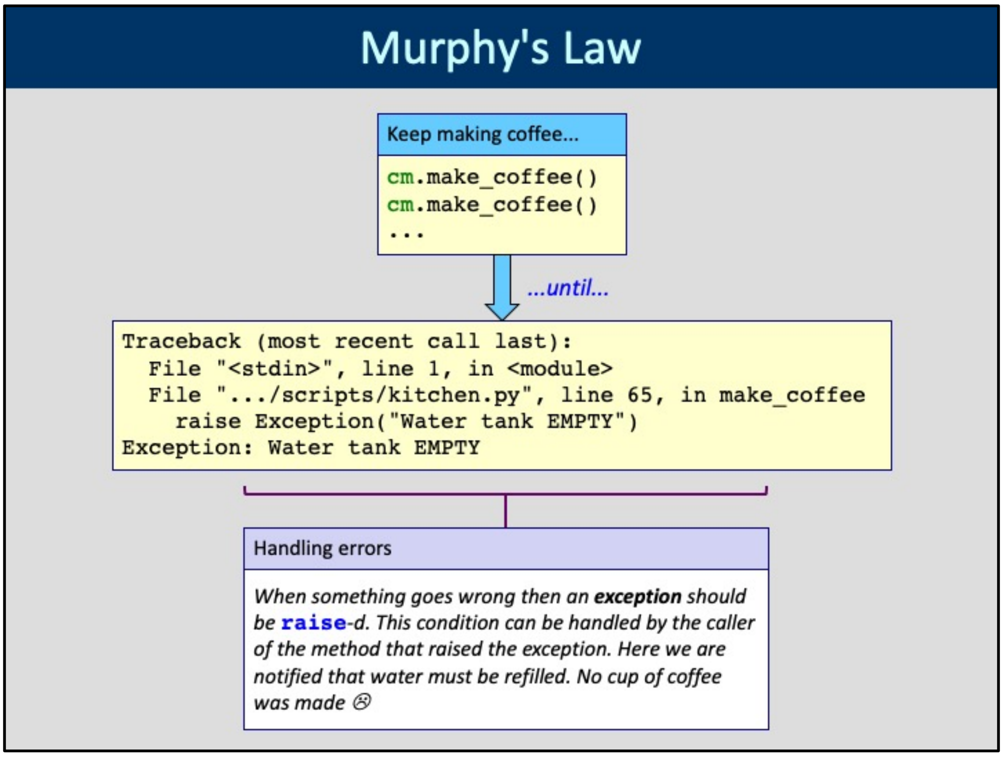
Encapsulation is one of the three most important aspects of object-oriented programming. Python does not enforce it which is considered a weakness by some. As always, there is a compromise between safety and usability. Guido van Rossum decided in favour of ease-of-use. It is possible to "fake" private data members in Python: just prepend an underscore in front of the name. This is not foolproof, though.
The paintings ("La maja vestida", "La maja desnuda") on the slide have been created by Francisco Goya. Because the Naked Maja was considered politically incorrect at the time, Goya was questioned by the Holy Inquisition. Luckily he was not prosecuted as his defense of following an artistic tradition was accepted. The paintings can be admired in the Prado in Madrid.



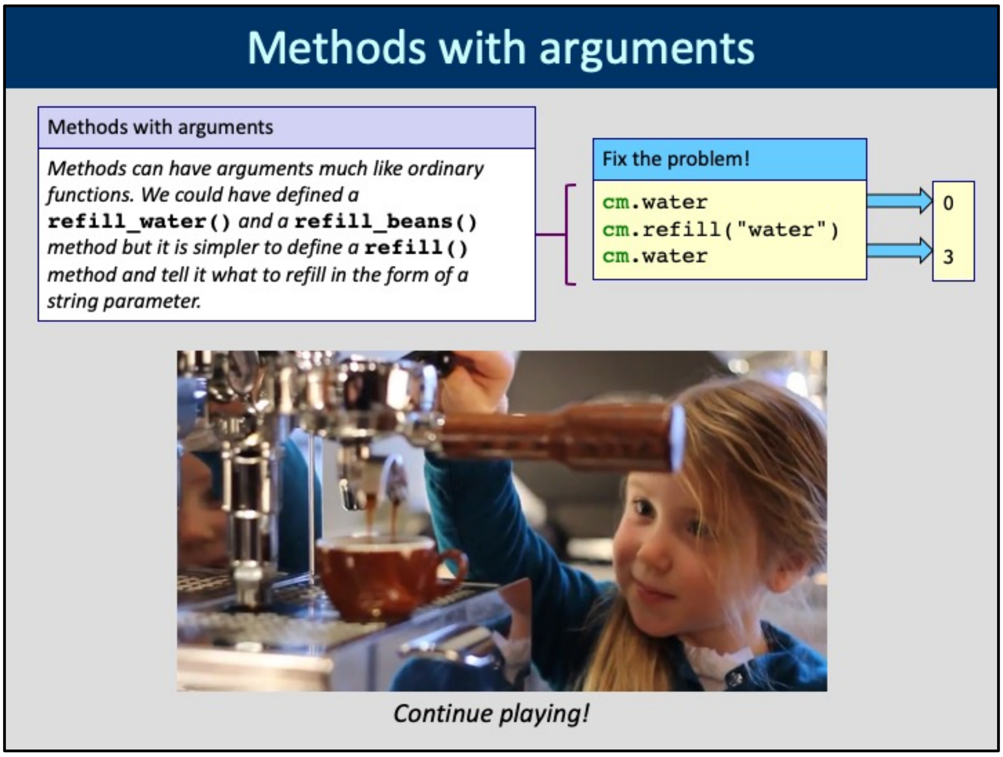
The @property decorator can be used to provide "read-only" access to "hidden" data members. Note, however, that there is no real privacy (encapsulation) in Python. In fact, the __name member in the Person class gets "mangled" to _Person__name and can be accessed as such directly.
Just follow the Python philosophy that "we are all consenting adults here". Putting one or two underscores in front of a class member only signifies an intention that this member is not to be used directly. You can if you want to, it's just not considered good form.



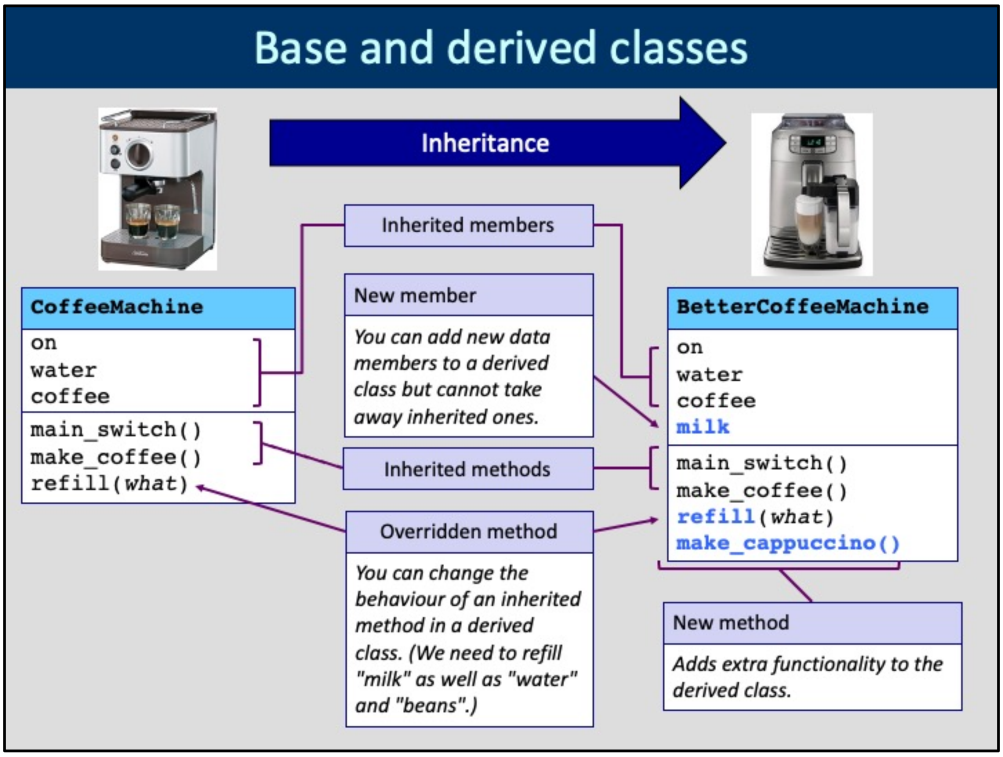
The `main_switch()` method just toggles the value of the `on` data member: if it was False (meaning "the coffee machine is off"), it will be set to True ("machine is on"). The `make_coffee()` method returns a result, just like many ordinary functions do. In addition it reduces the amounts of coffee beans and water by one unit each as "side effects".



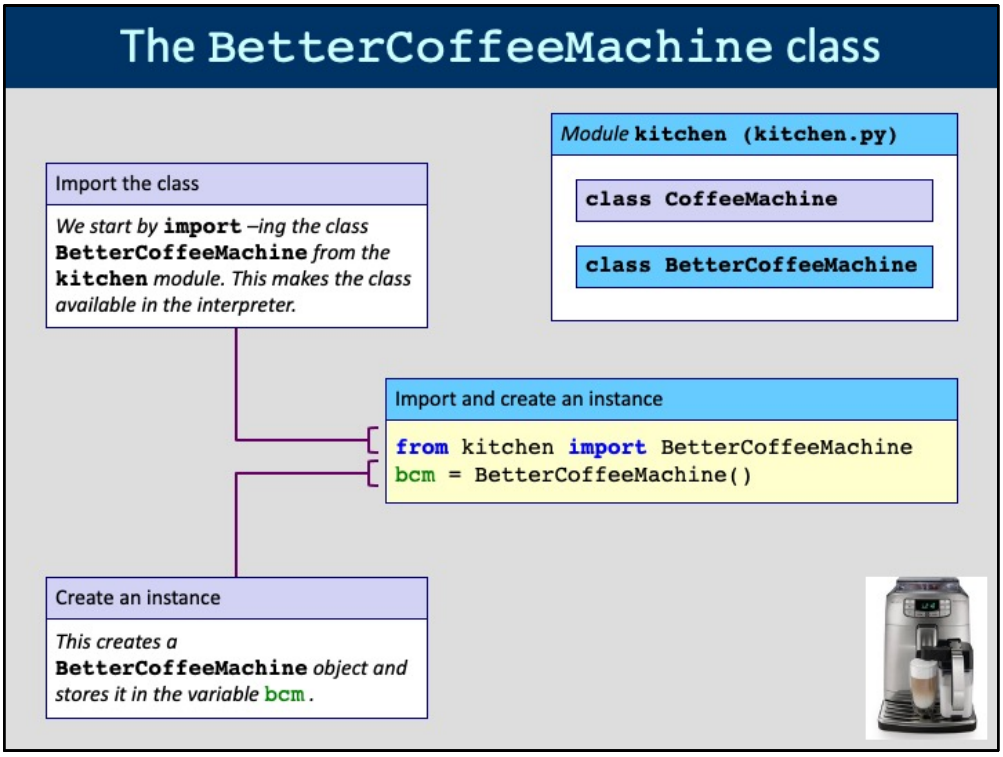
Exceptions are objects that represent errors or other "strange" conditions. When something "bad" occurs, you raise an exception. It is possible to store data in exception objects that describe what happened, most often this is an error message. In our example the problem was that the coffee machine ran out of water. The make_coffee() method raised an exception which is an instance of the standard Exception class and put the error message in it.
Exceptions can be handled in some other location in your code, i.e. they can be analysed and appropriate action can be taken. If you do nothing, the exception is finally handled by the Python interpreter. It prints some traceback information indicating where the problem happened. This is not too nice... We will learn later how to handle exceptions.



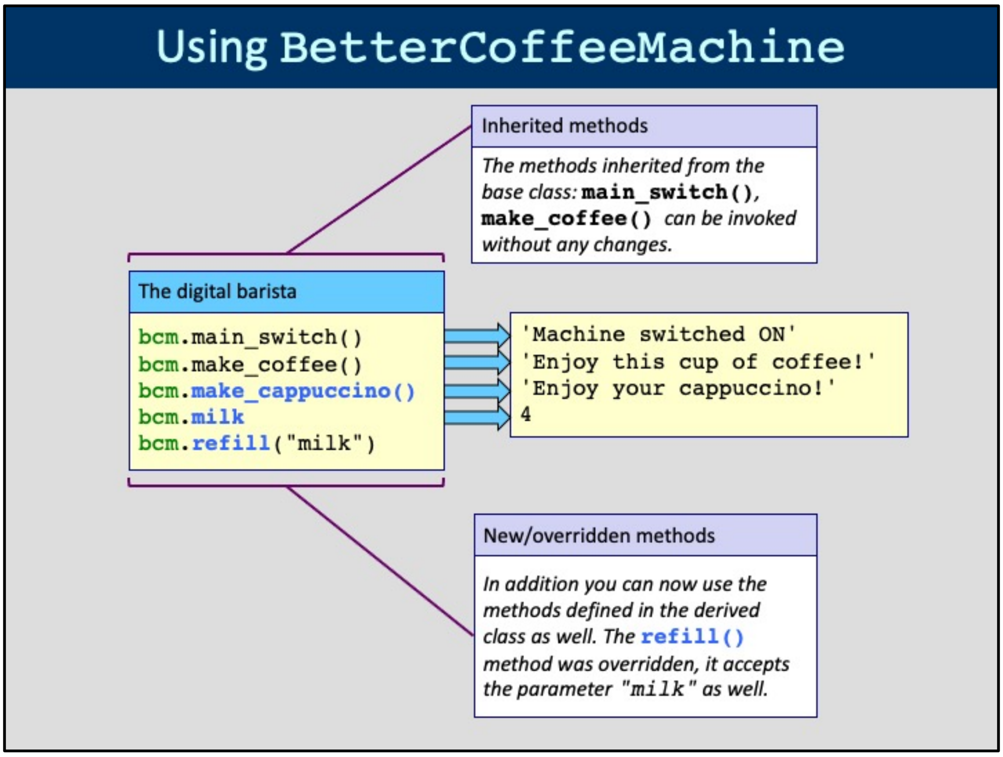
Instead of changing the data members directly, it is more prudent to modify an object's state by passing data into it via "setter functions". In our coffee machine model the `refill()` method is such a "setter function". As we will see later, it knows how much water or coffee it is supposed to fill and thus makes sure the CoffeeMachine object's internal state is always correct.



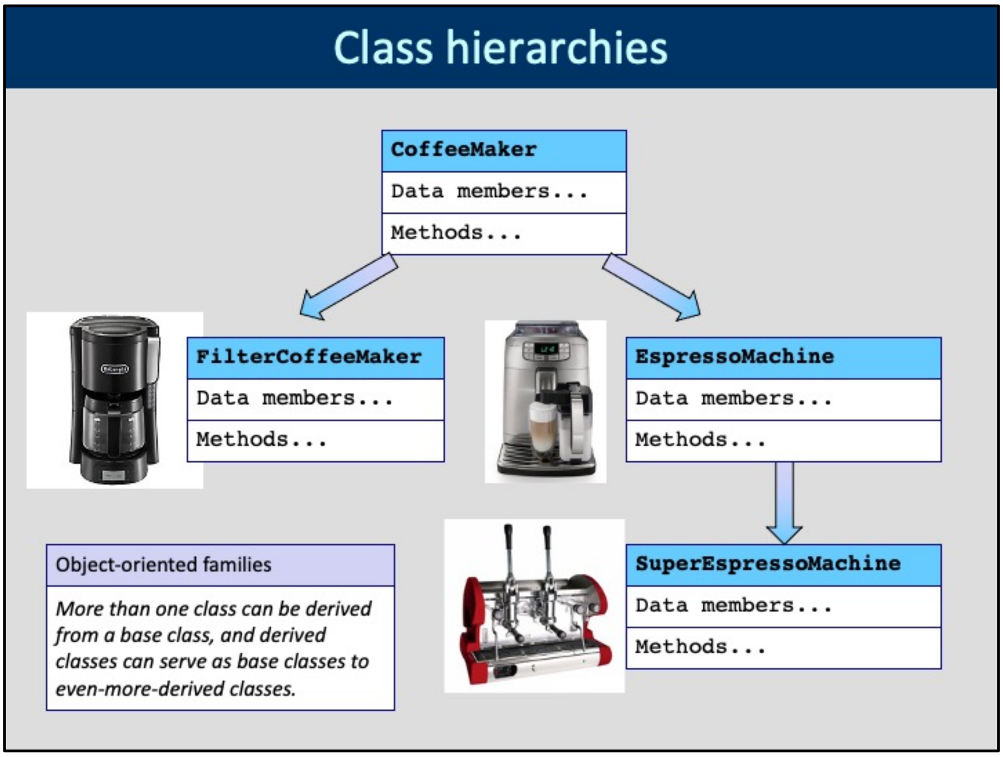
Inheritance plays an important role in object-oriented programming. It supports code re-use, and helps model "is-a" relationships. The BetterCoffeeMachine class can do everything the CoffeeMachine class did: better coffee machines are coffee machines. In addition, the BetterCoffeeMachine can make cappuccino. Generally, derived classes have more data members and methods than their base classes, but this is not mandatory.



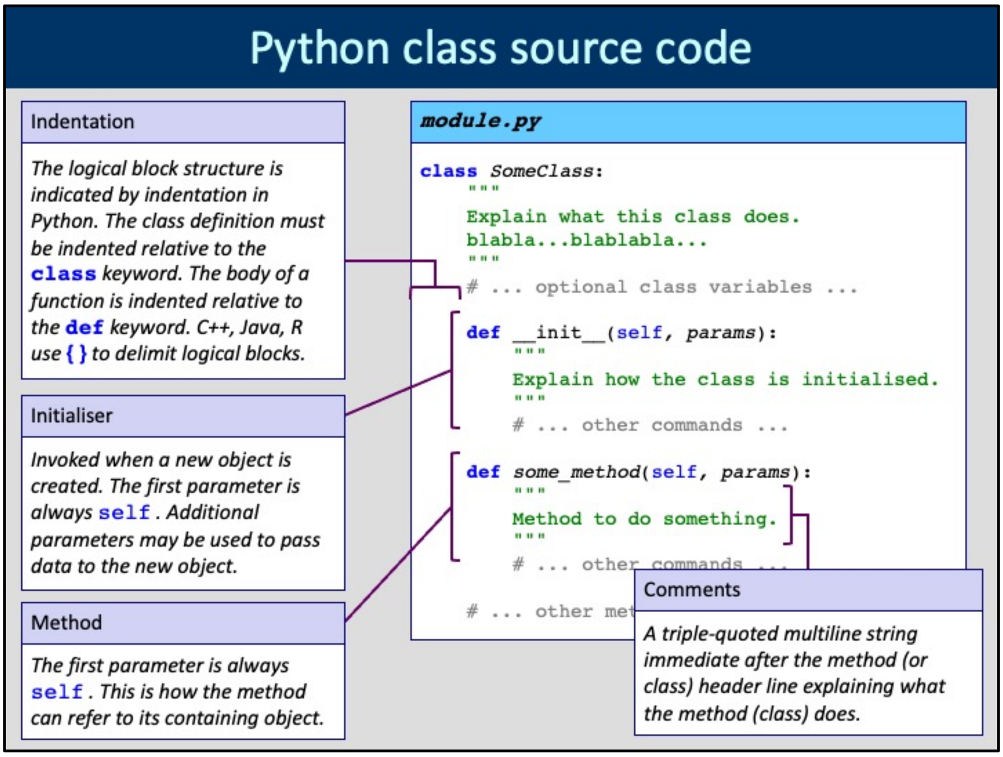
It turns out that the `kitchen` module already contains the `BetterCoffeeMachine` class as well ☺ . We can `import` it exactly as we did with the `CoffeeMachine` class, and then create an instance.



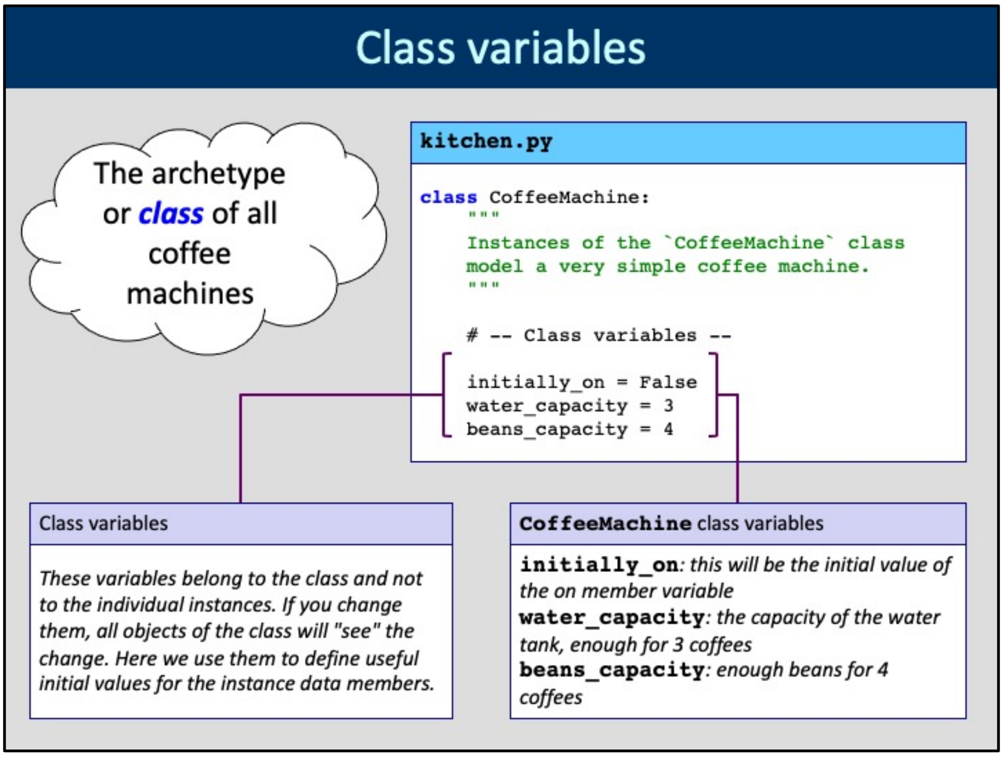
Let's try out the BetterCoffeeMachine to convince ourselves that the methods inherited from CoffeeMachine still work the same way, and that in addition the new and/or overridden methods also work as expected.



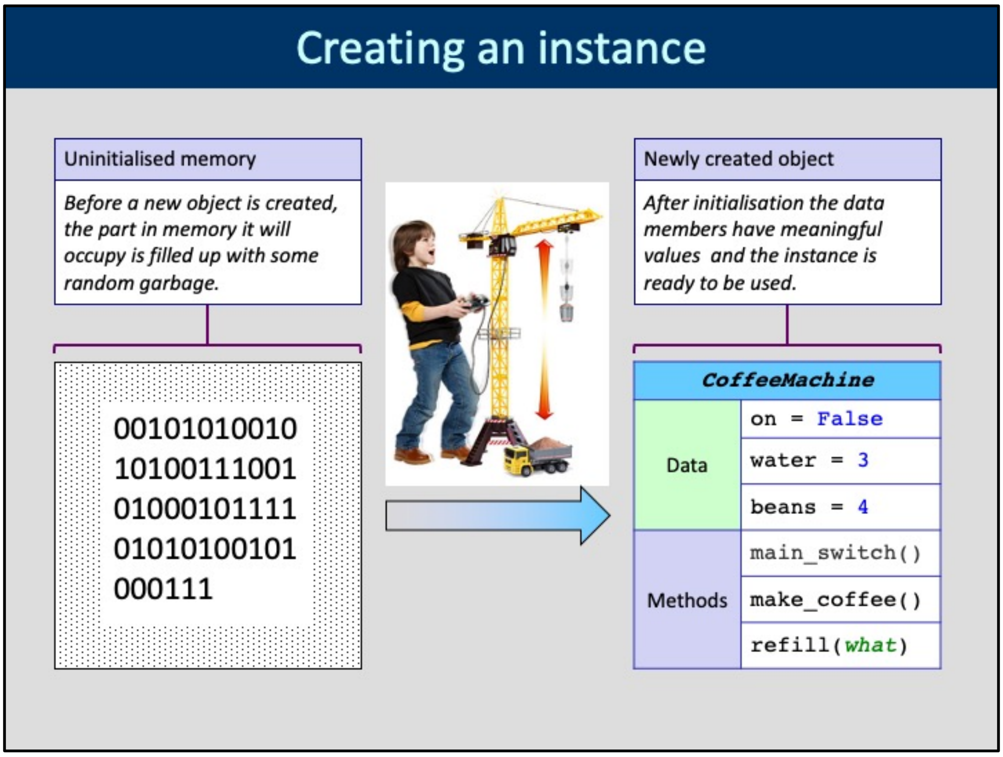
A Python class can inherit from more than one base class. This is an advanced feature that we won't discuss in this introductory course.



This slide shows the general layout of the source code of a Python class. We will analyse a concrete example, the source of the CoffeeMachine class, in the following slides.
In the following I will colour the Python keywords blue. This may not correspond to the syntax colouring you see in your editor.

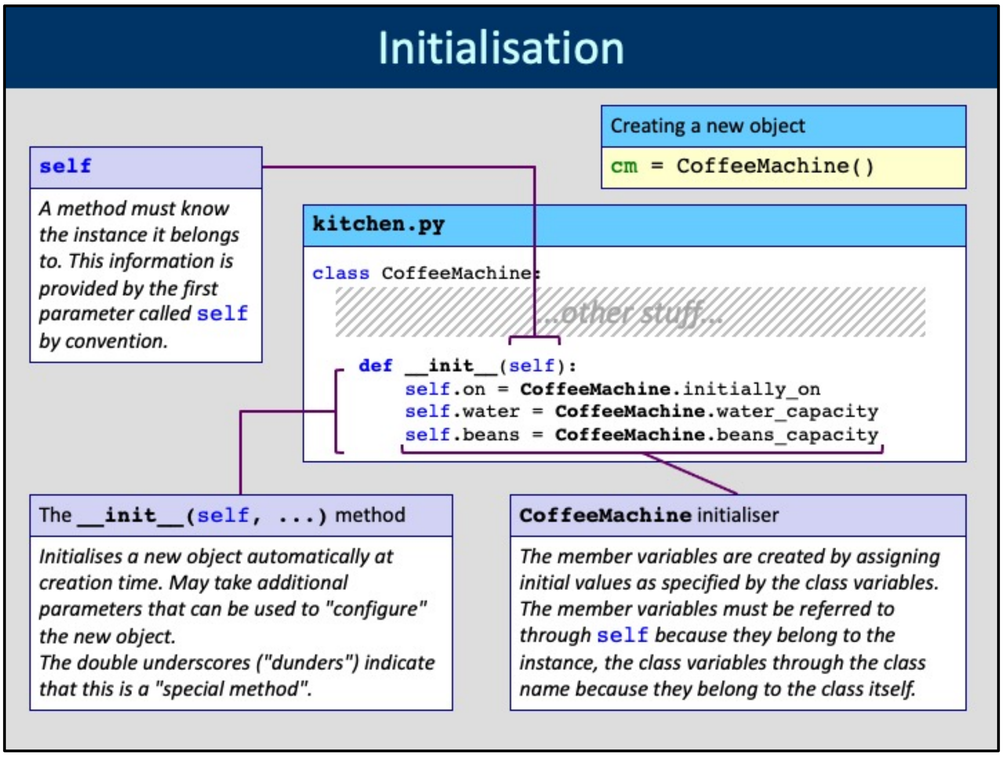


Class variables are most useful when they represent some class-wide constant values. Note, however, that (unlike in C++ for example) there are no "const" variables. Class variables are also public, they can thus be changed "from the outside" which can lead to various strange errors. You must be very careful not to mix up class and instance variables. If you set a class variable through an instance, then automatically an instance variable with the same name will be created, leading to further confusion.

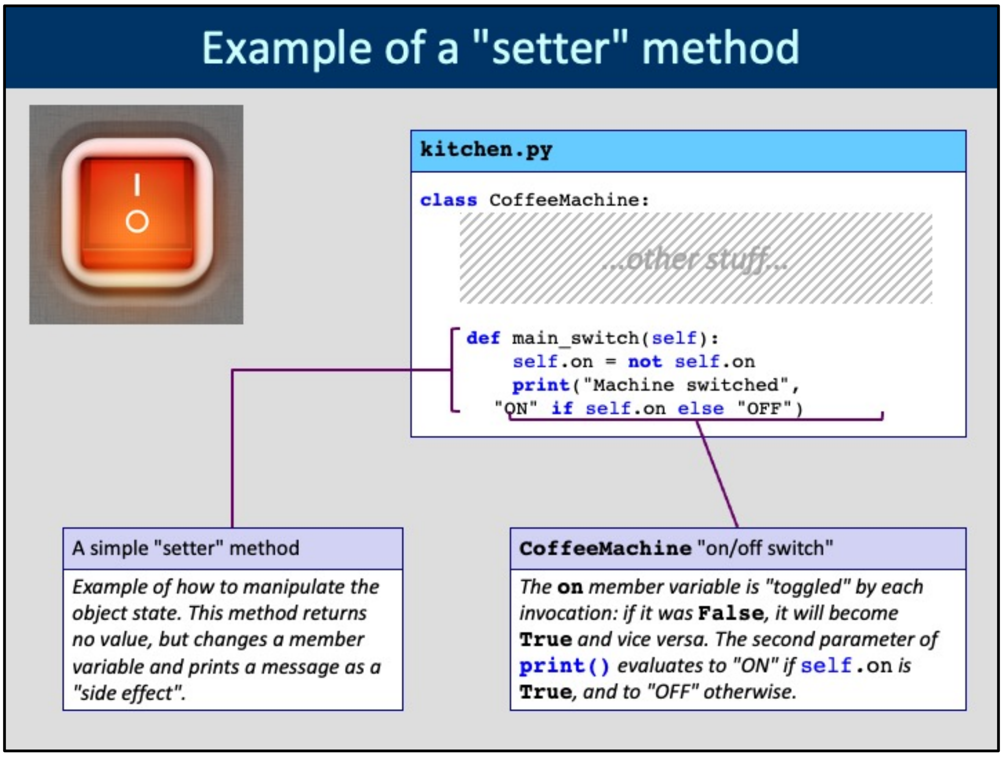


Creating a new object instance is analogous to unboxing a gadget. At the end you have to make sure that the gadget is set up properly.

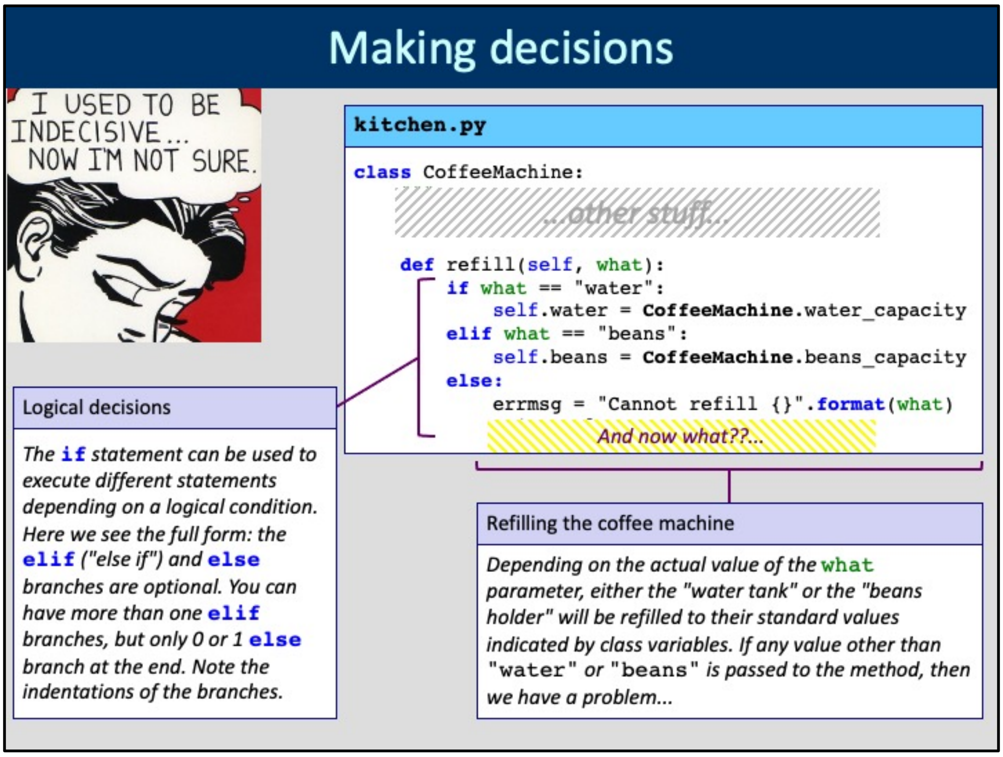
In programming terms, the memory set aside for a new object first contains random bits. We must set all those bits to well-defined values before we can use the new object. This task is performed by special methods called "constructors" in other object-oriented languages such as C++ or Java. In Python, first a method called __new__() is invoked which takes care of basic object construction. Programmers rarely if ever have to deal with this method directly. After __new__(), a second method called __init__() will be invoked. This method is responsible for the proper initialisation of the data members. Apart from very simple cases your classes always must have an __init__() method.



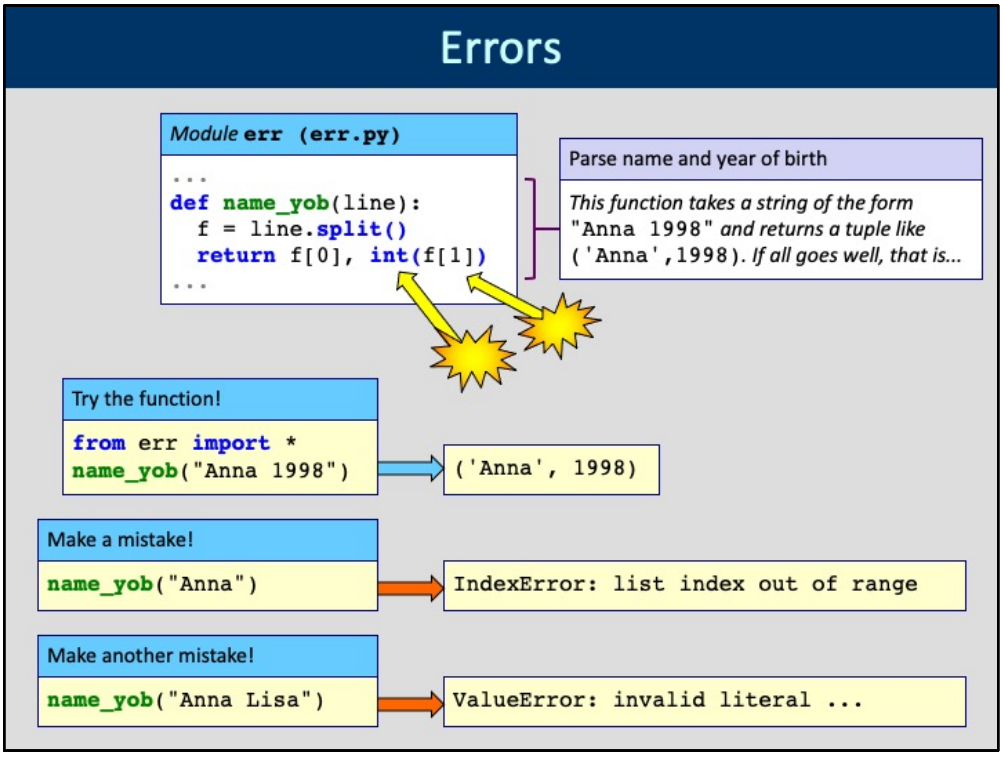
Most non-trivial classes will need some sort of initialisation to make sure the newly created objects are in a well-defined state. Initialisation usually involves setting the member variables (remember, assignment automatically creates a variable in Python!).
If the __init__() method takes additional parameters then it is possible to configure the new object in any way you like. In the example we could have written an __init__() method that fills the water tank only to half its capacity, for instance (although this would not have been terribly useful).



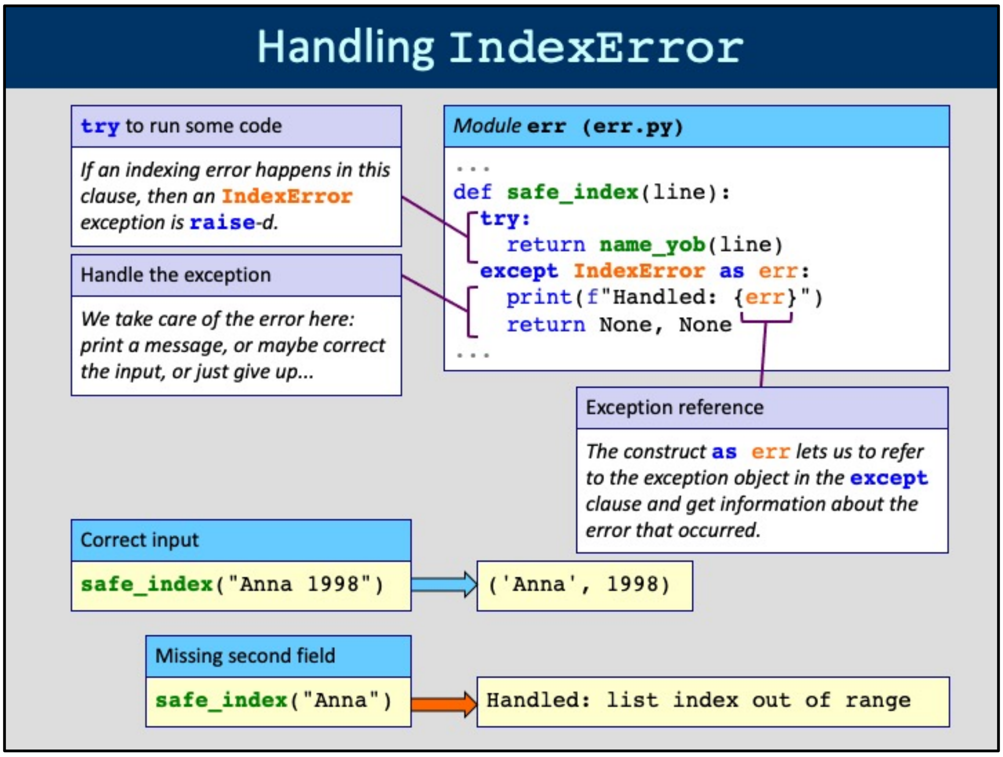
The main_switch() method demonstrates how to write a simple "setter" method in Python. "Setters" manipulate the internal state of the object they belong to. They may take parameters such as the new value of an internal data member, and they may return a result, e.g. the old value of the variable. In our simple example none of this is necessary.



Variants of the if ... else construct exist in most programming languages. The idea is always the same: if the condition is true, then execute the first branch, otherwise execute the branch after "else". The "else branch" is optional, if omitted and the condition is false, then execution continues directly after the if.
More complicated decision paths can be encoded by using "elif" branches. "elif" is the shorthand for "else if", and is also followed by an expression that evaluates to a Boolean value. In the refill() method above, IF the parameter `what` had the value "water", then the water tank is filled up, otherwise IF the parameter `what` was equal to "beans" then the coffee beans are replenished. If neither condition is true, then the `else` branch should be executed. We construct an error message, but then... what shall we do?



I wrote a simple function `name_yob()` that takes a string and parses it into a tuple consisting of a string (a person's name) and an integer (the person's year of birth). The function lives in the "err" module, together with some other functions (see following slides). When you invoke `name_yob()` with a string argument that corresponds to the specification then it works. However, I was lazy and did not add any error handling. If you invoke the function with a name only, or if the birth year cannot be converted to an integer then we get errors.

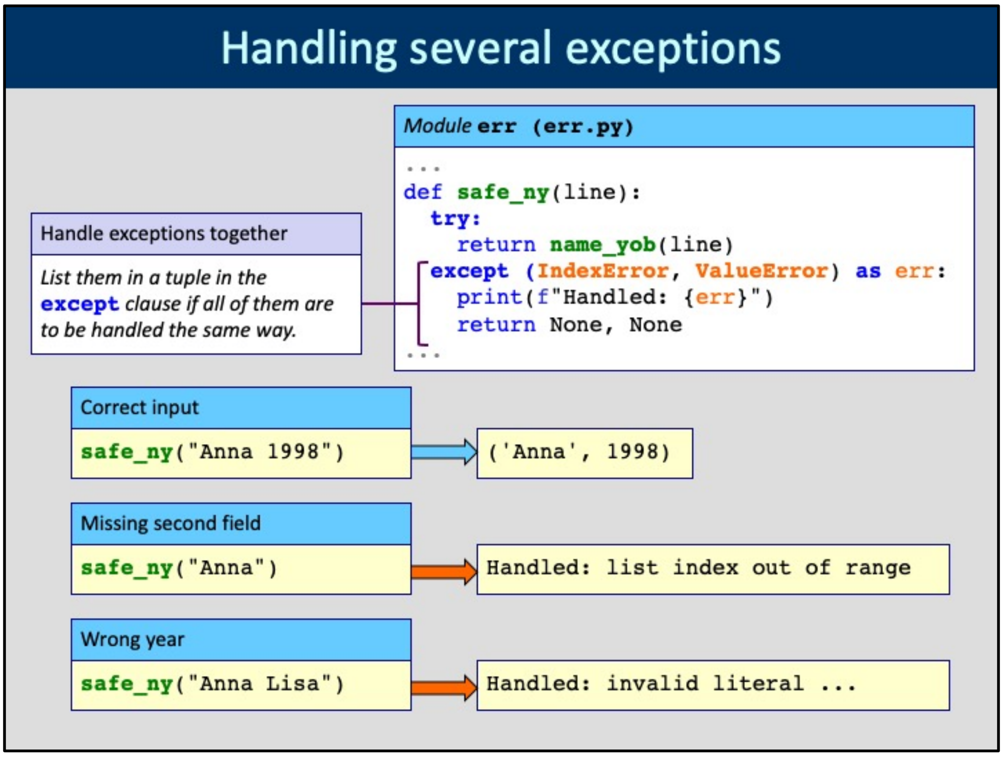


Python uses an error-handling mechanism called "exceptions". The idea is that when an error (or some other exceptional condition) occurs, then this is signalled by "raising an exception" (like raising a red flag). Exceptions are objects that usually carry information in their data members about the problem.

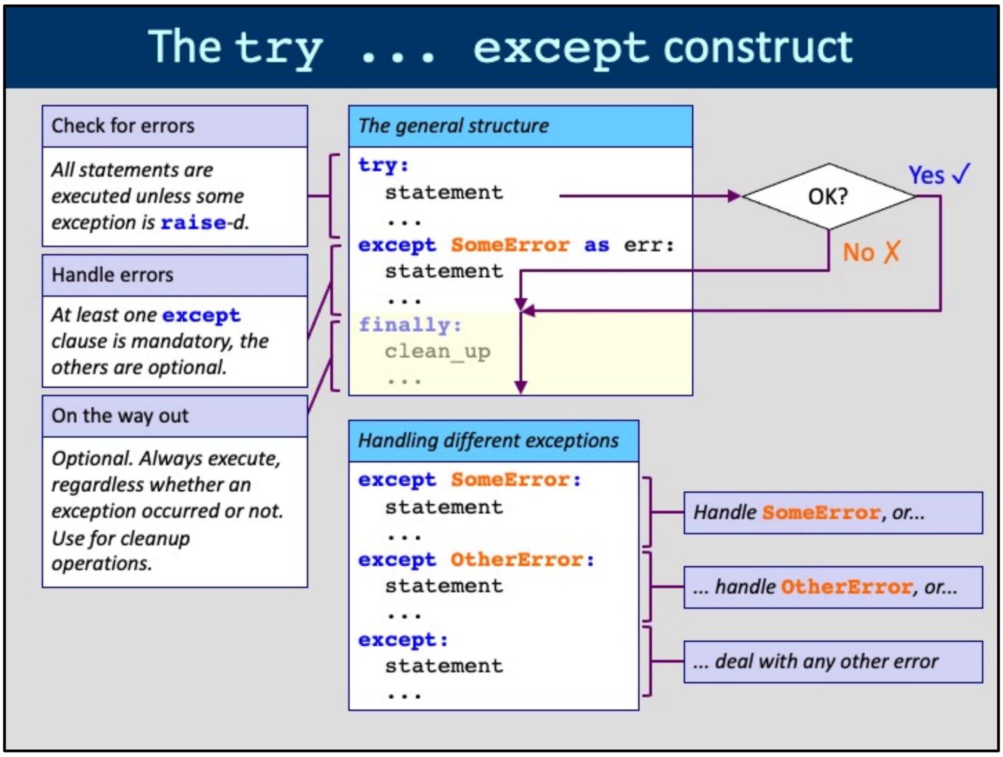
Statements that may raise exceptions are wrapped in a "try-block". If everything goes well, then the code in the try-block just runs. However, if an exception is `raise`-d somewhere inside, then Python looks for an `except` clause for that kind of exception. If a matching `except` clause is found, then its body is executed. Here you can take care of the error: you may just print some message, write to a log file, or even try and correct the situation somehow.

If there is no `except` clause for the exception, then the Python interpreter will handle it in a rather drastic manner: execution stops and a long "stack trace" is printed, which is embarrassing. Well-written scripts handle all possible exceptions on their own.

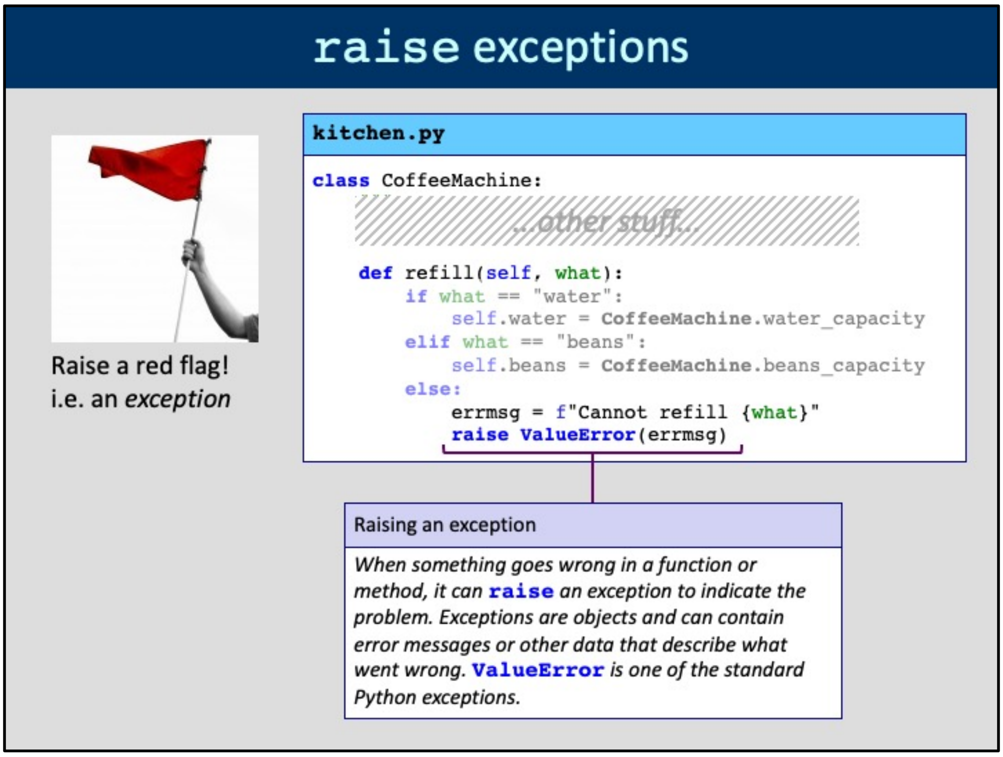
You can try exception handling with the `safe_index()` function, also from my "err" module.



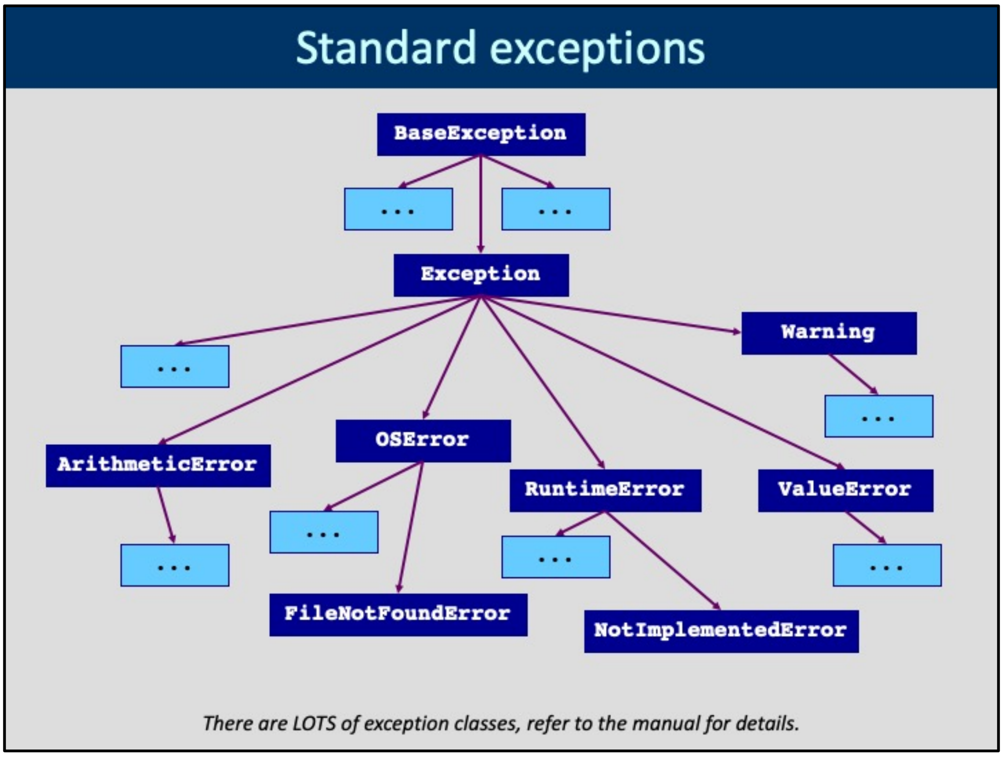
If several kinds of exceptions are to be handled the same way, then you can group them all in a tuple in the `except` clause. Otherwise you can have several `except` clauses in a `try` block.
This is implemented in the `safe_ny()` function from the "err" module for you.



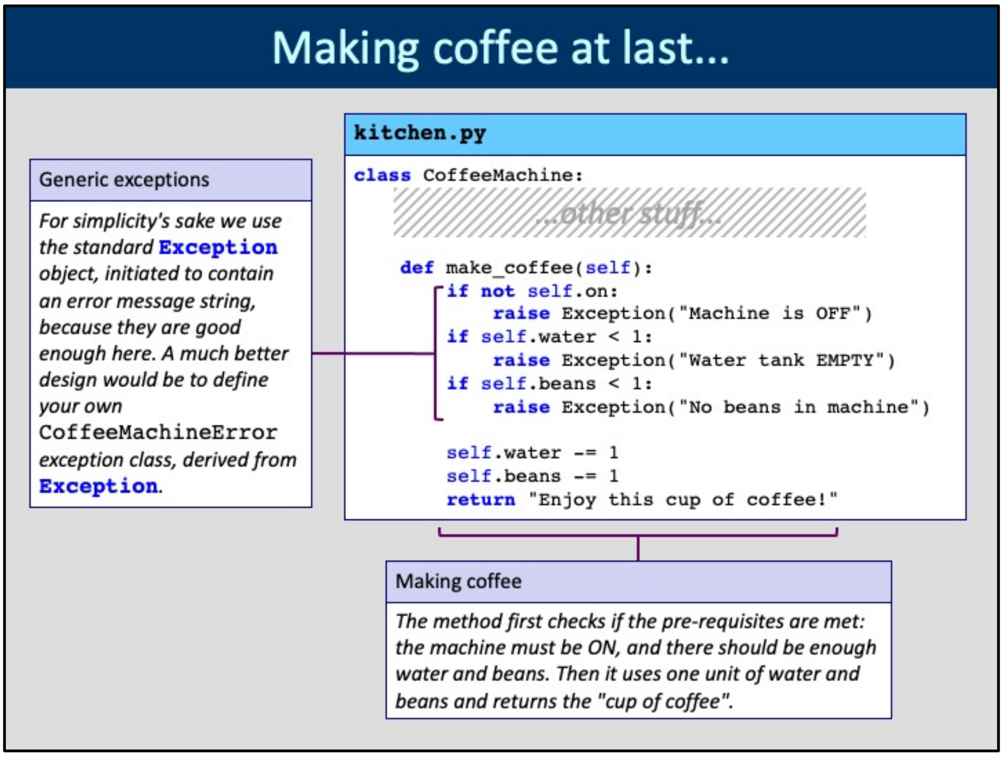
This slide does not discuss all the fine details of the try...except language construct (there are many). The essence: the statements in the `try` block are executed. If no exceptions are raised, execution continues after the end of the `try` block. If an exception is raised, then Python looks at the `except` clauses following the `try` block. If an `except` clause is found that catches the correct exception class (remember, exceptions are objects!), then the `except` code block is executed.
If there is an optional `finally` block, then its statements are executed "on the way out", irrespective of whether an exception has been handled or not. This is useful for "clean-up" operations such as closing files etc.
You may have more than one `except` block, each of them handles a different kind of exception. Only one of them is executed, so it's a good idea to order them from the most specific to the most general. Which is an `except:` clause that does not specify any exception type: this handles "everything".
`try` blocks may have an `else` clause after the exceptions, this is executed if no exceptions have occurred. I have not seen any important use case for this feature though.



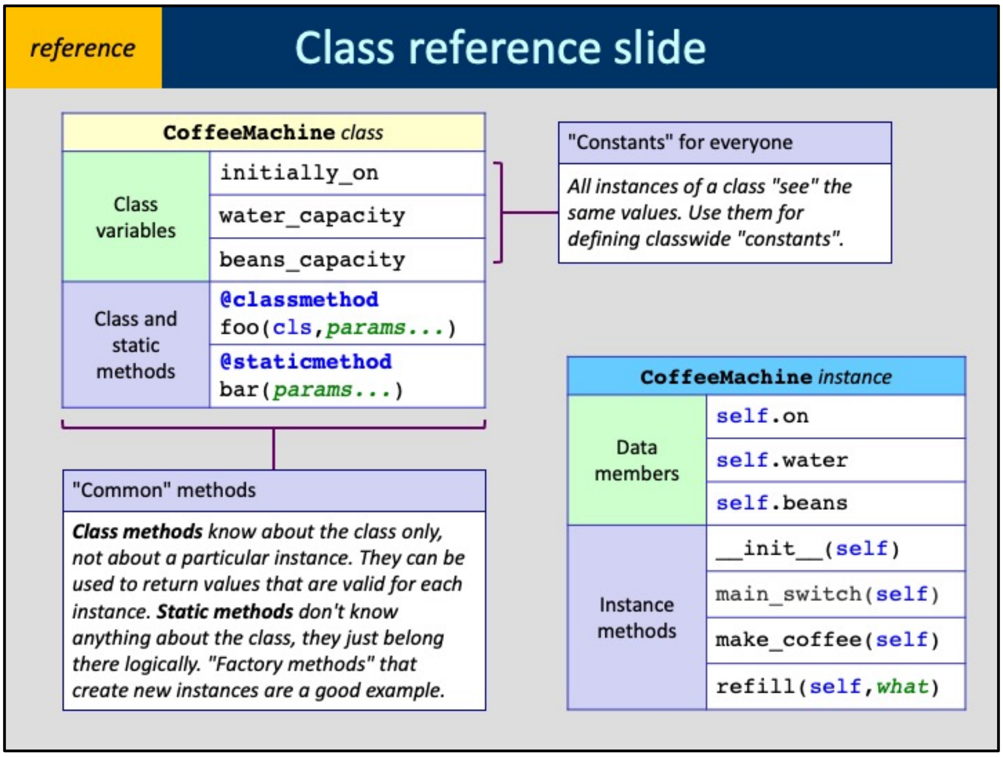
This is how you can signal to Python that something bad happened: you raise an exception. The code using the `refill()` method of `CoffeeMachine` shall wrap the invocation in a `try` block. The block does not have to surround the method call directly, because exceptions "propagate" through code blocks. That is, it's perfectly sufficient to invoke a function in a `try` block that invokes a function that invokes another function that may raise an exception.



1) Python gives you lots of standard exception classes that you can use in appropriate situations, e.g. you can raise a NotImplementedError exception if you want to indicate that a certain feature has not been implemented yet. All these exception classes are derived from BaseException. You can look them up in the online documentation.
2) You can create your own exceptions by inheriting from Exception or one of its subclasses. This is useful if you want to store specific information about the condition that caused the exception.
3) Note that this exception hierarchy is somewhat controversial, but that's what we have...



Which exception to raise? The programmer has a wide choice. Sometimes the standard exception classes are appropriate: for instance, raise a TypeError in a function that expects a list parameter and got a dictionary instead, or raise a ValueError when a "wrong value" (e.g. negative number for cell counts) was passed. Larger packages define their own exceptions that usually derive from Exception. We could have done that, too, but that's too much for a simple class like CoffeeMachine. In the end the class raises only standard Exception-s, passing an error message to its initialiser.



Most of the time we use instance data members and instance methods. We have seen that class variables (that belong to the class itself, not to any particular instance) can be useful to define classwide "constants". (There are no real constants in Python, you can change the value of any variable.)

We have not used class methods and static methods. These are declared using the @classmethod and @staticmethod decorators, respectively. The first parameter of a class method is "cls", not "self": it knows about the whole class as such, but does not belong to any individual instance. Class methods can be used to return values that are equally valid for each instance of the class and don't depend on the instance data members. Static methods are only loosely associated with the class. We will use later a static method to create objects of a class: this pattern is called "factory method".

How to write a derived class

CoffeeMachine *class*

Base class / superclass

BetterCoffeeMachine *class*

Derived class / subclass

Deriving a class

Simply write the name of the base class in parentheses after the name of the derived class in the class definition line.

kitchen.py

```
class CoffeeMachine:
    """
    Implements a basic coffee machine
    """
    # ... rest of the class

class BetterCoffeeMachine(CoffeeMachine):
    """
    Implements a better coffee machine
    """
    # ... rest of the class
```

Derived class variables

kitchen.py

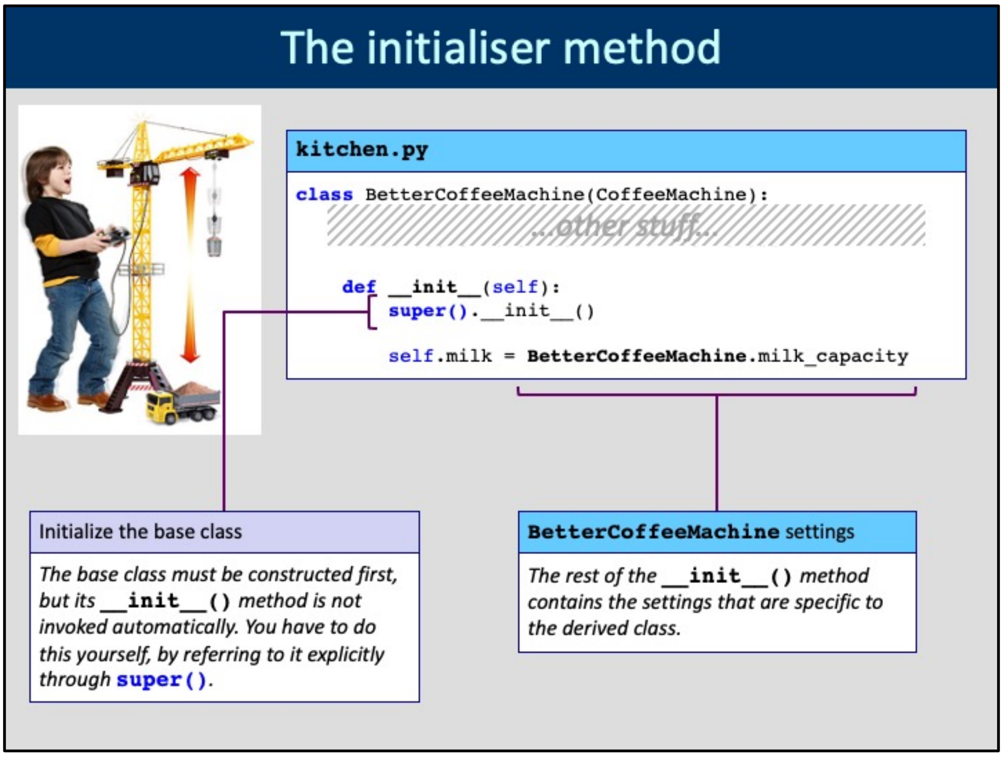
```
class BetterCoffeeMachine(CoffeeMachine):
    """
    A better machine can also make cappuccino.
    """

    # -- Class variables --

    milk_capacity = 5
```

What you cannot see here

The class variables **`initially_on`***,* **`water_capacity`** *and* **`beans_capacity`** *inherited from the base class are still available in the derived class. You do not need to define them again.*

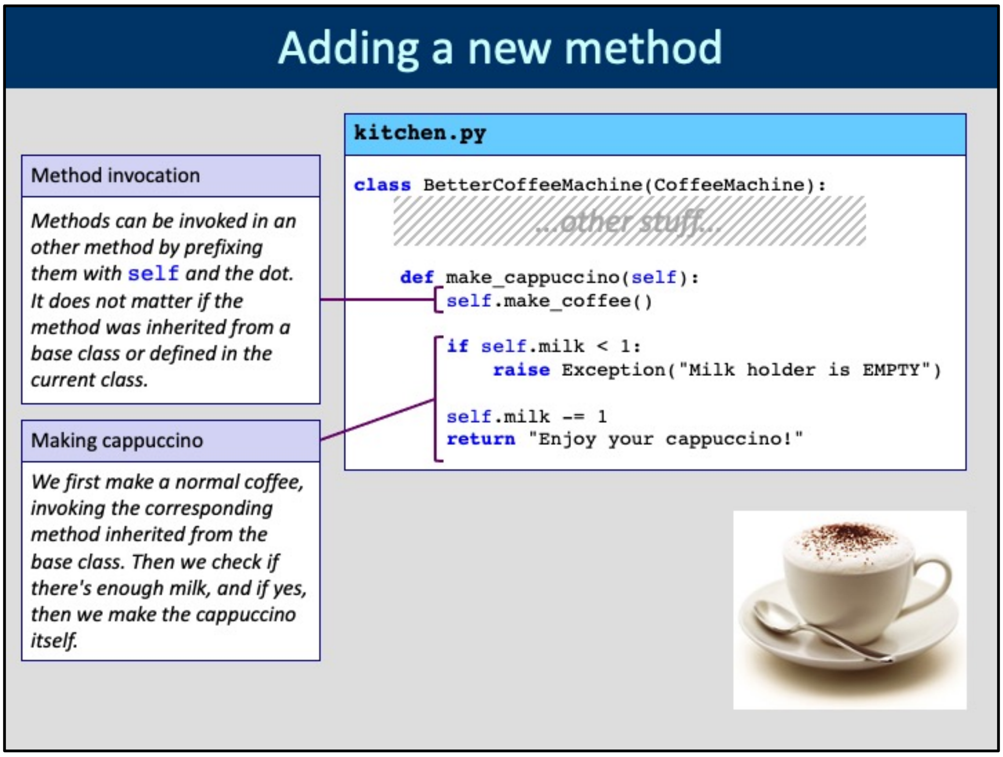
`BetterCoffeeMachine` **class variables**

`milk_capacity`*: enough milk for 5 cappuccinos. This class variable had to be added here because it is specific for the derived class.*

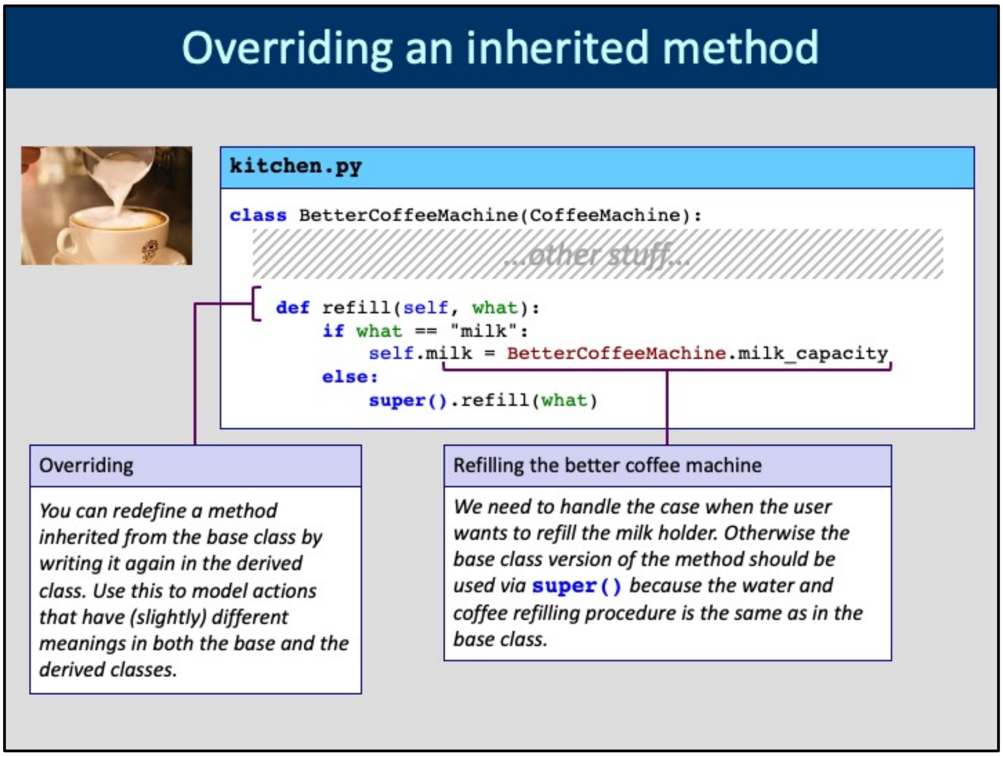


Derived classes can refer to their base classes using super() . In the initialisation example the __init__() method of the base class is invoked through super() which refers to the base class object. This is why the self parameter does not need to be passed.

You may see old-style code from Python 2.x days that uses something similar to CoffeeMachine.__init__(self) instead. This still works in Python 3 but you should use the super().__init__() idiom instead.



Here you can see a nice example of code reuse. We make cappuccino by making coffee first, and our CoffeeMachine class already knows how to do that. In the BetterCoffeeMachine derived class we just invoke the inherited `make_coffee()` method and then add the necessary ingredients.



You may override a method in a derived class only if its "signature" (its name and its list of parameters) is the same as in the base class. The version in the derived class may refer to the base class version via `super()`.