PV248 Python

Petr Ročkai

Part 1: Object Model

In this lecture, we will look at the role and the semantics of objects. In particular, we will look at the basic structure of a Python program and note that pretty much everything is an object.

Objects

3

- the basic 'unit' of OOP
- also known as 'instances'
- they bundle data and behaviour
- provide encapsulation
- local (object) invariants
- make code re-use easier

In object-oriented programming, an object is the basic building block of a program. Please note that in some contexts, objects are instead called instances, or class instances. We will use both terms interchangeably.
The traditional notion of an object holds that it is an entity which bundles data and behaviour. This view works quite well in Python. However, for the time being, let's forget the standard story about how the data is enclosed in a wall made of methods. This does not work in Python anyway, since attributes (data) are public.
The real benefit is that holding data in discrete packages, accompanied by the relevant chunks of functionality, allows us to think in terms of local invariants. A typical object wraps up a chunk of state, but of course not all combinations of values in its attributes need to be valid (and rarely are, for non-trivial objects). The constraints which govern valid states are what we call (local) invariants.
The methods (behaviour) are then written in such a way that they move the object from valid states to other valid states: in other words, they preserve invariants. In a correctly designed API, it should be impossible to get an object into an invalid state by only interacting with it through its public interface.
Finally, objects make it simpler to re-use code, through polymorphism, composition and inheritance. This was, in fact, one of the original motivations for OOP.

Classes

4

- each (Python) object belongs to a class
- templates for objects
- calling a class creates an instance
  - `my_foo = Foo()`
- classes themselves are also objects

Like with plain old data, where values are classified into data types, it is often advantageous to classify objects (which are a kind of value) into classes (which are a kind of data type).
Like any other data type, a class prescribes the structure of the data (attributes) and operations on its values (methods). The distinct feature of class-based object-oriented languages is the ease with which we can create new data types with complex, user-defined operations.
Aside: in Python-speak, since double underscores are so common, they have a special name: 'dunder'. We will call them that in this subject, since we will encounter quite a few of them.
Syntactically, classes in Python behave like functions, which create new instances of the class when called. In addition to creating a new empty object, it will also immediately call the `__init__` method of that object, forwarding any arguments passed to the original call.
Finally, and this is specific to Python, classes are themselves objects, and it is possible to interact with them at runtime. We will get back to this idea shortly.

Types vs Objects

5

- class system is a type system
- since Python 3, types are classes
- everything is dynamic in Python
  - variables are not type-constrained

Since classes are essentially data types, a class system is essentially a type system. In fact, all data types in Python are actually classes. Each object dynamically belongs to a class: in Python, variables do not carry types (i.e. each variable in Python can hold a value of any type). However, values (objects) do, and it is impossible to accidentally coerce an object into a different class – we say that Python has a strong dynamic type system.

Poking at Classes

6

- you can pass classes as function parameters
- you can create classes at runtime
- and interact with existing classes:
  - `{}.__class__`, `(0).__class__`
  - `{}.__class__.__class__`
  - compare `type(0)`, etc.
  - `n = numbers.Number(); n.__class__`

Of course, since this is Python, classes also exist at runtime: and as we will eventually learn, everything in Python that exists at runtime is an object. Hence, classes are also objects with attributes and methods and it is possible to interact with them like with any other object. This is rarely useful, but it does serve to illustrate a point about Python.
To demonstrate that classes really are objects, the slide gives a few examples of valid Python expressions. Try them yourself and think about what they mean.

Encapsulation

7

- objects hide implementation details
- classic types structure data
  - objects also structure behaviour
- facilitates loose coupling

While strictly speaking, there is no encapsulation in Python (hiding of data is not enforced), it is commonly approximated: attributes can be named with a single leading underscore to signify they are 'private' and directly accessing those outside of the class itself is frowned upon. But there is no enforcement mechanism.
Encapsulation is really just a way to structure your code to make local invariants (as mentioned earlier) easier to reason about. All code that could possibly violate a local constraint is bundled up within a class. If some outside code changes the data (attributes) directly and breaks the invariant, it is going to get exactly what it asked for.

The partitioning of program state that is achieved through encapsulation is also known as loose coupling. This is an important property of good programs.

Loose Coupling

8

- coupling is a degree of interdependence
- more coupling makes things harder to change
  - it also makes reasoning harder
- good programs are loosely coupled
- cf. modularity, composability

Coupling is a measure of how tangled the program is – how many other things break when you change one thing, and how far away the damage spreads out from the change. Consider a ticket booking system that started out as a small local thing, with only a few venues. But the system grew and now you are adding venues from another city – but for that, you need to change how venues are stored, since you do not have a 'city' field (they were all local).
In a loosely coupled system, you add the attribute (or a database column) and that's pretty much it: you may need to adjust a few methods of the venue class, but outside of the class, nothing needs to change.
In a tightly coupled system, on the other hand, it could easily turn out that now the reservation emails are completely jumbled. This might sound like a stretch, but consider a system which stores all the attributes in a list, and each component simply hard-codes the meaning of each index.
So `reservation[4]` is the venue name, and `reservation[7]` is the customer's family name. But you add the city, say as index 5, and now all other fields have shifted, and your entire application is completely broken and you are sending out mails that start 'Dear Britney Spears'. Of course, this is terrible programming and the example is slightly artificial, but things like this do happen.

Polymorphism

9

- objects are (at least in Python) polymorphic
- different implementation, same interface
  - only the interface matters for composition
- facilitates genericity and code re-use
- cf. 'duck typing'

One of the principles that helps to achieve loosely coupled programs is polymorphism, where different objects can fill the same role. This encourages loose coupling, because polymorphic objects separate the interface from the implementation quite strictly. If a function must be able to work with objects of different types, it can only do so through a fairly well-defined interface. The same principle applies to object composition.
One area where polymorphism is very important is generic functions: consider `len`, which is a builtin Python function that gives you a number of elements in a collection. It works equally well for lists, sets, dictionaries and so on. Or consider the `for` loop, that likewise works for multiple collection types. Or the addition operator: you can add integers, floating point numbers, but also strings or lists. Those are all different manifestations of polymorphism.
We have already mentioned that Python is a very dynamic language (and this will come up many more times in the future). Where polymorphism is concerned, this dynamic nature translates into duck typing: types are not rigorous entities like in other languages. This essentially says, that for type-checking purposes, if it quacks like a duck and walks like a duck, it may as well be a duck. Hence, a function that expects a duck will accept anything that has sufficiently duck-like behaviour. In particular, it needs to provide the right methods with the right signatures.

Generic Programming

10

- code re-use often saves time
  - not just coding but also debugging
  - re-usable code often couples loosely
- but not everything that can be re-used should be
  - code can be too generic
  - and too hard to read

Programming the same thing over and over again for slightly altered circumstances gets boring very quickly. And it is not a very efficient use of time. A lot of programming boils down to identifying such repetitive patterns and isolating the common parts. Many of the constructs in programming languages exist to allow exactly this:

- loops let us repeat statements over multiple values,
- functions let us execute a set of statements in different contexts (and with diferent values),
- conditionals allow us to write a sequence of statements that is almost the same in different circumstances,
- user-defined data types (records, classes) allow us to give structure to our data once and use it in many places.

Generic programming is a technique which mainly gives additional power to functions and objects. Under normal circumstances, functions are parametrized by values which are all of the same type. That is, functions take arguments, which have fixed types, but the values that are passed to the function vary from invocation to invocation.
With generic programming, the types can also vary. In Python, generic programming is so pervasive that you may fail to even notice it. You can often pass sets or lists into the same function, without even thinking about it. Or integers vs floating-point numbers. And so on. However, this is not a given. On the other hand, this level of flexibility is also part of the reason why Python programs execute slowly.

Attributes

11

- data members of objects
- each instance gets its own copy
  - like variables scoped to object lifetime
- they get names and values

Objects can have attributes (and in fact, most objects do). Attributes form the data associated with an object (as opposed to methods, which form the behaviours).
Attributes behave like variables, but instead of a running function, they are associated with an object. Each instance gets their own copy of attributes, just like each instance of a running function gets its own copy of local variables.
Also like variables, attributes have names and those names are bound to values (within each object instance). Alternatively, you can think of attributes like key-value pairs in a dictionary data structure. Incidentally, this is how most objects are actually implemented in Python.

Methods

12

- functions (procedures) tied to objects
- implement the behaviour of the object
- they can access the object (`self`)
- their signatures (usually) provide the interface
- methods are also objects

Methods are basically just functions, with two differences. The first is semantic: they are associated with the object and are allowed to access and change its internal state. In Python, this is essentially by custom – any function can, in principle, change the internal state of objects. However, in other languages, this is often not the case.
The second is syntactic: the first argument of a method is the object on which it should operate, but when the method is called, this argument is not passed in explicitly. Instead, we write `instance.method()` and this 'dot syntax' does two things:

1. First, it looks up the method to call: different classes (and hence, different objects) can provide different methods under the same name. In this sense, objects and classes work as a namespace.
2. It passes the object on the left side of the dot to the method as its first parameter.

The list of methods, along with the number and (implied) types of their arguments forms the interface of the object. In Python, it is also fairly normal that some attributes are part of the interface. However, this is not as bad as it sounds, because Python can transparently map attribute access (both read and write) to method calls, using the `@property` decorator. This means that exposing attributes is not necessarily a violation of encapsulation and loose coupling. We will talk about this ability in a later lecture.
Finally, as we have mentioned earlier, everything that exists at runtime is an object: methods exist at runtime, and methods are also objects (just like regular functions).

Class and Instance Methods

13

- methods are usually tied to instances
- recall that classes are also objects
- class methods work on the class (`cls`)
- static methods are just namespaced functions
- decorators `@classmethod`, `@staticmethod`

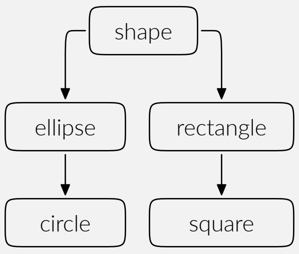
Besides 'standard' methods, which operate on objects (instances), in Python there are two other method types: static, which are really just plain functions hidden inside a class, and class methods which treat the class as an object. They are declared using decorators, like this:

```
@staticmethod
def a_static_method(x, y):
    pass
```

Design-wise, static and class methods are of minor significance.

Inheritance

14



- `class Ellipse( Shape ): ...`
- usually encodes an is-a relationship

In object oriented programs, inheritance plays two roles. The first is organizational: classes are arranged into a hierarchy where each subclass is a more specific version of its superclass. Instances of a subclass are automatically also instances of the superclass. We say that a circle is an ellipse which is a shape. Any interfaces and behaviours common to all shapes are also present on ellipses, circles, rectangles and so on.
The second role is code re-use. While the organizational role is undisputed, using inheritance for code re-use is somewhat controversial. This is because it often, though not always, goes against the former, which is seen as more important.
In the specific picture, maximizing code re-use through inheritance would invert the hierarchy: circle has a single diameter, while an ellipse has two semi-axes. Likewise, a square has one side length, but rectangle has two. The attributes of an ellipse are a superset of the attributes of a circle, and the same with square vs rectangle. Hence, re-use would push us to derive an ellipse from a circle and a rectangle from a square. But not every rectangle is a square – we have saved a bit of code, but ruined the hierarchy.

Multiple Inheritance

15

- more than one base class is possible
- many languages restrict this
- Python allows general M-I
  - `class Bat( Mammal, Winged ): pass`
- 'true' M-I is somewhat rare
  - typical use cases: mixins and interfaces

In both cases (inheritance as an organisational principle, as well as inheritance for code re-use), it sometimes makes sense for a given class to have more than one superclass. In general multiple inheritance, this is allowed without restrictions: a bat inherits both traits of mammals (e.g. that females may be pregnant) and of winged creatures (say, a wingspan). Class hierarchies with this type of multiple inheritance are rare in practice, since they become hard to reason about quite quickly.
There are two types of restricted multiple inheritance: one is mixins, which exist for code re-use, but do not participate in the is-a relationship. The second is interfaces, which is the exact opposite: they provide organization, but no code. In Python, interfaces are often known as Abstract Base Classes.

Mixins

16

- used to pull in implementation
  - not part of the is-a relationship
  - by convention, not enforced by the language
- common bits of functionality
  - e.g. implement `__gt__`, `__eq__` &c. using `__lt__`
  - you only need to implement `__lt__` in your class

Mixins are a form of inheritance (usually multiple inheritance) where only code re-use is a concern. That is, the base class and the derived class are not related conceptually. Some languages have a special 'mixin' construct for this, though more often, this is simply a convention. Programmers understand that they should not use the base type as a general form of all its derived classes. Python belongs to this latter category.

Interfaces

- realized as 'abstract' classes in Python
  - just throw a NotImplemented exception
  - document the intent in a docstring
- participates in is-a relationships
- partially displaced by duck typing
  - more important in other languages (think Java)

Interfaces are the dual to mixins: they only exist to give structure to the object hierarchy, but do not provide any code sharing. In Python, interfaces are rarely used, because duck typing covers most of the use cases traditionally served by interfaces.
Nonetheless, when an interface is called for in Python, there is no special syntax: by convention, an interface is a class with no attributes and with methods which all throw NotImplemented.

Composition

- attributes of objects can be other objects
  - (also, everything is an object in Python)
- encodes a has-a relationship
  - a circle has a center and a radius
  - a circle is a shape

Object composition is another important approach to code re-use (ostensibly much more important than inheritance, even though it is often neglected). Where inheritance encodes an is-a relationship, composition encodes a has-a relationship. Since in Python, everything is an object, each attribute is an instance of object composition. In languages with data types other than objects, the distinction is more useful.

Constructors

- this is the `__init__` method
- initializes the attributes of the instance
- can call superclass constructors explicitly
  - not called automatically (unlike C++, Java)
  - `MySuperClass.__init__( self )`
  - `super().__init__` (if unambiguous)

We have mentioned local invariants earlier. Object initialization is an important part of that story – the initialization method is responsible for putting the object into a consistent 'blank' or 'base' state. After initialization, all local invariants should hold. Correct methods invoked on consistent objects should keep the object consistent. Together, those two properties mean that correct objects should never violate their local invariants.
In Python, there is a bit of a catch: attributes of superclasses are initialized by the `__init__` method of that superclass, but this method is not called automatically when the object is created. Therefore, you need to always remember to call the `__init__` method of the superclass whenever appropriate.

Class and Object Dictionaries

- most objects are basically dictionaries
- try e.g. `foo.__dict__` (for a suitable `foo`)
- saying `foo.x` means `foo.__dict__["x"]`
  - if that fails, `type(foo).__dict__["x"]` follows
  - then superclasses of `type(foo)`, according to MRO
- this is what makes monkey patching possible

As we have seen earlier, object attributes behave like dictionaries, and in many cases, that's exactly how they are implemented in Python. It is possible to interact with this dictionary directly, through the (magic) attribute `__dict__`. The exceptions are built-in objects like `int`, `str` or `list` and slot-based objects (more on those in a later lecture).
On objects with dictionaries, saying `foo.x` is interpreted to mean `foo.__dict__['x']`. If the attribute (or method!) is not in the object dictionary, it is looked up in the class dictionary (remember that classes are objects too). Then superclasses and so on. The protocol is rather complicated – you can look up the details in the manual.
Remember that in case of methods, in addition to the lookup described above, the dot syntax also binds the first argument of the method to the object on the left of the dot.
It is possible to add new methods and attributes to objects with dictionaries at runtime (by simply adding them through the `__dict__` pseudo-attribute). This is often called 'monkey patching'. It is, however, not possible with objects without dictionaries (though methods can still be added to their classes, which do have dictionaries).

Writing Classes

```
class Person:
  def __init__( self, name ):
    self.name = name
  def greet( self ):
    print( "hello " + self.name )

p = Person( "you" )
p.greet()
```

The syntax for creating new classes is fairly straight-forward. Just remember that attributes are always created in the `__init__` method, never by writing them as variables alongside methods, like in most other languages.

Functions

- top-level functions/procedures are possible
- they are usually 'scoped' via the module system
- functions are also objects
  - try `print.__class__` (or `type(print)`)
- some functions are built in (`print`, `len`, ...)

While Python is object-based to the extreme, unlike some other languages (e.g. Java), free-standing functions are allowed, and common. Groups of related functions are bundled into modules (possibly along with some classes). Unlike in C, there is no single global namespace for functions. Like everything else in Python, functions are also objects. For instance:

```
def foo():
    print( "hello" )
```

```
>>> type( foo )
<class 'function'>
```

A few universally-useful functions are built into the interpreter, like print above, or len, hash, eval, range and so on. A complete list can be obtained by asking the interpreter, like this:

```
__builtins__.__dict__.keys()
```

However, besides functions, this list includes a number of classes: the standard types (int, dict and so on) and the standard exception classes (SyntaxError, NameError and so on). To show off the introspection capabilities of Python, let's get a list of just the built-in functions:

```
for n, o in __builtins__.__dict__.items():
    if type( o ) == type( print ):
        print( n )
```

Remember though: with great power comes great responsibility.

Modules in Python

23

- modules are just normal .py files
- import executes a file by name
  - it will look into system-defined locations
  - the search path includes the current directory
  - they typically only define classes & functions
- import sys → lets you use sys.argv
- from sys import argv → you can write just argv

In Python, there is no special syntax for creating modules: each source file can be imported as a module. It is customary that modules do not execute any 'top level' code (that is, statements outside of functions or classes), though this is not enforced in any way.
There is, however, special syntax for loading modules: the import keyword. On import foo, Python will look for a file named foo.py, first in the current directory and if that fails, in the standard library (which is located in a system-specific location).
When that file is located, it is executed as if it was a normal program, and at the end, all toplevel names that it has defined become available as attributes of the module. That is, if foo.py looks like this:

```
def hello():
    print( "hello" )
```

then bar.py may look like this:

```
import foo
foo.hello()
```

What this means is that the import in bar.py locates foo.py, executes it, then collects the toplevel definitions (in this case the function hello) and makes them available via the 'dot' syntax under the name of the module (foo.hello). Behind the scenes, the module is (of course) an object and the toplevel definitions from the file being imported are attached to that object as standard attributes.

Part 2: Memory Management & Builtin Types

Memory

25

- most program data is stored in 'memory'
  - an array of byte-addressable data storage
  - address space managed by the OS
  - 32 or 64 bit numbers as addresses
- typically backed by RAM

Programs primarily consist of two things: code, which drives the CPU, and data, which drives the program. Even though data has similar influence over the program as the program has over the processor, we usually think of data as passive: it sits around, waiting for the program to do about it. The program goes through conditionals and loops and at each turn, a piece of data decides which branch to take and whether to terminate the loop or perform another iteration. But of course, we are used to think about this in terms of the program reading and changing the data.
Both the data and the program is stored in memory. This memory is, from a programmer's viewpoint, a big array of bytes. There might be holes in it (indices which you cannot access), but otherwise the analogy works quite well. Importantly, addresses are really just indices, that is, numbers.
On the lowest level, most of the memory is the large-capacity dynamic random-access memory installed in the computer, though some of the bits and pieces are stored in static RAM on the CPU (cache) or even in the registers. The hardware, the operating system and the compiler (or interpreter) all conspire to hide this, though, and let us pretend that the memory is just an array of bytes.

Language vs Computer

26

- programs use high-level concepts
  - objects, procedures, closures
  - values can be passed around
- the computer has a single array of bytes
  - and a bunch of registers

When we write programs, we use high-level abstractions all the time: from simple functions, through objects all the way to lexical closures. Let us first consider a very simple procedure, with no local variables, no arguments and no return value. You could be excused for thinking that it is the most mundane thing imaginable.
However, consider that a procedure like that must be able to return to its caller, and for that, it needs to remember a return address. And this is true for any procedure that is currently executing. This gives rise to an execution stack, one of the most ubiquitous structures for organizing memory.
Contrast this with the flat array of bytes that is available at the lowest level of a computer. It is quite clear that even the simplest programs written in the simplest programming languages need to organize this flat memory, and that it is not viable to do this manually.

Memory Management

27

- deciding where to store data
- high-level objects are stored in flat memory
  - they have a given (usually fixed) size
  - have limited lifetime

This is the domain of memory management. It is an umbrella term

that covers a wide range of techniques, from very simple to quite complicated. The basic job of a memory management subsystem is to decide where to place data.
This data could be more or less anything: in case of the execution stack we have mentioned earlier, the data is the return addresses and the organizational principle is a stack. As procedures are called, a new address is pushed on top of the stack, and when it returns, an address is popped off. The stack is implemented as a single pointer: the address of the top of the stack. Pushing moves the address in one direction, while popping moves it in the opposite direction. Other than that, the data is stored directly in the flat and otherwise unstructured memory. Notably, even an extremely simple idea like this gives us very powerful abstraction.
However, when we say memory management, we usually have something a little more sophisticated in mind. We owe the simplicity of the stack to the fact that lifetimes of procedures are strictly nested. That is, the procedure which started executing last will always be the first to finish. That means that the data associated with that procedure can be forgotten before the data associated with its caller. This principle naturally extends to procedures with local variables.

Memory Management Terminology

28

- object: an entity with an address and size
  - can contain references to other objects
  - not the same as language-level object
- lifetime: when is the object valid
  - live: references exist to the object
  - dead: the object is unreachable – garbage

Not everything in a program is this simple. Some data needs to be available for a long time, while other pieces of data can be thrown away almost immediately. Some pieces of data can refer to other pieces of data (that is, pointers exist). In the context of memory management, such pieces of data are called objects, which is of course somewhat confusing.
These two properties (object lifetime and existence of pointers) are the most important aspects of a memory object, and of memory management in general. Unsurprisingly, they are also closely related. An object is alive if anything else that is alive refers to it. Additionally, local variables are always alive, since they are directly reachable through the 'stack pointer' (the address of the top of the execution stack).
Objects which are not alive are dead: what happens to those objects does not matter for further execution of the program. Since their addresses have been forgotten, the program can never look at the object again, and the memory it occupies can be safely reclaimed.

Memory Management by Type

29

- manual: `malloc` and `free` in C
- static automatic
  - e.g. stack variables in C and C++
- dynamic automatic
  - pioneered by LISP, widely used

There are three basic types of memory management. There is the manual memory management provided by the C library through the `malloc` and `free` functions. It is called manual because no effort is made to track the lifetimes of objects automatically. The programmer is fully responsible for ensuring that objects are released by calling `free` when their lifetime ends.
If `free` is called too soon, the program may get very confused when it tries to store two different objects in the same place in memory. If it is called too late (i.e. never), the program leaks memory: it will be unable re-use memory which is occupied by dead objects. This is wasteful, and can cause the program to crash because it no longer has space to store new objects.
Even though it completely ignores lifetimes, the machinery behind this 'manual' memory management is rather sophisticated. It needs to keep track of which pieces of memory are available, and upon request (a call to `malloc`), it needs to be able to quickly locate a suitable address. This address must be such that the next N bytes, where N was provided as a parameter to `malloc`, are currently unused. How to do this efficiently is a topic almost worth its own course.
In comparison, the static automatic approach, which corresponds to the execution stack is rather simple, if efficient. It is automatic in the sense that the programmer does not need to explicitly deal with lifetimes, though in this case, that is achieved because their structure is extremely rigid.
Finally, dynamic automatic memory management combines the 'good' aspects of both: the lifetimes can be arbitrary and are tracked automatically.

Automatic Memory Management

30

- static vs dynamic
  - when do we make decisions about lifetime
  - compile time vs run time
- safe vs unsafe
  - can the program read unused memory?

The static vs dynamic aspect of an automatic memory management system governs when the decisions are made about object lifetime. In a static system, the lifetime is computed ahead of time, e.g. by the compiler. In a dynamic system, such decisions are made at runtime.
Another aspect of memory management is safety. A program which uses safe memory management can never get into a situation when it attempts to use the same piece of memory for two different objects. There are multiple ways to achieve this, though by far the most common is to use a dynamic automatic approach to memory management, which is naturally safe. This is because memory associated with an object is never reclaimed as long as a reference (a pointer) to the object exists.
However, other options exist: a program with local variables but no pointers is also naturally safe, though its memory use is rather restricted. A system with both static lifetimes and with pointers is available in e.g. Rust (though the principle is much older, see also linear types).

Object Lifetime

31

- the time between `malloc` and `free`
- another view: when is the object needed
  - often impossible to tell
  - can be safely over-approximated
  - at the expense of memory leaks

Static Automatic

32

- usually binds lifetime to lexical scope
- no passing references up the call stack
  - may or may not be enforced
- no lexical closures
- examples: C, C++

Dynamic Automatic

- over-approximate lifetime dynamically
- usually easiest for the programmer
  - until you need to debug a space leak
- reference counting, mark & sweep collectors
- examples: Java, almost every dynamic language

Reference Counting

- attach a counter to each object
- whenever a reference is made, increase
- whenever a reference is lost, decrease
- the object is dead when the counter hits 0
- fails to reclaim reference cycles

Mark and Sweep

- start from a root set (in-scope variables)
- follow references, mark every object encountered
- sweep: throw away all unmarked memory
- usually stops the program while running
- garbage is retained until the GC runs

Memory Management in CPython

- primarily based on reference counting
- optional mark & sweep collector
  - enabled by default
  - configure via `import gc`
  - reclaims cycles

Refcounting Advantages

- simple to implement in a 'managed' language
- reclaims objects quickly
- no need to pause the program
- easily made concurrent

Refcounting Problems

- significant memory overhead
- problems with cache locality
- bad performance for data shared between threads
- fails to reclaim cyclic structures

Data Structures

- an abstract description of data
- leaves out low-level details
- makes writing programs easier
- makes reading programs easier, too

Building Data Structures

- there are two kinds of types in python
  - built-in, implemented in C
  - user-defined (includes libraries)
- both kinds are based on objects
  - but built-ins only look that way

Mutability

- some objects can be modified
  - we say they are mutable
  - otherwise, they are immutable
- immutability is an abstraction
  - physical memory is always mutable
- in python, immutability is not 'recursive'

Built-in: `int`

- arbitrary precision integer
  - no overflows and other nasty behaviour
- it is an object, i.e. held by reference
  - uniform with any other kind of object
  - immutable
- both of the above make it slow
  - machine integers only in C-based modules

Additional Numeric Objects

- `bool`: `True` or `False`
  - how much is `True + True`?
  - is `0` true? is empty string?
- `numbers.Real`: floating point numbers
- `numbers.Complex`: a pair of above

Built-in: `bytes`

- a sequence of bytes (raw data)
- exists for efficiency reasons
  - in the abstract is just a tuple
- models data as stored in files
  - or incoming through a socket
  - or as stored in raw memory

Properties of `bytes`

- can be indexed and iterated
  - both create objects of type `int`
  - try this sequence: `id(x[1])`, `id(x[2])`
- mutable version: `bytearray`
  - the equivalent of C `char` arrays

Built-in: `str`

- immutable unicode strings
  - not the same as bytes
  - bytes must be decoded to obtain `str`
  - (and `str` encoded to obtain `bytes`)
- represented as utf-8 sequences in CPython
  - implemented in `PyCompactUnicodeObject`

Built-in: `tuple`

- an immutable sequence type
  - the number of elements is fixed
  - so is the type of each element
- but elements themselves may be mutable
  - `x = []` then `y = (x, 0)`
  - `x.append(1)` $\rightarrow$ `y == ([1], 0)`
- implemented as a C array of object references

Built-in: `list`

- a mutable version of `tuple`
  - items can be assigned `x[3] = 5`
  - items can be `append`-ed
- implemented as a dynamic array
  - many operations are amortised $O(1)$
  - `insert` is $O(n)$

Built-in: `dict`

- implemented as a hash table
- some of the most performance-critical code
  - dictionaries appear everywhere in python
  - heavily hand-tuned C code
- both keys and values are objects

Hashes and Mutability

- dictionary keys must be hashable
  - this implies recursive immutability
- what would happen if a key is mutated?
  - most likely, the hash would change
  - all hash tables with the key become invalid
  - this would be very expensive to fix

Built-in: `set`

- implements the math concept of a set
- also a hash table, but with keys only
  - a separate C implementation
- mutable – items can be added
  - but they must be hashable
  - hence cannot be changed

Built-in: `frozenset`

- an immutable version of `set`
- always hashable (since all items must be)
  - can appear in `set` or another `frozenset`
  - can be used as a key in `dict`
- the C implementation is shared with `set`

Efficient Objects: `__slots__`

- fixes the attribute names allowed in an object
- saves memory: consider 1-attribute object
  - with `__dict__`: 56 + 112 bytes
  - with `__slots__`: 48 bytes
- makes code faster: no need to hash anything
  - more compact in memory $\rightarrow$ better cache efficiency