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`

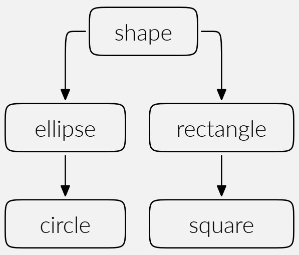
Besides 'standard' methods, which operate on objects (instances), in Python there are two other method types: static, which are really just plain functions hidden inside a class, and class methods which treat the class as an object. They are declared using decorators, like this:

```
@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 something with 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 thinking about this in terms of the program reading and changing the data.
Both the data and the program are 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 to 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 both simple and 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

33

- 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

34

- 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

35

- 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

36

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

Refcounting Advantages

37

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

Refcounting Problems

38

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

Data Structures

39

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

Building Data Structures

40

- 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

41

- 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`

42

- 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

43

- `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`

44

- 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`

45

- 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)` → `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 → better cache efficiency

Part 3: Text, JSON and XML

This lecture is the first part of a two-lecture block on persistent storage. This week, we will first briefly look at working with files, and then proceed to talk about text files specifically.

Transient Data

- lives in program memory
- data structures, objects
- interpreter state
- often implicit manipulation
- more on this next week

Persistent Data

- (structured) text or binary files
- relational (SQL) databases
- object and 'flat' databases (NoSQL)
- manipulated explicitly

Persistent Storage

- 'local' file system
  - stored on HDD, SSD, ...
  - stored somwhere in a local network
- 'remote', using an application-level protocol
  - local or remote databases
  - cloud storage &c.

Reading Files

58

- opening files: `open('file.txt', 'r')`
- files can be iterated

```
f = open( 'file.txt', 'r' )
for line in f:
    print( line )
```

Resource Acquisition

59

- plain `open` is prone to resource leaks
  - what happens during an exception?
  - holding a file open is not free
- pythonic solution: `with` blocks
  - defined in PEP 343
  - binds resources to scopes

Detour: PEP

60

- PEP stands for Python Enhancement Proposal
- akin to RFC documents managed by IETF
- initially formalise future changes to Python
  - later serve as documentation for the same
- <https://www.python.org/dev/peps/>

Using `with`

61

```
with open('/etc/passwd', 'r') as f:
    for line in f:
        do_stuff( line )
```

- still safe if `do_stuff` raises an exception

Finalizers

62

- there is a `__del__` method
- but it is not guaranteed to run
  - it may run arbitrarily late
  - or never
- not very good for resource management

Context Managers

63

- `with` has an associated protocol
- you can use `with` on any context manager
- which is an object with `__enter__` and `__exit__`
- you can create your own

Part 3.1: Text and Unicode

We will now turn our attention to text and its representation in a computer.

Representing Text

65

- ASCII: one byte = one character
  - total of 127 different characters
  - not very universal
- 8-bit encodings: 255 characters
- multi-byte encodings for non-Latin scripts

Representation of text in the computer used to be a relatively simple affair while it was English-only and one byte was one character. Things are not so simple anymore. Even with 8-bit character sets, the available alphabet is extremely limited, and can't even cover latin-based European languages. This led to huge amount of fragmentation and at the height of it, essentially each region had its own character encoding. Some of them had 2 or 3. Non-latin alphabets like Chinese or Japanese had no hope of fitting into single-byte encodings and have always used multiple bytes to encode a single character.

Unicode

66

- one character encoding to rule them all
- supports all extant scripts and writing systems
  - and a whole bunch of dead scripts, too
- approx. 143000 code points
- collation, segmentation, comparison, ...

A universal character encoding, with roots in the late 80s and early 90s. Of course, adoption was not immediate, though most computer systems nowadays use Unicode for representing and processing text. Nonetheless, you can still encounter software which will default to legacy 8-bit encodings, or even outright fall apart on Unicode text. Pretty much all extant scripts and languages are covered by recent revisions of Unicode. Besides character encoding, Unicode defines many other aspects of text processing and rendering. Sorting (collation) of strings is a huge, complicated topic unto itself. Likewise, segmentation – finding boundaries of graphemes and words – is a complex area covered by Unicode. Even seemingly simple matters like string equality are, in fact, quite hard.

Code Point

67

- basic unit of encoding characters
- letters, punctuation, symbols
- combining diacritical marks
- not the same thing as a character
- code points range from 1 to 10FFFF

Unicode Encodings

68

- deals with representing code points
- UCS = Universal Coded Character Set
  - fixed-length encoding
  - two variants: UCS-2 (16 bit) and UCS-4 (32 bit)
- UTF = Unicode Transformation Format
  - variable-length encoding
  - variants: UTF-8, UTF-16 and UTF-32

Grapheme

- technically 'extended grapheme cluster'
- a logical character, as expected by users
  - encoded using 1 or more code points
- multiple encodings of the same grapheme
  - e.g. composed vs decomposed
  - `U+0041 U+0300` vs `U+0C00`: À vs À

Segmentation

- breaking text into smaller units
  - graphemes, words and sentences
- algorithms defined by the unicode spec
  - Unicode Standard Annex #29
  - graphemes and words are quite reliable
  - sentences not so much (too much ambiguity)

Normal Form

- Unicode defines 4 canonical (normal) forms
  - NFC, NFD, NFKC, NFKD
  - NFC = Normal Form Composed
  - NFD = Normal Form Decomposed
- K variants = looser, lossy conversion
- all normalization is idempotent
- NFC does not give you 1 code point per grapheme

`str` vs `bytes`

- iterating `bytes` gives individual bytes
  - indexing is fast – fixed-size elements
- iterating `str` gives code points
  - slightly slower, because it uses UTF-8
  - does not iterate over graphemes
- going back and forth: `str.encode`, `bytes.decode`

Python vs Unicode

- no native support for unicode segmentation
  - hence no grapheme iteration or word splitting
- convert everything into NFC and hope for the best
  - `unicodedata.normalize()`
  - will sometimes break (we'll discuss regexes in a bit)
  - most people don't bother
  - correctness is overrated → worse is better

Regular Expressions

- compiling: `r = re.compile( r"key: (.*)" )`
- matching: `m = r.match( "key: some value" )`
- extracting captures: `print( m.group( 1 ) )`
  - prints `some value`
- substitutions: `s2 = re.sub( r"\s*$", '', s1 )`
  - strips all trailing whitespace in `s1`

Detour: Raw String Literals

- the `r` in `r"..."` stands for raw (not regex)
- normally, `\` is magical in strings
  - but `\` is also magical in regexes
  - nobody wants to write `\\s` &c.
  - not to mention `\\\\` to match a literal `\`
- not super useful outside of regexes

Detour: Other Literal Types

- byte strings: `b"abc"` → `bytes`
- formatted string literals: `f"x {y}"`

```
x = 12
print( f"x = {x}" )
```

- triple-quote literals: `"""xy"""`

Regular Expressions vs Unicode

```
import re
s = "\u0041\u0300" # À
t = "\u00c0"       # À
print( s, t )
print( re.match( "..", s ), re.match( "..", t ) )
print( re.match( "\w+$", s ), re.match( "\w+$", t ) )
print( re.match( "À", s ), re.match( "À", t ) )
```

Regexes and Normal Forms

- some of the problems can be fixed by NFC
  - some go away completely (literal unicode matching)
  - some become rarer (the "." and "\w" problems)
- most text in the wild is already in NFC
  - but not all of it
  - case in point: filenames on macOS (NFD)

Decomposing Strings

- recall that `str` is immutable
- splitting: `str.split(':')`
  - `None` = split on any whitespace
- split on first delimiter: `partition`
- better whitespace stripping: `s2 = s1.strip()`
  - also `lstrip()` and `rstrip()`

Searching and Matching

- `startswith` and `endswith`
  - often convenient shortcuts
- `find` = `index`
  - generic substring search

Building Strings

81

- format literals and str.format
- str.replace – substring search and replace
- str.join – turn lists of strings into a string

Part 3.2: Structured Text

JSON

83

- structured, text-based data format
- atoms: integers, strings, booleans
- objects (dictionaries), arrays (lists)
- widely used around the web &c.
- simple (compared to XML or YAML)

JSON: Example

84

```
{
    "composer": [ "Bach, Johann Sebastian" ],
    "key": "g",
    "voices": {
        "1": "oboe",
        "2": "bassoon"
    }
}
```

JSON: Writing

85

- printing JSON seems straightforward enough
- but: double quotes in strings
- strings must be properly \-escaped during output
- also pesky commas
- keeping track of indentation for human readability
- better use an existing library: `import json`

JSON in Python

86

- json.dumps = short for dump to string
- python dict/list/str/... data comes in
- a string with valid JSON comes out

Workflow

- just convert everything to dict and list
- run json.dumps or json.dump(data, file)

Python Example

87

```
d = {}
d["composer"] = ["Bach, Johann Sebastian"]
d["key"] = "g"
d["voices"] = { 1: "oboe", 2: "bassoon" }
json.dump( d, sys.stdout, indent=4 )
```

Beware: keys are always strings in JSON

Parsing JSON

88

- import json
- json.load is the counterpart to json.dump from above
  - de-serialise data from an open file
  - builds lists, dictionaries, etc.
- json.loads corresponds to json.dumps

XML

89

- meant as a lightweight and consistent redesign of SGML
  - turned into a very complex format
- heaps of invalid XML floating around
  - parsing real-world XML is a nightmare
  - even valid XML is pretty challenging

XML: Example

90

```
<Order OrderDate="1999-10-20">
  <Address Type="Shipping">
    <Name>Ellen Adams</Name>
    <Street>123 Maple Street</Street>
  </Address>
  <Item PartNumber="872-AA">
    <ProductName>Lawnmower</ProductName>
    <Quantity>1</Quantity>
  </Item>
</Order>
```

XML: Another Example

91

```
<BLOKY_OBSAH>
  <STUDENT>
    <OBSAH>25 bodů</OBSAH>
    <UCO>72873</UCO>
    <ZMENENO>20160111104208</ZMENENO>
    <ZMENIL>395879</ZMENIL>
  </STUDENT>
</BLOKY_OBSAH>
```

XML Features

92

- offers extensible, rich structure
  - tags, attributes, entities
  - suited for structured hierarchical data
- schemas: use XML to describe XML
  - allows general-purpose validators
  - self-documenting to a degree

XML vs JSON

93

- both work best with trees
- JSON has basically no features
  - basic data structures and that's it
- JSON data is ad-hoc and usually undocumented
  - but: this often happens with XML anyway

XML Parsers

94

- DOM = Document Object Model
- SAX = Simple API for XML
- expat = fast SAX-like parser (but not SAX)
- ElementTree = DOM-like but more pythonic

XML: DOM

95

- read the entire XML document into memory
- exposes the AST (Abstract Syntax Tree)
- allows things like XPath and CSS selectors
- the API is somewhat clumsy in Python

XML: SAX

96

- event-driven XML parsing
- much more efficient than DOM
  - but often harder to use
- only useful in Python for huge XML files
  - otherwise just use ElementTree

XML: ElementTree

97

```
for child in root:
    print child.tag, child.attrib

# Order { OrderDate: "1999-10-20" }
```

- supports tree walking, XPath
- supports serialization too

Part 4: Databases, SQL

NoSQL / Non-relational Databases

99

- umbrella term for a number of approaches
  - flat key/value and column stores
  - document and graph stores
- no or minimal schemas
- non-standard query languages

Key-Value Stores

100

- usually very fast and very simple
- completely unstructured values
- keys are often database-global
  - workaround: prefixes for namespacing
  - or: multiple databases

NoSQL & Python

101

- redis (redis-py) module (Redis is Key-Value)
- memcached (another Key-Value store)
- PyMongo for talking to MongoDB (document-oriented)
- CouchDB (another document-oriented store)
- neo4j or cayley (module pyley) for graph structures

SQL and RDBMS

102

- SQL = Structured Query Language
- RDBMS = Relational DataBase Management System
- SQL is to NoSQL what XML is to JSON
- heavily used and extremely reliable

SQL: Example

103

```
select name, grade from student;
select name from student where grade < 'C';
insert into student ( name, grade ) values
                    ( 'Random X. Student', 'C' );
select * from student
    join enrollment on student.id = enrollment.student
    join group on group.id = enrollment.group;
```

SQL: Relational Data

104

- JSON and XML are hierarchical
  - or built from functions if you like
- SQL is relational
  - relations = generalized functions
  - can capture more structure
  - much harder to efficiently process

SQL: Data Definition

- mandatory, unlike XML or JSON
- gives the data a rather rigid structure
- tables (relations) and columns (attributes)
- static data types for columns
- additional consistency constraints

SQL: Constraints

- help ensure consistency of the data
- foreign keys: referential integrity
  - ensures there are no dangling references
  - but: does not prevent accidental misuse
- unique constraints
- check constraints: arbitrary consistency checks

SQL: Query Planning

- an RDBMS makes heavy use of indexing
  - using B trees, hashes and similar techniques
  - indices are used automatically
- all the heavy lifting is done by the backend
  - highly-optimized, low-level code
  - efficient handling of large data

SQL: Reliability and Flexibility

- most RDBMS give ACID guarantees
  - transparently solves a lot of problems
  - basically impossible with normal files
- support for schema alterations
  - `alter table` and similar
  - nearly impossible in ad-hoc systems

SQLite

- lightweight in-process SQL engine
- the entire database is in a single file
- convenient python module, `sqlite3`
- stepping stone for a "real" database

Other Databases

- you can talk to most SQL DBs using python
- postgresql (`psycopg2`, ...)
- mysql / mariadb (`mysql-python`, `mysql-connector`, ...)
- big & expensive: Oracle (`cx_oracle`), DB2 (`pyDB2`)
- most of those are much more reliable than SQLite

SQL Injection

```
sql = "SELECT * FROM t WHERE name = '" + n + "'"
```

- the above code is bad, never do it
- consider the following

```
n = "x'; drop table students --"
n = "x'; insert into passwd (user, pass) ..."
```

Avoiding SQL Injection

- use proper SQL-building APIs
  - this takes care of escaping internally
- templates like `insert ... values (?, ?)`
  - the `?` get safely substituted by the module
  - e.g. the `execute` method of a cursor

PEP 249

- informational PEP, for library writers
- describes how database modules should behave
  - ideally, all SQL modules have the same interface
  - makes it easy to swap a database backend
- but: SQL itself is not 100% portable

SQL Pitfalls

- `sqlite` does not enforce all constraints
  - you need to `pragma foreign_keys = on`
- no portable syntax for autoincrement keys
- not all (column) types are supported everywhere
- no portable way to get the key of last insert

More Resources & Stuff to Look Up

- SQL: `https://www.w3schools.com/sql/`
- `https://docs.python.org/3/library/sqlite3.html`
- Object-Relational Mapping
- SQLAlchemy: constructing portable SQL

Part 5: Operators, Iterators and Exceptions

Callable Objects

- user-defined functions (module-level `def`)
- user-defined methods (instance and class)
- built-in functions and methods

User-defined Functions

- come about from a module-level `def`
- metadata: `__doc__`, `__name__`, `__module__`
- scope: `__globals__`, `__closure__`
- arguments: `__defaults__`, `__kwdefaults__`
- type annotations: `__annotations__`
- the code itself: `__code__`

Positional and Keyword Arguments

- user-defined functions have positional arguments
- and keyword arguments
  - `print("hello", file=sys.stderr)`
  - arguments are passed by name
  - which style is used is up to the caller
- variadic functions: `def foo(*args, **kwargs)`
  - `args` is a `tuple` of unmatched positional args
  - `kwargs` is a `dict` of unmatched keyword args

Lambdas

- `def` functions must have a name
- lambdas provide anonymous functions
- the body must be an expression
- syntax: `lambda x: print("hello", x)`
- standard user-defined functions otherwise

Instance Methods

- comes about as `object.method`
  - `print(x.foo)` → `<bound method Foo.foo of ...>`
- combines the class, instance and function itself
- `__func__` is a user-defined function object
- let `bar = x.foo`, then
  - `x.foo()` → `bar.__func__(bar.__self__)`

Iterators

- objects with `__next__` (since 3.x)
  - iteration ends on `raise StopIteration`
- iterable objects provide `__iter__`
  - sometimes, this is just `return self`
  - any `iterable` can appear in `for x in iterable`

```
class FooIter:
    def __init__(self):
        self.x = 10
    def __iter__(self): return self
    def __next__(self):
        if self.x:
            self.x -= 1
        else:
            raise StopIteration
        return self.x
```

Generators (PEP 255)

- written as a normal function or method
- they use `yield` to generate a sequence
- represented as special callable objects
  - exist at the C level in CPython

```
def foo(*lst):
    for i in lst: yield i + 1
list(foo(1, 2)) # prints [2, 3]
```

`yield from`

- calling a generator produces a generator object
- how do we call one generator from another?
- same as `for x in foo(): yield x`

```
def bar(*lst):
    yield from foo(*lst)
    yield from foo(*lst)
list(bar(1, 2)) # prints [2, 3, 2, 3]
```

Decorators

- written as `@decor` before a function definition
- `decor` is a regular function (`def decor(f)`)
  - `f` is bound to the decorated function
  - the decorated function becomes the result of `decor`
- classes can be decorated too
- you can ‘create’ decorators at runtime
  - `@mkdecor("moo")` (`mkdecor` returns the decorator)
  - you can stack decorators

127

```
def decor(f):
    return lambda: print("bar")
def mkdecor(s):
    return lambda g: lambda: print(s)

@decor
def foo(f): print("foo")
@mkdecor("moo")
def moo(f): print("foo")

# foo() prints "bar", moo() prints "moo"
```

128

List Comprehension

- a concise way to build lists
- combines a `filter` and a `map`

```
[ 2 * x for x in range(10) ]
[ x for x in range(10) if x % 2 == 1 ]
[ 2 * x for x in range(10) if x % 2 == 1 ]
[ (x, y) for x in range(3) for y in range(2) ]
```

129

Operators

- operators are (mostly) syntactic sugar
- `x < y` rewrites to `x.__lt__(y)`
- `is` and `is not` are special
  - are the operands the same object?
  - also the ternary (conditional) operator

130

Non-Operator Builtins

- `len(x)` → `x.__len__()` (length)
- `abs(x)`→ `x.__abs__()` (magnitude)
- `str(x)` → `x.__str__()` (printing)
- `repr(x)` → `x.__repr__()` (printing for `eval`)
- `bool(x)` and `if x`: `x.__bool__()`

131

Arithmetic

- a standard selection of operators
- `/` is floating point, `//` is integral
- `+=` and similar are somewhat magical
  - `x += y` → `x = x.__iadd__(y)` if defined
  - otherwise `x = x.__add__(y)`

132

```
x = 7         # an int is immutable
x += 3        # works, x = 10, id(x) changes

lst = [7, 3]
lst[0] += 3   # works too, id(lst) stays same

tup = (7, 3)  # a tuple is immutable
tup += (1, 1) # still works (id changes)
tup[0] += 3   # fails
```

133

Relational Operators

- operands can be of different types
- equality: `!=`, `==`
  - by default uses object identity
- ordering: `<`, `<=`, `>`, `>=` (`TypeError` by default)
- consistency is not enforced

134

Relational Consistency

- `__eq__` must be an equivalence relation
- `x.__ne__(y)` must be the same as `not x.__eq__(y)`
- `__lt__` must be an ordering relation
  - compatible with `__eq__`
  - consistent with each other
- each operator is separate (mixins can help)
  - or perhaps a class decorator

135

Collection Operators

- `in` is also a membership operator (outside `for`)
  - implemented as `__contains__`
- indexing and slicing operators
  - `del x[y]` → `x.__delitem__(y)`
  - `x[y]` → `x.__getitem__(y)`
  - `x[y] = z` → `x.__setitem__(y, z)`

136

Conditional Operator

- also known as a ternary operator
- written `x if cond else y`
  - in C: `cond ? x : y`
- forms an expression, unlike `if`
  - can e.g. appear in a lambda
  - or in function arguments, &c.

Exceptions

137

- an exception interrupts normal control flow
- it's called an exception because it is exceptional
  - never mind StopIteration
- causes methods to be interrupted
  - until a matching except block is found
  - also known as stack unwinding

Life Without Exceptions

138

```
int fd = socket( ... );
if ( fd < 0 )
    ... /* handle errors */
if ( bind( fd, ... ) < 0 )
    ... /* handle errors */
if ( listen( fd, 5 ) <  0 )
    ...  /* handle errors */
```

With Exceptions

139

```
try:
    sock = socket.socket( ... )
    sock.bind( ... )
    sock.listen( ... )
except ...:
    # handle errors
```

Exceptions vs Resources

140

```
x = open( "file.txt" )
# stuff
raise SomeError
```

- who calls x.close()
- this would be a resource leak

Using finally

141

```
try:
    x = open( "file.txt" )
    # stuff
finally:
    x.close()
```

- works, but tedious and error-prone

Using with

142

```
with open( "file.txt" ) as f:
    # stuff
```

- with takes care of the finally and close
- with x as y sets y = x.__enter__()
  - and calls x.__exit__(...) when leaving the block

The @property decorator

143

- attribute syntax is the preferred one in Python
- writing useless setters and getters is boring

```
class Foo:
    @property
    def x(self): return 2 * self.a
    @x.setter
    def x(self, v): self.a = v // 2
```

Part 6: Closures, Coroutines, Concurrency

Concurrency & Parallelism

145

- threading – thread-based parallelism
- multiprocessing
- concurrent – future-based programming
- subprocess
- sched, a general-purpose event scheduler
- queue, for sending objects between threads

Threading

146

- low-level thread support, module threading
- Thread objects represent actual threads
  - threads provide start() and join()
  - the run() method executes in a new thread
- mutexes, semaphores &c.

The Global Interpreter Lock

147

- memory management in CPython is not thread-safe
  - Python code runs under a global lock
  - pure Python code cannot use multiple cores
- C code usually runs without the lock
  - this includes `numpy` crunching

Multiprocessing

148

- like `threading` but uses processes
- works around the GIL
  - each worker process has its own interpreter
- queued/sent objects must be pickled
  - see also: the `pickle` module
  - this causes substantial overhead
  - functions, classes &c. are pickled by name

Futures

149

- like coroutine `await` but for subroutines
- a `Future` can be waited for using `f.result()`
- scheduled via `concurrent.futures.Executor`
  - `Executor.map` is like `asyncio.gather`
  - `Executor.submit` is like `asyncio.create_task`
- implemented using process or thread pools

Native Coroutines (PEP 492)

150

- created using `async def` (since Python 3.5)
- generalisation of generators
  - `yield from` is replaced with `await`
  - an `__await__` magic method is required
- a coroutine can be suspended and resumed

Coroutine Scheduling

151

- coroutines need a scheduler
- one is available from `asyncio.get_event_loop()`
- along with many coroutine building blocks
- coroutines can actually run in parallel
  - via `asyncio.create_task` (since 3.7)
  - via `asyncio.gather`

Async Generators (PEP 525)

152

- `async def` + `yield`
- semantics like simple generators
- but also allows `await`
- iterated with `async for`
  - `async for` runs sequentially

Execution Stack

153

- made up of activation frames
- holds local variables
- and return addresses
- in dynamic languages, often lives in the heap

Variable Capture

154

- variables are captured lexically
- definitions are a dynamic / run-time construct
  - a nested definition is executed
  - creates a closure object
- always by reference in Python
  - but can be by-value in other languages

Using Closures

155

- closures can be returned, stored and called
  - they can be called multiple times, too
  - they can capture arbitrary variables
- closures naturally retain state
- this is what makes them powerful

Objects from Closures

156

- so closures are essentially code + state
- wait, isn't that what an object is?
- indeed, you can implement objects using closures

The Role of GC

157

- memory management becomes a lot more complicated
- forget C-style 'automatic' stack variables
- this is why the stack is actually in the heap
- this can go as far as form reference cycles

Coroutines

158

- coroutines are a generalisation of subroutines
- they can be suspended and re-entered
- coroutines can be closures at the same time
- the code of a coroutine is like a function
- a suspended coroutine is like an activation frame

Yield

159

- suspends execution and 'returns' a value
- may also obtain a new value (cf. `send`)
- when re-entered, continue where we left off

```
for i in range(5): yield i
```

Send

160

- with yield, we have one-way communication
- but in many cases, we would like two-way
- a suspended coroutine is an object in Python
  - with a send method which takes a value
  - send re-enters the coroutine

Yield From and Await

161

- yield from is mostly a generator concept
- await basically does the same thing
  - call out to another coroutine
  - when it suspends, so does the entire stack

Suspending Native Coroutines

162

- this is not actually possible
  - not with async-native syntax anyway
- you need a yield
  - for that, you need a generator
  - use the types.coroutine decorator

Event Loop

163

- not required in theory
- useful also without coroutines
- there is a synergistic effect
  - event loops make coroutines easier
  - coroutines make event loops easier

Part 7: Communication & HTTP with asyncio

Running Programs (the old way)

165

- os.system is about the simplest
  - also somewhat dangerous – shell injection
  - you only get the exit code
- os.popen allows you to read output of a program
  - alternatively, you can send input to the program
  - you can't do both (would likely deadlock anyway)
  - runs the command through a shell, same as os.system

Low-level Process API

166

- POSIX-inherited interfaces (on POSIX systems)
- os.exec: replace the current process
- os.fork: split the current process in two
- os.forkpty: same but with a PTY

Detour: bytes vs str

167

- strings (class str) represent text
  - that is, a sequence of unicode points
- files and network connections handle data
  - represented in Python as bytes
- the bytes constructor can convert from str
  - e.g. b = bytes("hello", "utf8")

Running Programs (the new way)

168

- you can use the subprocess module
- subprocess can handle bidirectional IO
  - it also takes care of avoiding IO deadlocks
  - set input to feed data to the subprocess
- internally, run uses a Popen object
  - if run can't do it, Popen probably can

Getting subprocess Output

169

- available via run since Python 3.7
- the run function returns a CompletedProcess
- it has attributes stdout and stderr
- both are bytes (byte sequences) by default
- or str if text or encoding were set
- available if you enabled capture_output

Running Filters with Popen

170

- if you are stuck with 3.6, use Popen directly
- set stdin in the constructor to PIPE
- use the communicate method to send the input
- this gives you the outputs (as bytes)

171

```
import subprocess
from subprocess import PIPE
input = bytes( "x\na\nb\ny", "utf8")
p = subprocess.Popen(["sort"], stdin=PIPE,
                     stdout=PIPE)
out = p.communicate(input=input)
# out[0] is the stdout, out[1] is None
```

Subprocesses with asyncio

172

- import asyncio.subprocess
- create_subprocess_exec, like subprocess.run
  - but it returns a Process instance
  - Process has a communicate async method
- can run things in background (via tasks)
  - also multiple processes at once

Protocol-based asyncio subprocesses

173

- let loop be an implementation of the asyncio event loop
- there's subprocess_exec and subprocess_shell
  - sets up pipes by default
- integrates into the asyncio transport layer (see later)
- allows you to obtain the data piece-wise
- https://docs.python.org/3/library/asyncio-protocol.html

Sockets

174

- the socket API comes from early BSD Unix
- socket represents a (possible) network connection
- sockets are more complicated than normal files
  - establishing connections is hard
  - messages get lost much more often than file data

Socket Types

175

- sockets can be internet or unix domain
  - internet sockets connect to other computers
  - Unix sockets live in the filesystem
- sockets can be stream or datagram
  - stream sockets are like files (TCP)
  - you can write a continuous stream of data
  - datagram sockets can send individual messages (UDP)

Sockets in Python

176

- the socket module is available on all major OSes
- it has a nice object-oriented API
  - failures are propagated as exceptions
  - buffer management is automatic
- useful if you need to do low-level networking
  - hard to use in non-blocking mode

Sockets and asyncio

177

- asyncio provides sock_* to work with socket objects
- this makes work with non-blocking sockets a lot easier
- but your program needs to be written in async style
- only use sockets when there is no other choice
  - asyncio protocols are both faster and easier to use

Hyper-Text Transfer Protocol

178

- originally a simple text-based, stateless protocol
- however
  - SSL/TLS, cryptography (https)
  - pipelining (somewhat stateful)
  - cookies (somewhat stateful in a different way)
- typically between client and a front-end server
- but also as a back-end protocol (web server to app server)

Request Anatomy

179

- request type (see below)
- header (text-based, like e-mail)
- content

Request Types

- GET – asks the server to send a resource
- HEAD – like GET but only send back headers
- POST – send data to the server

Python and HTTP

180

- both client and server functionality
  - import http.client
  - import http.server
- TLS/SSL wrappers are also available
  - import ssl
- synchronous by default

Serving Requests

181

- derive from BaseHTTPRequestHandler
- implement a do_GET method
- this gets called whenever the client does a GET
- also available: do_HEAD, do_POST, etc.
- pass the class (not an instance) to HTTPServer

Serving Requests (cont'd)

182

- HTTPServer creates a new instance of your Handler
- the BaseHTTPRequestHandler machinery runs
- it calls your do_GET etc. method
- request data is available in instance variables
  - self.path, self.headers

Talking to the Client

183

- HTTP responses start with a response code
  - self.send_response(200, 'OK')
- the headers follow (set at least Content-Type)
  - self.send_header('Connection', 'close')
- headers and the content need to be separated
  - self.end_headers()
- finally, send the content by writing to self.wfile

Sending Content

184

- self.wfile is an open file
- it has a write() method which you can use
- sockets only accept byte sequences, not str
- use the bytes(string, encoding) constructor
  - match the encoding to your Content-Type

HTTP and asyncio

185

- the base asyncio currently doesn't directly support HTTP
- but: you can get aiohttp from PyPI
- contains a very nice web server
  - `from aiohttp import web`
  - minimum boilerplate, fully asyncio-ready

Aside: The Python Package Index

186

- colloquially known as PyPI (or cheese shop)
  - do not confuse with PyPy (Python in almost-Python)
- both source packages and binaries
  - the latter known as wheels (PEP 427, 491)
  - previously python eggs
- <https://pypi.python.org>

SSL and TLS

187

- you want to use the `ssl` module for handling HTTPS
  - this is especially true server-side
  - `aiohttp` and `http.server` are compatible
- you need to deal with certificates (loading, checking)
- this is a rather important but complex topic

Certificate Basics

188

- certificate is a cryptographically signed statement
  - it ties a server to a certain public key
  - the client ensures the server knows the private key
- the server loads the certificate and its private key
- the client must validate the certificate
  - this is typically a lot harder to get right

SSL in Python

189

- start with `import ssl`
- almost everything happens in the `SSLContext` class
- get an instance from `ssl.create_default_context()`
  - you can use `wrap_socket` to run an SSL handshake
  - you can pass the context to `aiohttp`
- if `httpd` is a `http.server.HTTPServer`:

```
httpd.socket = ssl.wrap_socket( httpd.socket, ... )
```

HTTP Clients

190

- there's a very basic `http.client`
- for a more complete library, use `urllib.request`
- `aiohttp` has client functionality
- all of the above can be used with `ssl`
- another 3rd party module: Python Requests

Part 8: Low-level asyncio

IO at the OS Level

192

- often defaults to blocking
  - `read` returns when data is available
  - this is usually OK for files
- but what about network code?
  - could work for a client

Threads and IO

193

- there may be work to do while waiting
  - waiting for IO can be wasteful
- only the calling (OS) thread is blocked
  - another thread may do the work
  - but multiple green threads may be blocked

Non-Blocking IO

194

- the program calls `read`
  - `read` returns immediately
  - even if there was no data
- but how do we know when to `read`?
  - we could poll
  - for example call `read` every 30ms

Polling

195

- trade-off between latency and throughput
  - sometimes, polling is okay
  - but is often too inefficient
- alternative: IO dispatch
  - useful when multiple IOs are pending
  - wait only if all are blocked

select
196

- takes a list of file descriptors
- block until one of them is ready
  - next read will return data immediately
- can optionally specify a timeout
- only useful for OS-level resources

Alternatives to select
197

- select is a rather old interface
- there is a number of more modern variants
- poll and epoll system calls
  - despite the name, they do not poll
  - epoll is more scalable
- kqueue and kevent on BSD systems

Synchronous vs Asynchronous
198

- the select family is synchronous
  - you call the function
  - it may wait some time
  - you proceed when it returns
- OS threads are fully asynchronous

The Thorny Issue of Disks
199

- a file is always 'ready' for reading
- this may still take time to complete
- there is no good solution on UNIX
- POSIX AIO exists but is sparsely supported
- OS threads are an option

IO on Windows
200

- select is possible (but slow)
- Windows provides real asynchronous IO
  - quite different from UNIX
  - the IO operation is directly issued
  - but the function returns immediately
- comes with a notification queue

The asyncio Event Loop
201

- uses the select family of syscalls
- why is it called async IO?
  - select is synchronous in principle
  - this is an implementation detail
  - the IOs are asynchronous to each other

How Does It Work
202

- you must use asyncio functions for IO
- an async read does not issue an OS read
- it yields back into the event loop
- the fd is put on the select list
- the coroutine is resumed when the fd is ready

Timers
203

- asyncio allows you to set timers
- the event loop keeps a list of those
- and uses that to set the select timeout
  - just uses the nearest timer expiry
- when a timer expires, its owner is resumed

Blocking IO vs asyncio
204

- all user code runs on the main thread
- you must not call any blocking IO functions
- doing so will stall the entire application
  - in a server, clients will time out
  - even if not, latency will suffer

DNS
205

- POSIX: getaddrinfo and getnameinfo
  - also the older API gethostbyname
- those are all blocking functions
  - and they can take a while
  - but name resolution is essential
- asyncio internally uses OS threads for DNS

Signals
206

- signals on UNIX are very asynchronous
- interact with OS threads in a messy way
- asyncio hides all this using C code

Native Coroutines (Reminder)
207

- delared using async def

```
async def foo():
    await asyncio.sleep( 1 )
```

- calling foo() returns a suspended coroutine
- which you can await
  - or turn it into an asyncio.Task

Tasks

208

- asyncio.Task is a nice wrapper around coroutines
  - create with asyncio.create_task()
- can be stopped prematurely using cancel()
- has an API for asking things:
  - done() tells you if the coroutine has finished
  - result() gives you the result

Tasks and Exceptions

209

- what if a coroutine raises an exception?
- calling result will re-raise it
  - i.e. it continues propagating from result()
- you can also ask directly using exception()
  - returns None if the coroutine ended normally

Asynchronous Context Managers

210

- normally, we use with for resource acquisition
  - this internally uses the context manager protocol
- but sometimes you need to wait for a resource
  - __enter__() is a subroutine and would block
  - this won't work in async-enabled code
- we need __enter__() to be itself a coroutine

async with

211

- just like wait but uses __aenter__(), __aexit__()
  - those are async def
- the async with behaves like an await
  - it will suspend if the context manager does
  - the coroutine which owns the resource can continue
- mainly used for locks and semaphores

Part 9: Python Pitfalls

Mixing Languages

213

- for many people, Python is not a first language
- some things look similar in Python and Java (C++, ...)
  - sometimes they do the same thing
  - sometimes they do something very different
  - sometimes the difference is subtle

Python vs Java: Decorators

214

- Java has a thing called annotations
- looks very much like a Python decorator
- in Python, decorators can drastically change meaning
- in Java, they are just passive metadata
  - other code can use them for meta-programming though

Class Body Variables

215

```
class Foo:
  some_attr = 42
```

- in Java/C++, this is how you create instance variables
- in Python, this creates class attributes
  - i.e. what C++/Java would call static attributes

Very Late Errors

216

```
if a == 2:
    priiiint("a is not 2")
```

- no error when loading this into python
- it even works as long as a != 2
- most languages would tell you much earlier

Very Late Errors (cont'd)

217

```
try:
    foo()
except TyyyypeError:
    print("my mistake")
```

- does not even complain when running the code
- you only notice when foo() raises an exception

Late Imports

218

```
if a == 2:
    import foo
    foo.say_hello()
```

- unless a == 2, mymod is not loaded
- any syntax errors don't show up until a == 2
  - it may even fail to exist

Block Scope

219

```
for i in range(10): pass
print(i) # not a NameError
```

- in Python, local variables are function-scoped
- in other languages, i is confined to the loop

Assignment Pitfalls

```
x = [ 1, 2 ]
y = x
x.append( 3 )
print( y ) # prints [ 1, 2, 3 ]
```

- in Python, everything is a reference
- assignment does not make copies

Equality of Iterables

- `[0, 1] == [0, 1]` → `True` (obviously)
- `range(2) == range(2)` → `True`
- `list(range(2)) == [0, 1]` → `True`
- `[0, 1] == range(2)` → `False`

Equality of `bool`

- `if 0: print( "yes" )` → nothing
- `if 1: print( "yes" )` → yes
- `False == 0` → `True`
- `True == 1` → `True`
- `0 is False` → `False`
- `1 is True` → `False`

Equality of `bool` (cont'd)

- `if 2: print( "yes" )` → yes
- `True == 2` → `False`
- `False == 2` → `False`

- `if '': print( "yes" )` → nothing
- `if 'x': print( "yes" )` → yes
- `'' == False` → `False`
- `'x' == True` → `False`

Mutable Default Arguments

```
def foo( x = [] ):
    x.append( 7 )
    return x
foo() # [ 7 ]
foo() # [ 7, 7 ]... wait, what?
```

Late Lexical Capture

```
f = [ lambda x : i * x for i in range( 5 ) ]
f[ 4 ]( 3 ) # 12
f[ 0 ]( 3 ) # 12 ... ?!

g = [ lambda x, i = i: i * x for i in range( 5 ) ]
g[ 4 ]( 3 ) # 12
g[ 0 ]( 3 ) # 0 ... fml

h = [ ( lambda x : i * x )( 3 ) for i in range( 5 ) ]
h # [ 0, 3, 6, 12 ] ... i kid you not
```

Dictionary Iteration Order

- in python <= 3.6
  - small dictionaries iterate in insertion order
  - big dictionaries iterate in 'random' order
- in python 3.7
  - all in insertion order, but not documented
- in python >= 3.8
  - guaranteed to iterate in insertion order

List Multiplication

```
x = [ [ 1 ] * 2 ] * 3
print( x ) # [ [ 1, 1 ], [ 1, 1 ], [ 1, 1 ] ]
x[ 0 ][ 0 ] = 2
print( x ) # [ [ 2, 1 ], [ 2, 1 ], [ 2, 1 ] ]
```

Forgotten Await

```
import asyncio
async def foo():
    print( "hello" )
async def main():
    foo()
asyncio.run( main() )
```

- gives warning `coroutine 'foo' was never awaited`

Python vs Java: Closures

- captured variables are `final` in Java
- but they are mutable in Python
  - and of course captured by reference
- they are whatever you tell them to be in C++

Explicit `super()`

- Java and C++ automatically call parent constructors
- Python does not
- you have to call them yourself

Setters and Getters

```
obj.attr
obj.attr = 4
```

- in C++ or Java, this is an assignment
- in Python, it can run arbitrary code
  - this often makes getters/setters redundant

Part 10: Testing, Profiling

Why Testing

- reading programs is hard
- reasoning about programs is even harder
- testing is comparatively easy

- difference between an example and a proof

What is Testing

- based on trial runs
- the program is executed with some inputs
- the outputs or outcomes are checked
- almost always incomplete

Testing Levels

- unit testing
  - individual classes
  - individual functions
- functional
  - system
  - integration

Testing Automation

- manual testing
  - still widely used
  - requires human
- semi-automated
  - requires human assistance
- fully automated
  - can run unattended

Testing Insight

- what does the test or tester know?
- black box: nothing known about internals
- gray box: limited knowledge
- white box: ‘complete’ knowledge

Why Unit Testing?

- allows testing small pieces of code
- the unit is likely to be used in other code
  - make sure your code works before you use it
  - the less code, the easier it is to debug
- especially easier to hit all the corner cases

Unit Tests with `unittest`

- `from unittest import TestCase`
- derive your test class from `TestCase`
- put test code into methods named `test_*`
- run with `python -m unittest program.py`
  - add `-v` for more verbose output

```
from unittest import TestCase

class TestArith(TestCase):
    def test_add(self):
        self.assertEqual(1, 4 - 3)
    def test_leq(self):
        self.assertTrue(3 <= 2 * 3)
```

Unit Tests with `pytest`

- a more pythonic alternative to `unittest`
  - `unittest` is derived from JUnit
- easier to use and less boilerplate
- you can use native python `assert`
- easier to run, too
  - just run `pytest` in your source repository

Test Auto-Discovery in `pytest`

- `pytest` finds your testcases for you
  - no need to register anything
- put your tests in `test_.py or _test.py`
- name your testcases (functions) `test_*`

Fixtures in pytest

243

- sometimes you need the same thing in many testcases
- in unittest, you have the test class
- pytest passes fixtures as parameters
  - fixtures are created by a decorator
  - they are matched based on their names

244

```
import pytest
import smtplib

@pytest.fixture
def smtp_connection():
    return smtplib.SMTP("smtp.gmail.com", 587)

def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
```

Property Testing

245

- writing test inputs is tedious
- sometimes, we can generate them instead
- useful for general properties like
  - idempotency (e.g. serialize + deserialize)
  - invariants (output is sorted, …)
  - code does not cause exceptions

Using hypothesis

246

- property-based testing for Python
- has strategies to generate basic data types
  - int, str, dict, list, set, …
- compose built-in generators to get custom types
- integrated with pytest

247

```
import hypothesis
import hypothesis.strategies as s

@hypothesis.given(s.lists(s.integers()))
def test_sorted(x):
    assert sorted(x) == x # should fail

@hypothesis.given(x=s.integers(), y=s.integers())
def test_cancel(x, y):
    assert (x + y) - y == x # looks okay
```

Going Quick and Dirty

248

- goal: minimize time spent on testing
- manual testing usually loses
  - but it has almost 0 initial investment
- if you can write a test in 5 minutes, do it
- useful for testing small scripts

Shell 101

249

- shell scripts are very easy to write
- they are ideal for testing IO behaviour
- easily check for exit status: set -e
- see what is going on: set -x
- use diff -u to check expected vs actual output

Shell Test Example

250

```
set -ex
python script.py < test1.in | tee out
diff -u test1.out out
python script.py < test2.in | tee out
diff -u test2.out out
```

Continuous Integration

251

- automated tests need to be executed
- with many tests, this gets tedious to do by hand
- CI builds and tests your project regularly
  - every time you push some commits
  - every night (e.g. more extensive tests)

CI: Travis

252

- runs in the cloud (CI as a service)
- trivially integrates with pytest
- virtualenv out of the box for python projects
- integrated with github
- configure in .travis.yml in your repo

CI: GitLab

253

- GitLab has its own CI solution (similar to travis)
- also available at FI
- runs tests when you push to your gitlab
- drop a .gitlab-ci.yml in your repository
- automatic deployment into heroku &c.

CI: Buildbot

254

- written in python/twisted
  - basically a framework to build a custom CI tool
- self-hosted and somewhat complicated to set up
  - more suited for complex projects
  - much more flexible than most CI tools
- distributed design

CI: Jenkins

255

- another self-hosted solution, this time in Java
  - widely used and well supported
- native support for python projects (including pytest)
  - provides a dashboard with test result graphs &c.
  - supports publishing sphinx-generated documentation

Print-based Debugging

256

- no need to be ashamed, everybody does it
- less painful in interpreted languages
- you can also use decorators for tracing
- never forget to clean your program up again

257

```
def debug(e):
    f = sys._getframe(1)
    v = eval(e, f.f_globals, f.f_locals)
    l = f.f_code.co_filename + ':'
    l += str(f.f_lineno) + ':'
    print(l, e, '=', repr(v), file=sys.stderr)

x = 1
debug('x + 1')
```

The Python Debugger

258

- run as `python -m pdb program.py`
- there's a built-in `help` command
- `next` steps through the program
- `break` to set a breakpoint
- `cont` to run until end or a breakpoint

What is Profiling

259

- measurement of resource consumption
- essential info for optimising programs
- answers questions about bottlenecks
  - where is my program spending most time?
  - less often: how is memory used in the program

Why Profiling

260

- 'blind' optimisation is often misdirected
  - it is like fixing bugs without triggering them
  - program performance is hard to reason about
- tells you exactly which point is too slow
  - allows for best speedup with least work

Profiling in Python

261

- provided as a library, `cProfile`
  - alternative: `profile` is slower, but more flexible
- run as `python -m cProfile program.py`
- outputs a list of lines/functions and their cost
- use `cProfile.run()` to profile a single expression

262

```
# python -m cProfile -s time fib.py

 ncalls  tottime  percall file:line(function)
13638/2    0.032    0.016 fib.py:1(fib_rec)
      2    0.000    0.000 {builtins.print}
      2    0.000    0.000 fib.py:5(fib_mem)
```

Part 11: Linear Algebra & Symbolic Math

Numbers in Python

264

- recall that numbers are objects
- a tuple of real numbers has 300% overhead
  - compared to a C array of `float` values
  - and 350% for integers
- this causes extremely poor cache use
- integers are arbitrary-precision

Math in Python

265

- numeric data usually means arrays
  - this is inefficient in python
- we need a module written in C
  - but we don't want to do that ourselves
- enter the SciPy project
  - pre-made numeric and scientific packages

The SciPy Family

- numpy: data types, linear algebra
- scipy: more computational machinery
- pandas: data analysis and statistics
- matplotlib: plotting and graphing
- sympy: symbolic mathematics

Aside: External Libraries

- until now, we only used bundled packages
- for math, we will need external libraries
- you can use pip to install those
  - use `pip install --user <package>`

Aside: Installing numpy

- the easiest way may be with pip
  - this would be pip3 on aisa
- linux distributions usually also have packages
- another option is getting the Anaconda bundle
- detailed instructions on https://scipy.org

Arrays in numpy

- compact, C-implemented data types
- flexible multi-dimensional arrays
- easy and efficient re-shaping
  - typically without copying the data

Entering Data

- most data is stored in `numpy.array`
- can be constructed from a `list`
  - a list of lists for 2D arrays
- or directly loaded from / stored to a file
  - binary: `numpy.load`, `numpy.save`
  - text: `numpy.loadtxt`, `numpy.savetxt`

LAPACK and BLAS

- BLAS is a low-level vector/matrix package
- LAPACK is built on top of BLAS
  - provides higher-level operations
  - tuned for modern CPUs with multiple caches
- both are written in Fortran
  - ATLAS and C-LAPACK are C implementations

Element-wise Functions

- the basic math function arsenal
- powers, roots, exponentials, logarithms
- trigonometric (sin, cos, tan, …)
- hyperbolic (sinh, cosh, tanh, …)
- cyclometric (arcsin, arccos, arctan, …)

Matrix Operations in numpy

- `import numpy.linalg`
- multiplication, inversion, rank
- eigenvalues and eigenvectors
- linear equation solver
- pseudo-inverses, linear least squares

Additional Linear Algebra in scipy

- `import scipy.linalg`
- LU, QR, polar, etc. decomposition
- matrix exponentials and logarithms
- matrix equation solvers
- special operations for banded matrices

Where is my Gaussian Elimination?

- used in lots of school linear algebra
- but not the most efficient algorithm
- a few problems with numerical stability
- not directly available in numpy

Numeric Stability

- floats are imprecise / approximate
- multiplication is not associative
- iteration amplifies the errors

```
0.1**2 == 0.01        # False
1 / ( 0.1**2 - 0.01 ) # 5.8·10^17

a = (0.1 *  0.1) * 10
b =  0.1 * (0.1  * 10)
1 / ( a - b ) # 7.21·10^16
```

LU Decomposition

- decompose matrix A into simpler factors
- $PA = LU$ where
  - P is a permutation matrix
  - L is a lower triangular matrix
  - U is an upper triangular matrix
- fast and numerically stable

Uses for LU

278

- equations, determinant, inversion, …
- e.g. $\det(A) = \det(P^{-1}) \cdot \det(L) \cdot \det(U)$
  - where $\det(U) = \prod_i U_{ii}$
  - and $\det(L) = \prod_i L_{ii}$

Numeric Math

279

- float arithmetic is messy but incredibly fast
- measured data is approximate anyway
- stable algorithms exist for many things
  - and are available from libraries
- we often don't care about exactness
  - think computer graphics, signal analysis, …

Symbolic Math

280

- numeric math sucks for 'textbook' math
- there are problems where exactness matters
  - pure math and theoretical physics
- incredibly slow computation
  - but much cleaner interpretation

Linear Algebra in sympy

281

- uses exact math
  - e.g. arbitrary precision rationals
  - and roots thereof
  - and many other computable numbers
- wide repertoire of functions
  - including LU, QR, etc. decompositions

Exact Rationals in sympy

282

```
from sympy import *
a = QQ( 1 ) / 10 # QQ = rationals
Matrix( [ [ sqrt( a**3 ), 0, 0 ],
          [ 0, sqrt( a**3 ), 0 ],
          [ 0, 0, 1 ] ] ).det()
# result: 1/1000
```

numpy for Comparison

283

```
import numpy as np
import numpy.linalg as la
a = 0.1
la.det( [ [ np.sqrt( a**3 ), 0, 0 ],
          [ 0, np.sqrt( a**3 ), 0 ],
          [ 0, 0, 1 ] ] )
# result: 0.0010000000000000002
```

General Solutions in Symbolic Math

284

```
from sympy import *
x = symbols( 'x' )
Matrix( [ [ x, 0, 0 ],
          [ 0, 1, 0 ],
          [ 0, 0, x ] ] ).det()
# result: x ** 2
```

Symbolic Differentation

285

```
x = symbols( 'x' )
diff( x**2 + 2*x + log( x/2 ) )
# result: 2*x + 2 + 1/x

diff( x**2 * exp(x) )
# result: x**2 * exp( x ) + 2 * x * exp( x )
```

Algebraic Equations

286

```
solve( x**2 - 7 )
# result: [ -sqrt( 7 ), sqrt( 7 ) ]

solve( x**2 - exp( x ) )
# result: [ -2 * LambertW( -1/2 ) ]

solve( x**4 - x )
# result: [ 0, 1, -1/2 - sqrt(3) * I/2,
#           -1/2 + sqrt(3) * I/2 ] ; I**2 = -1
```

Ordinary Differential Equations

287

```
f = Function( 'f' )
dsolve( f( x ).diff( x ) ) # f'(x) = 0
# result: Eq( f( x ), C1 )

dsolve( f( x ).diff( x ) - f(x) ) # f'(x) = f(x)
# result: Eq( f( x ), C1 * exp( x ) )

dsolve( f( x ).diff( x ) + f(x) ) # f'(x) = -f(x)
# result: Eq( f( x ), C1 * exp( -x ) )
```

Symbolic Integration

288

```
integrate( x**2 )
# result: x**3 / 3

integrate( log( x ) )
# result: x * log( x ) - x

integrate( cos( x ) ** 2 )
# result: x/2 + sin( x ) * cos( x ) / 2
```

Numeric Sparse Matrices

- sparse = most elements are 0
- available in `scipy.sparse`
- special data types (not `numpy` arrays)
  - do not use `numpy` functions on those
- less general, but more compact and faster

Fourier Transform

- continuous: $\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) \exp(-2\pi i x \xi)\,\mathrm{d}x$
- series: $f(x) = \sum_{n=-\infty}^{\infty} c_n \exp(\frac{i 2\pi n x}{P})$
- real series: $f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \sin(\frac{2\pi n x}{P}) + b_n \cos(\frac{2\pi n x}{P}))$
  - (complex) coefficients: $c_n = \frac{1}{2}(a_n - i b_n)$

Discrete Fourier Transform

- available in `numpy.fft`
- goes between time and frequency domains
- a few different variants are covered
  - real-valued input (for signals, `rfft`)
  - inverse transform (`ifft`, `irfft`)
  - multiple dimensions (`fft2`, `fftn`)

Polynomial Series

- the `numpy.polynomial` package
- Chebyshev, Hermite, Laguerre and Legendre
  - arithmetic, calculus and special-purpose operations
  - numeric integration using Guassian quadrature
  - fitting (polynomial regression)

Part 12: Statistics

Statistics in `numpy`

- a basic statistical toolkit
  - averages, medians
  - variance, standard deviation
  - histograms
- random sampling and distributions

Linear Regression

- very fast model-fitting method
  - both in computational and human terms
  - quick and dirty first approximation
- widely used in data interpretation
  - biology and sociology statistics
  - finance and economics, especially prediction

Polynomial Regression

- higher-order variant of linear regression
- can capture acceleration or deceleration
- harder to use and interpret
  - also harder to compute
- usually requires a model of the data

Interpolation

- find a line or curve that approximates data
- it must pass through the data points
  - this is a major difference to regression
- more dangerous than regression
  - runs a serious risk of overfitting

Linear and Polynomial Regression, Interpolation

- regressions using the least squares method
  - linear: `numpy.linalg.lstsq`
  - polynomial: `numpy.polyfit`
- interpolation: `scipy.interpolate`
  - e.g. piecewise cubic splines
  - Lagrange interpolating polynomials

Pandas: Data Analysis

- the Python equivalent of R
  - works with tabular data (CSV, SQL, Excel)
  - time series (also variable frequency)
  - primarily works with floating-point values
- partially implemented in C and Cython

Pandas Series and DataFrame

- `Series` is a single sequence of numbers
- `DataFrame` represents tabular data
  - powerful indexing operators
  - index by column → series
  - index by condition → filtering

Pandas Example

```
scores = [ ('Maxine', 12), ('John', 12),
           ('Sandra', 10) ]
cols = [ 'name', 'score' ]
df = pd.DataFrame( data=scores, columns=cols )
df['score'].max() # 12
df[ df['score'] >= 12 ] # Maxine and John
```