PV248 Python

Petr Ročkai

October 22, 2020

Part 1: Object Model

Objects

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

Classes

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

Types vs Objects

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

Poking at Classes

- 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__`

Encapsulation

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

Loose Coupling

- 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

Polymorphism

- 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'

Generic Programming

- 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

Attributes

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

Methods

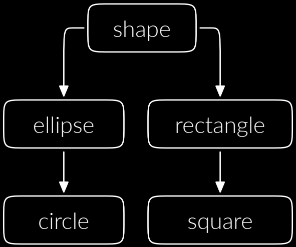
- functions (procedures) tied to objects
- implement the behaviour of the object
- they can access the object (`self`)
- their signatures (usually) provide the interface
- methods are also objects

Class and Instance Methods

- 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`

Inheritance



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

Multiple Inheritance

- 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

Mixins

- 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

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)

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

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)

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

Writing Classes

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

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

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`, ...)

Modules in Python

- 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`

Part 2: Memory Management & Builtin Types

Memory

- 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

Language vs Computer

- 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

Memory Management

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

Memory Management Terminology

- 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

Memory Management by Type

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

Automatic Memory Management

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

Object Lifetime

- 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

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

Dynamic Automatic

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

Reference Counting

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

Mark and Sweep

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

Memory Management in CPython

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

Refcounting Advantages

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

Refcounting Problems

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

Data Structures

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

Building Data Structures

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

Mutability

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

Built-in: int

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

Additional Numeric Objects

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

Built-in: bytes

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

Properties of `bytes`

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

Built-in: `str`

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

Built-in: `tuple`

- an immutable sequence type
  - the number of elements is fixed
  - so is the type of each element
- but elements themselves may be mutable
  - `x = []` then `y = (x, 0)`
  - `x.append(1)` → `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

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

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

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

Resource Acquisition

- 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

- 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`

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

- still safe if `do_stuff` raises an exception

Finalizers

- 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

- `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

Representing Text

- 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

Unicode

- 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, ...

Code Point

- 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

- 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

```python
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

- 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

- 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

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

JSON: Writing

- 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

- `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

```
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

- `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

- 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

```
<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

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

XML Features

- 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

- 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

- 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

- 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

- 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

```
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

- 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

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

NoSQL & Python

- `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

- 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

```
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

- 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
- class objects
- objects with a `__call__` method

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`

```python
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

```python
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

```python
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"
```

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) ]
```

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

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__()`

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)`

```
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
```

Relational Operators

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

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

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)`

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

- 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

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

With Exceptions

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

Exceptions vs Resources

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

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

Using finally

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

- works, but tedious and error-prone

Using `with`

```python
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

- 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

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

Threading

- 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

- 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

- 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

- 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)

- 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

- 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)

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

Execution Stack

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

Variable Capture

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- `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

- 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

- 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)

- `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

- 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`

- 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)

- 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

- 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`

- 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`)

```python
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`

- `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

- 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

- 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

- 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

- 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`

- `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

- 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

- 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

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

Serving Requests

- 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)

- `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

- 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

- `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`

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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`

- 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`

- `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

- 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

- 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

- `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

- 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

- 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

- `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`

- 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

- 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

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

Native Coroutines (Reminder)

- 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

- `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

- 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

- 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`

- 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

- 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

- 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

```
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

```
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)

```
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

```
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

```
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

```python
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

```python
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

```python
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

```python
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`

- 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

```python
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

- 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`

- 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`

```python
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

- 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

- 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

```
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

- 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

- 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

- 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

- 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

- 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

- 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

```python
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

- 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

- 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

- '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

- 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

```
# 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

- 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

- 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

- 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

- 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

- 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`

- 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

```
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

```
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

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

Symbolic Differentation

```
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

```
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

```
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

```
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: $f(x) = \sum_{n=-\infty}^{\infty} c_n \exp(\frac{i2\pi nx}{P})$
- real series: $f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \sin(\frac{2\pi nx}{P}) + b_n \cos(\frac{2\pi nx}{P}))$
  - (complex) coefficients: $c_n = \frac{1}{2}(a_n - ib_n)$

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
```