PV248 Python

Petr Ročkai

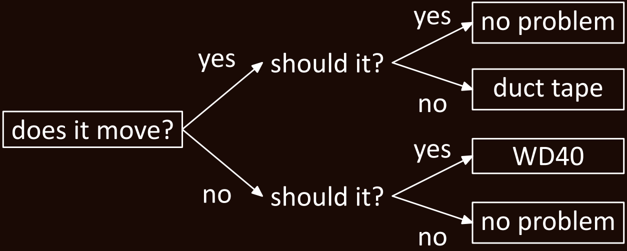
Programming vs Languages

- Python is unobtrusive (by design)
- if you can program, you can program in Python
- there are idiosyncracies (of course)
- but you will mostly get by

Programming vs Jobs

- we all want to write beautiful programs
  - but you didn’t sleep for 2 nights
  - and this thing is going into production tomorrow
- sometimes you get a chance to clean up later
  - and sometimes you don’t

Engineering Flowchart



Python makes for decent duct tape and WD40.

In This Course

- you will not learn to write beautiful programs
- we will try to do things with minimum effort
  - perfect is the enemy of good
- ugly comes in shades
  - you should always write passable code
  - there is a balance to strike

… ugly, cont’d

- there are two main schools of writing software
  - do the right thing
  - worse is better
- https://www.jwz.org/doc/worse-is-better.html

The Right Thing

- simplicity: interface first, implementation second
- correctness: required
- consistency: required
- completeness: more important than simplicity

Worse is Better

- simplicity: implementation first
- correctness: simplicity goes first
- consistency: less important than both
- completeness: least important

Design Schools

- there are pros and cons to both
- right thing is often expensive
- worse is better often wins
- which one do you think Python belongs to?

Disclaimer

- I am not a Python programmer
- please don't ask sneaky language-lawyer questions

Goals

- learn to use Python in practical situations
- have a look at existing packages and what they can do
- code up some cool stuff, have fun

Organisation

- there are 2 standard seminar groups
  - attendance is compulsory (minus 2 absences)
  - one virtual work-at-home group
- the lecture and seminars on 2.10. are cancelled

Coursework

- there will be a set of exercises each week
- you should mostly do these within the seminar
- please make a public `git` (or `hg`) repository
  - we are all adults here – do not copy
  - i will collect the repository addresses

Exercise Grading

- exercises are binary: pass or fail
- you will get 4 chances on each to get right
- failing is the same as missing the deadline

Exercise Deadlines

- 7 days, worth 2 points
- 14 days, worth 1.5 point
- Monday 17.12., worth 1.25 points
- Tuesday 12.2., worth 1 point

Passing the Course

- you can get
  - 24 points for exercises
  - 4 points for seminar attendance
  - 4 points for a small project
- you need 20 points to pass

Stuff We Could Try

- working with text, regular expressions
- plotting stuff with bokeh or matplotlib
- talking to SQL databases
- talking to HTTP servers
- being an HTTP server
- implementing a JSON-based REST API
- parsing YAML and/or JSON data
- … (suggestions welcome)

Some Resources

- `https://docs.python.org/3/` (obviously)
- `https://msivak.fedorapeople.org/python/`
- study materials in IS
- `help()`
- google, stack overflow, …

Part 1: Text & Regular Expressions

Repository Structure

- create a directory for each week
- name them `01-text` and so on
  - the `-text` doesn't really matter
  - scripts will be looking for `01*`
- program names must be exactly as specified

Reading Input

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

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

Regular Expressions

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

Other String Operations

- better whitespace stripping: `s2 = s1.strip()`
- splitting: `str.split(';')`

Dictionaries

- associative arrays: map (e.g.) strings to numbers
- nice syntax: `dict = { 'foo': 1, 'bar': 3 }`
- nice & easy to work with
- can be iterated: `for k, v in dict.items()`

Counters

- `from collections import Counter`
- like a dictionary, but the default value is `0`
- `ctr = Counter()`
- compare `ctr['baz'] += 1` with `dict`

Command Line

- we will often need to process command arguments
- in Python, those are available in the `sys` module
- `import sys`
- arguments are in `sys.argv` (a list)

Exercise 1: Input

- get yourself a git/mercurial/darcs repository
- grab input data (`scorelib.txt`) from study materials
- read and process the text file
- use regular expressions to extract data
- use dictionaries to collect stats
- beware! hand-written, somewhat irregular data

Exercise 1: Output

- print some interesting statistics
  - how many pieces by each composer?
  - how many pieces composed in a given century?
  - how many in the key of c minor?
- bonus if you are bored: searching
  - list all pieces in a given key
  - list pieces featuring a given instrument (say, bassoon)

Exercise 1: Invocation

- `./stat.py ./scorelib.txt composer`
- `./stat.py ./scorelib.txt century`

Exercise 1: Example Output

- Telemann, G. P.: 68
- Bach, J. S.: 79
- Bach, J. C.: 6
- …

For centuries:

- 16th century: 3
- 17th century: 11
- 18th century: 32

Cheat Sheet

`for line in open('file', 'r')`	read lines
`dict = {}`	an empty dictionary
`dict[key] = value`	set a value in a dictionary
`r = re.compile(r"(.*):")`	compile a regexp
`m = r.match("foo: bar")`	match a string
`if m is None: continue`	match failed, loop again
`print(m.group(1))`	extract a capture
`for k, v in dict.items()`	iterate a dictionary
`print("%d, %d" % (12, 1337))`	print some numbers

Part 2: Objects and Classes

Objects

- the basic “unit” of OOP
- they bundle data and behaviour
- provide encapsulation
- make code re-use easier
- also known as “instances”

Classes

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

Poking at Classes

- `{}.__class__`
- `{}.__class__.__class__`
- `(0).__class__`
- `[].__class__`
- compare `type(0)`, etc.
- `n = numbers.Number(); n.__class__`

Types vs Objects

- class system is a type system
- “duck typing”: quacks, walks like a duck
- since python 3, types are classes
- everything is dynamic in python
  - you can create new classes at runtime
  - you can pass classes as function parameters

Encapsulation

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

Weak Coupling

- coupling is a degree of interdependence
- more coupling makes hard to change things
  - it also makes reasoning harder
- good programs are weakly 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 weakly
- 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

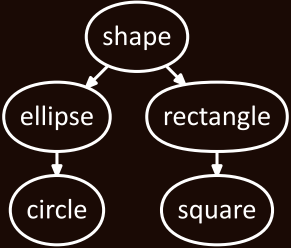
- functions (procedures) tied to objects
- they can access the object (`self`)
- implement the behaviour of the object
- their signatures (usually) provide the interface
- methods are also objects

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

Writing Classes

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

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

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`

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`, …)

Exercise 2: Objects

- create a class hierarchy for printed scores
- define (at least) the folowing classes
  - `Print`, `Edition`, `Composition`, `Voice`, `Person`
- define suitable constructors (`__init__`)
- you can use additional helper classes

Prints, Editions & Compositions

- printed score belongs to an edition
- an edition has an author (an editor)
- edition of is a particular composition
- the composition has an author (composer)
- both editors and composers are people

Voices

- compositions can have multiple voices
- each voice has a range and a name (instrument)
- one or both may be unknown
- ranges are written using a double dash (`--`)

The `Print` class

- attributes
  - `edition` (instance of `Edition`)
  - `print_id` (integer, from `Print Number:`)
  - `partiture` (boolean)
- method `format()`
  - reconstructs and prints the original stanza
- method `composition()` (= `edition.composition`)

The `Edition` class

- attributes
  - `composition` (instance of `Composition`)
  - `authors` (a list of `Person` instances)
  - `name` (a string, from the `Edition:` field, or `None`)

The `Composition` class

- attributes
  - `name`, `incipit`, `key` and `genre` (strings or `None`)
  - `year` (integer if an integral year is given or `None`)
  - `voices` (a list of `Voice` instances)
  - `authors` (a list of `Person` instances)

`Voice` and `Person`

- `Voice` attributes
  - `name`, `range` (strings or `None`)
- `Person` attributes
  - `name` (string)
  - `born`, `died` (integers or `None`)

Exercise 2: Parsing

- write a `load(filename)` function that reads the text
  - this will be the same `scorelib.txt` as before
- the function returns a list of `Print` instances
- the list should be sorted by the print number (`print_id`)

Exercise 2: Module

- the classes should live in `scorelib.py`
- add a simple test script, `test.py`
  - this will take a single filename
  - invocation: `./test.py scorelib.txt`
  - run `load()` on that filename
  - call `format()` on each `Print`, add empty lines

Part 3: Persistent Data

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.

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's and lists
- 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 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

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

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

Aside: 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/`

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
- 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

Exercise 3: Importing Data

- create an empty `scorelib.dat` from `scorelib.sql`
- start by importing composers & editors into the database
  - then continue with scores &c.
- use the classes from previous exercise
  - you can copy & extend them
  - you can also use inheritance or composition

Exercise 3: Database Structure

- defined in `scorelib.sql` (see study materials)
- test with: `sqlite3 scorelib.dat < scorelib.sql`
- you can `rm scorelib.dat` any time to start over
- consult comments in `scorelib.sql`
- do not store duplicate rows

Exercise 3: Requirements

- the structure in `scorelib.sql` is compulsory
- you must use SQLite 3
- parsing proceeds using rules from exercise 2
- each row in each table must be unique
    - special rules for people, see next slide

Exercise 3: Storing People

- the name alone must be unique
- merge `born` and `died` fields
  - `NULL` iff it is `None` in all instances
  - resolve conflicts arbitrarily

Exercise 3: Invocation

- the script should be called `import.py`
- `./import.py scorelib.txt scorelib.dat`
- first argument is the input text file
- second argument is the output SQLite file
  - assume that this file does not exist
  - the script must also set up the schema

SQL Cheat Sheet

- `INSERT INTO table (c1, c2) VALUES (v1, v2)`
- `SELECT (c1, c2) FROM table WHERE c1 = "foo"`

`sqlite3` Cheats

- `conn = sqlite3.connect( "scorelib.dat" )`
- `cur = conn.cursor()`
- `cur.execute( "... values (?, ?)", (foo, bar) )`
- `conn.commit()` (don’t forget to do this)

Part 4: Memory (Data) Model

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, well, 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
  - can contain references to other objects
  - have limited lifespan

Memory Management Terminology

- object: an entity with an address and size
  - not the same as language-level object
- lifetime: when is the object valid
  - live: references exist to the object
  - dead: the object 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

Dynamic Automatic

- over-approximate lifetime dynamically
- usually easiest for the programmer
  - until you need to debug a space leak
- reference counting, mark & sweep collectors

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
- 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`

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 types in Python
  - built-in, implemented in C
  - user-defined (includes libraries)
- both types 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

Exercise 4: Preliminaries

- pull data from `scorelib.dat` using SQL
- print the results as (nicely formatted) JSON
- invocation: `./search.py Bach`
  - the `scorelib.dat` will not be your own
  - you must not use the text data

Exercise 4: Part 1

- write a script `getprint.py`
  - the input is a print number (argument)
  - the output is a list of composers (stdout)
- each composer is a dictionary
- `name`, `born` and `died`

Exercise 4: Part 1 Output

```
$ ./getprint.py 645
[
    { "name": "Graupner, Christoph",
      "born": 1683, "died": 1760 },
    { "name": "Grünewald, Gottfried" }
]
```

Exercise 4: Part 1 Hints

- you will need to use SQL joins
- `select ... from person join score_authors on person.id = score_author.composer ... where print.id = ?`
- the result of `cursor.execute` is iterable

Exercise 4: Part 2

- write a script `search.py`
- the input is a composer name substring
- the output is a list of all matching composer names
  - along with all their prints in the database
- hint: `... where person.name like "%Bach%"`

Exercise 4: Part 2 Output

```
$ ./search.py Bach
{
    "Bach, Johann Sebastian": [
        { "Print Number": 111,
          "Title": "Konzert für ..." , ... },
        { "Print Number": 139, ... }, ...
    ],
    "Bach, Johann Christian": ...,
    ...
}
```

Part 5: Numeric Data

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: 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`

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 from a `list`
  - a list of list 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

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

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

- useful in differential problems and functional analysis
- the `numpy.polynomial` package
- Chebyshev, Hermite, Laguerre and Legendre
- arithmetic, calculus and special-purpose operations

Statistics in `numpy`

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

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
```

Exercise 5: Warm-Up 1

- create a matrix from a list of lists
- compute and print (to stdout)
  - rank and determinant
  - inverse (if applicable)
- all operations are in `numpy.linalg`

Exercise 5: Warm-Up 2

- a simple non-homogeneous linear equation solver
- put the coefficients in a list of lists
- put the constants in a list of numbers
- use `linalg.solve` from `numpy`
- make sure you understand what is going on

Exercise 5: Intro

- ‘nice’ equations, invocation: `./eqn.py input.txt`
- parse a human-readable system of equations
- variables → single letters, coefficients → integers
- only + and − are allowed
- print the solution to stdout (using variable names)

Exercise 5: Unique Solution

- decide a unique solution exists
- if so, print the solution

```
2x + 3y = 5
 x -  y = 0
solution: x = 1, y = 1
```

Exercise 5: No Solution

- print `no solution` if the system is inconsistent

```
x + y = 4
x + y = 5
no solution
```

Exercise 5: Multiple Solutions

- it may also be under-determined
- only print the dimension of the solution space

```
x + y - z = 0
x = 0
solution space dimension: 1
```

Exercise 5: Details

- the right hand side is always a constant
  - and is the only constant term
- print the solution/result to `stdout`
  - solutions come in alphabetical order
- there are spaces around operators and =
  - no space between a coefficient and a variable

Exercise 5: Hints

- `linalg.solve` assumes unique solution
  - you can use Rouché-Capelli to check
- you can obtain a rank with `linalg.matrix_rank`

Part 6: Advanced Constructs

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

```
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]
```

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

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

```
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 are 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

Exercise 6: Fourier Transform

- continuous: $\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) \exp(-2\pi i x \xi)\, \mathrm{dx}$
- series:
  - $f(x) = \sum_{n=-\infty}^{\infty} c_n \exp\left(\frac{i2\pi nx}{P}\right)$
- real series:
  - $f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \sin\left(\frac{2\pi nx}{P}\right) + b_n \cos\left(\frac{2\pi nx}{P}\right)\right)$
  - $c_n = \frac{1}{2}(a_n - ib_n)$

Exercise 6: Signal Basics

- sample rate: number of samples per second
- we process the signal in equal-sized chunks
  - P is the (time) length of the analysis window
  - N is the number of samples
- use non-overlapping analysis windows

Exercise 6: FFT in `numpy`

- `rfft` gives you the c_n of the real series
  - $f(x) = \sum_{n=0}^{N/2} c_n \exp(\frac{i2\pi nx}{P})$
  - $N/2$ because of the Nyquist frequency limit
- we are only interested in amplitudes: $|c_n|$
  - amplitude of a complex number: `numpy.abs`

Exercise 6: Input

- a `.wav` file, PCM, sample rate 8–48 kHz
  - such that it will be accepted by `wave.open`
  - may be stereo or mono, 16 bit samples
- average the channels for stereo input
- ignore the final (incomplete) analysis window
- you can use `struct.unpack` to decode the samples

Exercise 6: Output

- a peak is a frequency component with amplitude $\geq 20a$
  - where a is the average amplitude in the same window
- print the highest- and lowest-frequency peak encountered
  - in the form `low = 37, high = 18000`
  - print `no peaks` if there are no peaks
  - the numbers are in Hz, precision = exactly 1 Hz

Exercise 6: Invocation & Hints

- invocation: `./peaks.py audio.wav`
  - the output goes to `stdout`
  - only a single line for the entire file
- think about how precision relates to N
- generate simple sine wave inputs for testing
  - also a sum of sine waves at different frequencies

Part 7: Advanced Constructs 2, Pitfalls

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.

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

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

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

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

Using `finally`

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

- works, but tedious and error-prone

Using `with`

```
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
```

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 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

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

Exercise 7: Music Analysis

- invocation: `./music.py 440 audio.wav`
  - 440 is the frequency of the pitch a’
  - `audio.wav` is the same as for exercise 6
- use a sliding window for .1 second precision
- print peak pitches instead of frequencies

Exercise 7: Output

```
01.0-02.3 e+0 gis+0 b+0
10.0-12.0 b'+10
12.0-12.7 C+0 e-3
```

- consider only the 3 most prominent peaks
- print 1 line for each segment with the same peaks
  - print nothing for segments with no peaks
  - order the peaks by increasing frequency

Exercise 7: Pitch Formatting

- pitch names: c, cis, d, es, e, f, fis, g, gis, a, bes, b
- octaves (Helmholtz): A,, / A, / A / a / a’ / a’’ and so on
- pitches use a logarithmic scale
  - if a’ is 440 Hz, then a is 220 Hz and A is 110 Hz
- valid pitch examples: fis / Cis / bes’ / Es,

Exercise 7: Pitch Deviation

- not all pitches are exactly ‘right’
  - i.e. they won’t exactly match a named pitch
- cent is 1/100 the distance between semitones
  - remember that this is a logarithmic scale
- print the closest named pitch and the deviation in cents
  - if a’ = 440 Hz, then 448 Hz is a’ + 31 cents
  - likewise, 115 Hz is Bes – 23 cents

Exercise 7: Peak Clustering

- most instruments have complex spectra
  - individual notes are not pure sine waves
- this can lead to peak clustering
  - that is multiple peaks next to each other (1Hz apart)
  - consider only the strongest peak in each cluster
  - if equal, pick the one closer to the center of the cluster

Part 8: Testing, Debugging & 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

```
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`

```
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

```
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)
```

Exercise 8: Statistics

- fetch `points.csv` from study materials
  - each column is one deadline of one exercise
  - each line is one student, cells are points
- an average student has average points in each column
- you can use `pandas` and/or `numpy` if you like

Exercise 8: Bulk Stats

- invocation: `./stat.py file.csv <mode>`
- `<mode>` is one of: `dates`, `deadlines`, `exercises`
- in each mode, list all such entities along with
  - `mean`, `median`, `first` and `last` quartile of points
  - number of students that `passed` (points > 0)
- the output is a JSON dictionary of dictionaries
- date `YYYY-MM-DD`, exercise `NN`, deadline `YYYY-MM-DD/NN`

Bulk Output (`stat.py`)

```
{ "01": { "mean": 1, "median": 1, ... },
  "02": { ..., "passed": 60, ... }, ... }
```

or

```
{ "2018-09-26": { ... "last": 2.5, ... },
  "2018-10-03": { ... "passed": 20, ... },
  ... } }
```

Exercise 8: Individual Stats

- invocation: `./student.py file.csv <id>`
- `<id>` is the student identifier or `average`
- output `mean` and `median` points per exercise
- a number of `passed` exercises and `total` points
- a linear regression for cumulative points in time
  - keys: `regression slope` (intercept is 0)
- expected date to pass the 16 and 20 point marks
  - keys: `date 16` and `date 20`

Per-Student Output (`student.py`)

```
{ "mean": 1.66, "median": 1.5,
  "total": 10, "passed": 6,
  "regression slope": 0.2,
  "date 16": "2018-12-05",
  "date 20": "2018-12-25" }
```

Part 9: Communication, HTTP

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

- only 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`)

```
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 (browser) 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

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

Exercise 9: Forwarding HTTP

- invocation: `./http-forward.py 9001 example.com`
  - listen on the specified port (9001 above) for HTTP
  - use `example.com` as the upstream for `GET`
- for `GET` requests:
  - forward the request as-is to the upstream
  - send back JSON to your client (see next slide)
- for `POST` requests
  - accept JSON data, construct request, proceed as `GET`
  - supply suitable default headers unless overridden

Exercise 9: `GET` Requests

- the reply to the client must be valid JSON dictionary
- send the upstream response code as `code`
  - or `"timeout"` (by default after 1 second)
- send all the received headers to the client
- if the response is valid JSON, include it under `json`
  - include it as a string in `content` otherwise

Exercise 9: `POST` Requests

- read a JSON dictionary from the request content; keys:
  - `type` – string, either `GET` (default) or `POST`
  - `url` – string, the address to fetch
  - `headers` – dictionary, the headers to send
  - `content` – the content to send if `type` is `POST`
  - `timeout` – number of seconds to wait for completion
- if the JSON is invalid, set `code` to `"invalid json"`
  - also if a crucial key is missing (`url`, `content` for `POST`)

```
# POST request content
{ "type": "GET", "url": "http://example.com",
  "headers": { "Accept-Encoding": "...", ... },
  "timeout": 3 }

# reply from http-forward.py
{ "code": 200
  "headers": { "Content-Length": ... },
  "json": ... }
```

Exercise 9: Bonus

- handle SSL/TLS when connecting to your upstream
  - specified by `https` as a protocol in `url`
- include a boolean `certificate valid` in response JSON
  - rely on the default system trusted CA certs
  - also `certificate for` with a list of hostnames
- get 0.5 extra point (regardless of which deadline you pass)

Part 10: Closures, Coroutines &c.

Exercise 10: CGI

- invocation: `./serve.py 9001 dir`
- listen on the specified port (9001 in this case)
- serve the content of `dir` over HTTP
- treat files named `.cgi` specially (see next slide)
- serve anything else as static content

Exercise 10: Running CGI Scripts

- if a `.cgi` file is requested, run it
- adhere to the CGI protocol
  - request info goes into environment variables
  - the stdout of the script goes to the client
  - refer to RFC 3875 and/or Wikipedia
- do not forget to deal with `POST` requests

Exercise 10: Various

- no need to auto-index directories
- you must handle concurrent connections
  - even while a CGI script is running
- you must handle arbitrarily large data
  - this applies to static files
  - but also to CGI script outputs

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 clousre 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 11: `asyncio`, Projects

IO at the OS Level

- often defaults to blocking
  - `read` returns when data is available
  - this is usually OK for file
- 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

Exercise 11: Tic Tac Toe

- write a game server for (3x3) tic tac toe
- invocation: `./ttt.py port`
  - listen on the given port (number)
  - serve HTTP (only GET requests)
  - all responses are JSON dictionaries

Exercise 11: Start

- `GET /start?name=string`
- returns a numeric `id`
  - multiple games may run in parallel
- the game starts with an empty board
- player 1 plays first

Exercise 11: Status

- `GET /status?game=id`
- if the game is over:
  - set `winner` to 0 (draw), 1 or 2
- otherwise set:
  - `board` is a list of lists of numbers
  - 0 = empty, 1 and 2 indicate the player
  - `next` 1 or 2 (who plays next)

Exercise 11: Playing

- `GET /play?game=id&player=1&x=1&y=2`
- must validate the request
- set `status` to either `"ok"` or `"bad"`
  - if `status` is `"bad"`, set `message`
  - `message` is free-form text for the user

Exercise 12: Tic Tac Toe Client

- include `ttt.py` from exercise 11
  - add a `/list` request
  - returns a JSON list of games
  - each is a dict with `name` and `id`
- invocation: `client.py host port`

Exercise 12: User Interface

- start by offering a list of games
  - only offer games with empty boards
- the user enters the numeric id to join
  - joining makes you player 2
- typing `new` starts a new game
  - you start as player 1

Exercise 12: Polling

- ask for status ~once per second
- while waiting, print (once)
  - `waiting for the other player`
- draw an up-to-date board
  - use `_`, `x` and `o`, no spaces

Exercise 12: Gameplay

- prompt with `your turn (o):` (or `x`)
  - read x and y (whitespace separated)
  - if invalid, print `invalid input`
  - then ask again (until satisfied)
- on game over, print `you lose` or `you win`

Exercise 12: Bonus

- make an interactive graphical interface
  - make the interaction mouse-based
  - use `pygame` or `pyglet`
- must be ready for the last seminar
  - you can get 1 extra point

Projects

- you can earn 4 points
  - that's 2 exercises worth
  - the effort should match that
- submit by the end of the exam period
- this is a fallback option
  - exercises and reviews are preferred

Project Grading

- there is only 1 automated option (see DF)
  - can be evaluated repeatedly
- everything else is evaluated manually
  - should work 100% on first try
  - you get at most one retry
  - expect latency of about a week

Project Reviews

- projects can be reviewed before submission
  - excluding the machine-corrected variant
  - you can seek multiple reviews
  - getting at least one is strongly recommended
- otherwise same rules as for exercises
  - review point limits are shared

Project Topics

- do not try to sell something you already have
- seek approval before you start working
  - put a `project.txt` in your repository
  - I will make a note in the IS notebook
- it is okay to come up with your own
  - but I may request changes

Project Idea: Breakout

- write a breakout clone (game)
  - or another game of similar complexity
  - do not settle for absolute bare-bones
  - add simple sound effects or animation
- you can use `pygame` or `pyglet`

Project Idea: Scorelib Redux

- write an editor for the score database
  - should be practically usable
  - work with the SQL representation
- you can use `pyqt5`
- alternatively `flask` or `django`
  - might need some javascript
  - you can also use `aiohttp` and AJAX

Project Idea: A Real Tuner

- should work in real time
- process microphone input
  - alternatively work with a recording
  - in which case, provide a slider
- visualize the outputs
  - try `pygame` or `pyglet`

Part 12: Modules and Packages

Code Modularity

- common tasks are bundled as functions
- functions can be bundled into classes
  - often contains shared state (via attributes)
- classes are bundled into modules
  - simpler than classes: usually no data
- modules can be bundled into packages

Why Modularity

1. managing size and complexity
2. management of names
3. code re-use and sharing

Code Size

- there are natural limits on function size
  - long functions are hard to understand
  - likewise on class sizes
- this also holds for modules
  - big modules are hard to use
  - but even harder to maintain

Naming Things

- human brain is highly context-sensitive
  - same name can refer to many things
  - consider a method called `open`
- there is no optimal length for a name
  - wider scopes require longer names
  - long names in narrow scopes are wasteful

Namespaces

- a hierarchical approach to names
  - use a short name from within the scope
  - use a longer name from outside
- with a built-in mechanism for shortcuts
- realized by classes, modules, packages

Python Modules

- creating a single module is simple
- a collection of re-usable code
  - mainly classes (`class`)
  - and functions (`def`, `async def`)
- there is no special syntax
  - a file, basically the same as a script

Python Packages

- a package is a bundle of modules
- realized as a file system directory
  - it must have an `__init__.py`
  - but it could be empty
- this is what gives us `import foo.bar`

Package Mechanics

- the `__init__.py` has two roles
  - prevent conflicts with non-package directories
  - provide definitions
- `import foo` will load `foo/__init__.py`

More on Import

- `import` loads and evaluates the module
- it creates an object to represent it
- creates a variable in the current scope
- assigns the object to the variable
- `import` is somewhat like `def`

Bytecode

- CPython is actually a bytecode interpreter
- there is a frontend which parses code
  - and emits an intermediate representation
  - which can be stored as bytecode
- bytecode is stored in `.pyc` files
- and for modules, it is cached under `__pycache__`

Modules Written in C

- those are implemented as shared libraries
  - `.so` on UNIX (typically ELF shared object)
  - `.pyd` on Windows (really a PE DLL file)
- the lookup is the same as for `.py` modules
- functions show up as built-in functions

The View from C

- CPython objects are of type `PyObject *`
- C APIs exist to create and use objects
- recall that modules are just objects
- a special function `PyInit_modname()`
  - say `PyInit_spam()` in `spam.so`
  - `import` calls this to create the object

Built-in Modules

- some modules are completely built into CPython
- internally, they are much like C modules
- may be for efficiency or for low-level system access
- the `sys` module is always built-in
  - `sys.path` is needed to load any other modules

Modules are Garbage-Collected

- `sys.modules` holds references to all loaded modules
- it's possible to remove modules from there
- importing again will then reload the module
- the old version can be garbage-collected
- some C modules are excluded from this mechanism

Distributing Packages: Reminder

- python packages are distributed via PyPI
- source trees are different from installed modules
- extra metadata in the source tree
  - info about authors, links to resources
  - most importantly package dependencies

Source Trees

- python is not a compiled language
  - the source code is what is installed
- some packages also contain C code
  - think number crunching in `numpy`
  - this must be actually compiled
- there's also unit tests of course

`setup.py`

- a script that installs your package
- it knows where to put it and how
- also knows how to build C code
- usually written using `setuptools`

Versioning

- so you have made a package…
  - it is probably not complete
  - and it may have some bugs in it
- you add features, fix bugs…
  - other people already use it
  - you need to make a new version

Version Numbers

- often `major.minor` or `major.minor.patch`
  - for example: `python 3.6.5`
- a change in `major` indicates incompatibility
  - like when `print x` no longer works in python 3
- `minor` is for non-breaking feature additions
- `patch` is for bug fixes

Dependencies

- packages are meant for re-use
- so you want to use some package
  - your users will need it too
  - maybe you need a dozen
- sure enough, packages need other packages
  - this is ripe for automation

Dependency Chasing

- `setup.py` could just download dependencies
  - `setuptools` automate this for you
  - and use PyPI to find the packages
- it also only downloads what is missing
- `pip` will find you the ‘toplevel’ package

Versioned Dependencies

- so you use function `bar` from package `foo`
  - but it only appeared in version `2.4`
- so you need package `foo` newer than `2.4`
- but `foo` was then removed in version `3`
  - no time right now to deal with that
- welcome to dependency hell

Chasing Dependencies Redux

- versioning makes dependencies NP-hard
- dependencies may be impossible to satisfy
- mistakes happen with version numbers too
  - those usually affect other packages
- this is a problem in every complex software system

Versioning Strategies

- optimistic dependencies
  - maybe next `foo` major won’t break my code
  - if it does, my package breaks and i must fix it
- defensive dependencies
  - next major of `foo` will probably break my code
  - i use `baz 1.1` and `foo 2.4` and depend on `foo < 3`
  - around comes `baz 1.2` but it needs `foo 3.1`

Questions & (maybe) Answers