

import

Libraries, `import`, and "batteries included"

"Batteries included" is part of the Python philosophy. The intention is to provide lots of useful stuff in the standard library. There are also third-party libraries, such as Numpy and Pandas (we will cover them later). We access libraries using `import`.

The first module worth learning about is `math`, which contains `sqrt`, `sin`, and much more. First, let's observe what happens if we try to use them incorrectly.

```
In [2]: sin(3.0)
```

```
NameError Traceback (most recent call last)
<ipython-input-2-b4bff8c02c99> in <module>
----> 1 sin(3.0)

NameError: name 'sin' is not defined
```

```
In [3]: math.sin(3.0)
```

```
NameError Traceback (most recent call last)
<ipython-input-3-a44dae5ba0f4> in <module>
----> 1 math.sin(3.0)

NameError: name 'math' is not defined
```

Both of these are wrong. Before we can use a library (a module), we have to *import* it as follows.

```
In [4]: import math
print(math.sqrt(16))
print(math.sin(2 * math.pi * 0.5))
print(math.floor(2.5))
print(math.log(math.e**16))
```

```
4.0
1.2246467991473532e-16
2
15.999999999999998
```

As we can see, after `import math`, we can access the functions and constants in the `math` module using dot-notation. Here, `math` is a *module*, whereas in `s.startswith("a")` above, `s` was an *object*. But the same notation is used to access the "members" of a module or an object.

Even after importing, we still can't just write:

```
In [5]: sin(3.0)
```

```
NameError Traceback (most recent call last)
<ipython-input-5-b4bff8c02c99> in <module>
----> 1 sin(3.0)

NameError: name 'sin' is not defined
```

Exercise: calculate $e^{2\sin(2\pi)}$.

```
In [ ]: import math
math.exp(2 * math.sin(2 * math.pi))
```

Exercise: Browse through Python Module of the Week: <https://pymotw.com/3/>

Forms of `import`

```
In [6]: import math
math.sin(2 * math.pi)
```

```
Out[6]: -2.4492935982947064e-16
```

```
In [ ]: from math import sin, cos, sqrt, pi
sin(2 * pi)
```

```
In [ ]: from math import *
sin(2 * pi)
```

```
In [7]: from math import sin as my_favourite_periodic_fn
from math import pi as pie
my_favourite_periodic_fn(2 * pie)
```

```
Out[7]: -2.4492935982947064e-16
```

All of these are common. But

```
from math import *
```

is usually frowned-on for stylistic reasons: it "pollutes" the namespace.

Writing our own modules

When we have written some re-usable code, often we'll package it up in a module of its own.

Exercise Paste our `newton()` code (reproduced below) into a file named `newton.py`. Then in a Jupyter Notebook (or `ipython` or `spyder` if you prefer), `import` it and run it.

```
In [8]: def newton(a, x0, tol=10**-8): # define a new function
```

```
    """Newton's method for finding square roots."""
    x = x0 # initial value
```

```
    while True: # forever
```

```
        print(x)
```

```
        y = (x + a/x) / 2 # update y
```

```
        if abs(x - y) < tol: # x and y arbitrarily close
```

```
            break # exit the infinite loop
```

```
        x = y # update x
```

```
    return x
```

```
newton(64, 1) # call the function ("function invocation")
```

```
1
```

```
32.5
```

```
17.234615384615385
```

```
10.474036101145005
```

```
8.292191785986859
```

```
8.005147977880979
```

```
8.000001655289593
```

```
8.000000000000017
```

```
Out[8]: 8.000000000000017
```

```
In [ ]:
```

Observe a malfunction: that call `newton(64, 1)` runs at `import` time. We could just delete it, but it's nice to include some tests/examples in the file.

The solution is to move that example call into a function, and call it only when the user runs `$ python newton.py` at the command-line (i.e. not via `import`), as follows:

```
def main():
    newton(64, 1)
if __name__ == "__main__":
    main()
```

Here, `__name__` is a special variable which holds a `str`. When a file such as `newton.py` is run directly from the command-line, it will have the value `__main__`.

Exercise: What value will it have when the file is instead `import` ed?

Exercise: change your `newton.py` as suggested. Now if you're in Jupyter Notebook, restart the session using `Kernel -> Restart & Run All`. If you're in `ipython`, just quit and start again. This is necessary because `import newton` by itself would do nothing when `newton` has already been imported, so our changes would not take effect.

Re-do the `import` to convince yourself that the code is not running at `import` time.

The Substitution Model and the Environment Model

The substitution model

Question: how does Python interpret a complex expression?

From the inside out.

An expression is a part of a program, consisting of values, variables, operators, and function calls. It can be simple, like `3`, or complex, like `math.sin(math.sqrt(x+12))`. Anyway, it has a single value (which could be a compound data structure -- but don't confuse a complex expression with a compound data structure). How can we tell what that value is?

You are already familiar with the inside-out *substitution model* from basic arithmetic. To evaluate $\sin(\sqrt{x+12})$: first get the value of x , then $x+12$, then $\sqrt{x+12}$, etc. At each step, we mentally substitute in the value (e.g. 16, if we suppose $x = 4$) in place of the sub-expression (e.g. $x + 12$).

```
math.sin(math.sqrt(x+12))  
  
= math.sin(math.sqrt(16)) (because x = 4)  
= sin(4.0) (because sqrt(16) = 4, and sqrt always returns a floating-point number)  
= -.7568 (because sin(4) = -.7568...)
```

Thus, the expression `math.sin(math.sqrt(x+12))` has the value `-.7568...`.

The substitution model in programming is just an extension of the substitution model in basic arithmetic. The value doesn't have to be a floating-point number as here.

For example: suppose we have two functions, f and g , and we want $f(g(x))$ for some x . Just like in maths, we can then write:

```
z = f(g(x))
```

If it helps, we can think of this as:

```
y = g(x)  
z = f(y)
```

Here's an example using the same concept, but with list access instead of function calls.

```
In [1]: M = [  
    [0, 1, 2, 3],  
    [4, 5, 6, 7],  
    [8, 9, 10, 11]  
]
```

We can access elements as follows:

```
print(M[1][2])
```

What does that expression mean? *Apply the substitution model!*

```
print(M[1][2])
```

is the same as

```
T = M[1] # == [4, 5, 6, 7]  
print(T[2]) # == 6
```

How did I know to take `M[1]` before `[1][2]`?

That is, why is it treated as `(M[1])[2]` as opposed to `M([1][2])`?

This is one of the rules of operator precedence in the language. The `[]` operator binds left-to-right. It has to be this way since `[1][2]` doesn't mean anything on its own!

The substitution model

To evaluate a complex expression:

- Find the *innermost* simple expression, evaluate it, and *substitute it* with the result.
- Repeat until the result is simple.

Namespaces, Scope, and the Environment Model

When the innermost simple expression is a *function call*, the story is the same (evaluate it and substitute the result), but we also have to think about variable scope.

Namespaces

In Python, a *namespace* is an environment in which names ("variables") are mapped to values. For example, within a function:

```
def f(x):  
    y = 3  
    return x + y  
f(2)
```

the name `y` is mapped to an object of type `int` with value `3`. The name `x` is mapped to another object.

"Mapped to" means that when Python is interpreting the code and encounters a name `y`, it can *look it up in the namespace* and find the corresponding value.

So, it's useful to think of a namespace as being a bit like a dictionary:

```
f_namespace = {"x": 2, "y": 3}
```

Every program begins with the *global* namespace. When we call a function, a new *local* namespace is created, nested inside the current namespace.

Objects, functions, and modules all create their own namespaces when they are called/created/imported. E.g. a name can be created inside a function called by a function attached to an object created by a function attached to an object created by a module.

For example, in a file called `math.py` there is an expression `pi = 3.14159...`. (Ok, we are simplifying a bit here.)

`pi` is in the `math` namespace, and becomes accessible as `math.pi`.

Scope

Names are accessible (*in scope*) from outer to inner ("nested") frames, but not the other way around.

```
In [1]: z = 4  
def f(x):  
    y = 3  
    return x + y # z is in scope!  
f(2)
```

Out[1]: 9

```
In [2]: z = 4  
def f(x):  
    y = 3  
    return x + y + z # z is in scope!  
f(2)  
print(y) # y is NOT in scope
```

```
NameError Traceback (most recent call last)  
<ipython-input-2-358a1e62cf3e> in <module>  
      4     return x + y + z # z is in scope!  
      5 f(2)  
----> 6 print(y) # y is NOT in scope
```

```
NameError: name 'y' is not defined
```

Thus, namespaces can "hide" names from each other.

This is essential to creating large programs and re-using code by other people. In particular, it means that I can use `x` as a variable name without fearing a clash with a variable in some function in the `math` library, or in some module my co-worker is writing.

Further reading: https://sebastianraschka.com/Articles/2014_python_scope_and_namespaces.html

How to call a function

- Evaluate the arguments passed-in;
- Create a new namespace with new variables (names given by formal parameters, values given by above evaluation);
- Run the function body;
- If it refers to any names not among the new variables, look in the enclosing namespace (recursively if necessary);
- Return a value and substitute it for the function call in the enclosing namespace;
- Delete the namespace we have created.

Exercise: evaluate this expression:

```
def f(x):  
    return x[1] > 0  
print(19 if f([-2, -3, -4, -2]) else 18)
```

Go to new namespace:

```
x: [-2, -3, -4, -2] # create new name
```

Run the body:

```
return x[1] > 0
```

Substitute

```
return -3 > 0
```

Substitute

```
return False
```

Go back to enclosing namespace:

```
print(19 if f([-2, -3, -4, -2]) else 18)
```

and substitute:

```
print(19 if False else 18)
```

Substitute

```
print(18)
```

By the way, this strange inside-out if-else-statement is Python's way of writing the *ternary expression* which would be written `f() ? 19 : 18` in Java or C.

Mentally simulating the interpreter

- The substitution model (like arithmetic)
- The environment model (namespaces and scope)
- Operator precedence (use parentheses when in doubt)
- Implicit state (e.g. in for-loops).

Mentally simulating the interpreter is a core skill in reading and writing code. It means pretending to "be" the interpreter, and knowing what it knows.

Compound data types

We've seen the basic data types -- integers, floating-point numbers, strings, Booleans. There are also *compound* data types, sometimes called *containers*.

The most basic is the `list`. A list is an ordered collection of values. You can create a list by enclosing values in square brackets, separated by commas.

```
In [1]: primes = [2, 3, 5, 7, 11]
names = ["John", "Paul", "George", "Ringo"]
import math
sines = [math.sin(0.1), math.sin(0.2), math.sin(0.3)] # not just constants
examples = [primes, names, sines] # list of lists also ok
mixed = ["a", "b", 3, sines] # no restriction on types
```

You can add items to a list, delete items, find items, sort the items, etc.

```
In [2]: names.append("Fred")
print(names)
names.remove("Fred")
print(names)
print(names.index("Paul"))
names.sort()
print(names)
del names[2]
print(names)

["John", "Paul", "George", "Ringo", "Fred"]
["John", "Paul", "George", "Ringo"]
1
["George", "John", "Paul", "Ringo"]
["George", "John", "Ringo"]
```

```
In [4]: names = ["John", "Paul", "George", "Ringo"] # reset to original order
```

We can join two lists together, and we can calculate the length of a list with `len()`.

```
In [5]: x = [3, 4, 5] + [9, 10, 11]
print(x)
print(len(x))
```

Iteration with `for`

The `for` keyword introduces a different type of loop. It's always `for x in L`.

`x` becomes the name of a new variable with a different value at each iteration. `L` is a list (or something which, like a list, is a sequence of multiple values).

```
In [5]: for x in names:
    print(x)

John
Paul
George
Ringo
```

`range` is a useful function for use with `for`:

```
In [6]: for x in range(5):
    print(x)

0
1
2
3
4
```

Exercise (Think Python 10.3): Write a function that takes a list of numbers and returns the cumulative sum; that is, a new list where the *i*th element is the sum of the first *i* + 1 elements from the original list. For example, the cumulative sum of `[1, 2, 3]` is `[1, 3, 6]`.

```
In [ ]: def cumulative_sum(L):
    s = 0
    result = []
    for x in L:
        s += x
        result.append(s)
    return result

cumulative_sum([1, 2, 3])
```

Notice a common pattern here: we created an *empty list* `[]`, then gradually `append` ed to it.

Exercise: what would happen if we tried to iterate over the empty list? Guess, then try it.

```
L = []
for i in L:
    print("Hello")
```

Exercise (Think Python Section 10.13 Part 2). Suppose `L` is a list, and you want to append an item `x`.

```
In [ ]: L = [5, 6, 7]
x = 9
```

There are two ways to do it:

```
In [ ]: L.append(x) # preferred
```

```
In [ ]: L = L + [x] # less preferred
```

And four ways not to:

```
L.append([x]) # WRONG!
L = L.append(x) # WRONG!
L + [x] # WRONG!
L = L + x # WRONG!
```

Notice that in some cases you will see an error (a crash), and in others there will be no crash, but it won't do what you wanted/expected.

Useful things to use with `for`

`zip` takes two lists and makes a list of pairs.

We can destructure the pair using `for x, y in zip(...)` instead of `for x in zip(...)`:

```
In [7]: for x, y in zip(names, ["guitar", "bass", "guitar", "drums"]):
    print(x, y)

John guitar
Paul bass
George guitar
Ringo drums
```

`enumerate` takes a single list and returns a list of index-item pairs. Again, we can destructure each pair.

```
In [10]: names
```

```
Out[10]: ['John', 'Paul', 'George', 'Ringo']
```

```
In [12]: for i, x in enumerate(names):
    if i % 2:
        print(i, x)
```

```
1 Paul
3 Ringo
```

Exercise: Given the following lists, combine them using `enumerate` and a *three-way* `zip` to print this:

```
1: John Lennon, guitar
2: Paul McCartney, bass
3: George Harrison, guitar
4: Ringo Starr, drums
```

```
In [ ]: n1 = ["John", "Paul", "George", "Ringo"]
n2 = ["Lennon", "McCartney", "Harrison", "Starr"]
inst = ["guitar", "bass", "guitar", "drums"]
```

```
In [ ]: # destructuring
for i, (a, b, inst) in enumerate(zip(n1, n2, inst)):
    print("%d: %s %s %s" % (i, a, b, inst))
```

Tuples

A tuple is like a list, but enclosed in round brackets, and it is *immutable*: you can't change it!

```
In [16]: t = (4, 5, 6)
t = list(t)
t[1] = 17
t = tuple(t)
print(t)

(4, 17, 6)
```

```
In [ ]:
```

```
In [ ]:
```

Usually we use tuples to mimic rows in databases -- each "field" has potentially different semantics and/or a different type -- whereas we use lists where all the data is of the same type -- more like a column in a database.

Indexing and slicing

Strings, lists, and tuples all have some very useful operations in common. You can get the *n*th element by putting *n* in square brackets:

```
In [18]: L = [5, 6, 7]
L[1]
[10, 12, 12][2]
```

```
Out[18]: 12
```

The first element is element 0:

For convenience, we can count backwards in indices. The last element can be indexed as -1:

```
In [19]: L[-1]
```

```
Out[19]: 7
```

```
In [20]: L[-2]
```

```
Out[20]: 6
```

```
In [21]: L[1] == L[-2]
```

```
Out[21]: True
```

We can use list indices on the left hand side of an assignment as well:

```
In [22]: L[1] = 12
print(L)

[5, 12, 7]
```

Slices

A slice is a sublist (or substring, or sub-tuple). We get a slice using the colon `:`, with two indices indicating the `start` and `end` of the sublist, e.g. `L[1:3]`.

 Slicing in Python

Credit: <http://infohost.nmt.edu/tcc/help/lang/python/docs.html>

We can omit the `start` index, and it defaults to 0, or the `end` index, and it defaults to `len(L)`, or even both!

With slices, it helps to think of an index as pointing "just before" the corresponding position, as shown.

Exercises

- What if you try an index too large for the list?
- Think Python exercise 6.6: write a recursive function `is_palindrome` which works by checking whether the first and last elements of a string are equal, and if so, calls itself on the remainder.

in

The `in` operator checks whether some compound data structure contains a given object. It also works on strings.

```
In [23]: 5 in L
```

```
Out[23]: True
```

```
In [24]: s = "xylophone"
if "ph" in s:
    print("I found ph")
```

```
I found ph
```

Sets

A set is an unordered collection of non-duplicate items. (Contrast to a list, which is ordered and can contain duplicates.)

```
In [25]: names
```

```
Out[10]: ['John', 'Paul', 'George', 'Ringo']
```

```
In [12]: for i, x in enumerate(names):
    if i % 2:
        print(i, x)
```

```
1 Paul
3 Ringo
```

Exercise: Given the following lists, combine them using `enumerate` and a *three-way* `zip` to print this:

```
1: John Lennon, guitar
2: Paul McCartney, bass
3: George Harrison, guitar
4: Ringo Starr, drums
```

```
In [ ]: n1 = ["John", "Paul", "George", "Ringo"]
n2 = ["Lennon", "McCartney", "Harrison", "Starr"]
inst = ["guitar", "bass", "guitar", "drums"]
```

```
In [ ]: # destructuring
for i, (a, b, inst) in enumerate(zip(n1, n2, inst)):
    print("%d: %s %s %s" % (i, a, b, inst))
```

Dictionaries

A dictionary is like a set of keys each paired with a value. It's really useful!

```
In [26]: d = {"name": "Bob"}
d["name"] = "Fred"
d["age"] = 37
print(d)
print(d["age"])
print("name" in d) # "in" looks in the *keys*
print("Fred" in d) # it does not look in the *values*
print("job" in d) # the key "job" doesn't exist, so we'll see a KeyError
```

```
{'name': 'Bob', 'age': 37}
True
False
```

```
-----  
KeyError: 'job' Traceback (most recent call last)  
<ipython-input-26-127afe1567e> in <module>
      6 print("name" in d) # "in" looks in the *keys*
      7 print("Fred" in d) # it does not look in the *values*
----> 8 print(d["job"]) # the key "job" doesn't exist, so we'll see a KeyError
```

```
KeyError: 'job'
```

We should notice a type of regularity/coherence in Python syntax. Lists use square brackets; sets use curly brackets; tuples use round brackets. In all cases the items are separated by commas. A dictionary is a set of key-value pairs, so it uses curly brackets like a set -- but now the items have to be pairs, each pair having a colon `:` in the middle.

Exercises

- Can we use indexing and slicing with strings? With sets? With dictionaries? Why/why not? Try them and observe the result.
- Can we use `len` with lists? With sets? With dictionaries? Why/why not?
- What happens if we say `for x in d`, a dict? Try it.

Dictionary

Consider this complex nested data structure. Describe its structure: "it is a `list` of ... of ... of ... of dicts where each key is a ... and each value is a ..."

```
In [ ]: students = [
    {
        "name": "Bruce Wayne",
        "age": 34,
        "ID": "1234",
        "modules": [
            "CT5123": {
                "grades": [55, 68],
                "attendance": [False, True, True, True, True],
            },
            "CT5234": {
                "grades": [45, 90],
                "attendance": [True, False, False, True, True],
            }
        ]
    },
    {
        "name": "Peter Parker",
        "age": 21,
        "ID": "0126",
        "modules": [
            "CT5123": {
                "grades": [90, 90, 90],
                "attendance": [False, True, True, True, True],
            },
            "CT5234": {
                "grades": [60, 74],
                "attendance": [False, True, True, True, True]
            }
        ]
    }
]
```

Observe that we can access any element using an expression like `students[0]["name"]`, which gets the first student's name. Write a similar expression that picks out Parker's attendance record in CT5123, and then another to get Wayne's most recent grade in CT5234.

Exercise: Why did I write "ID" as a string, not an integer?

```
In [27]: students[0].get("name")
```

```
Out[27]: 'Bruce Wayne'
```

```
In [28]: students[1].get("name")
```

```
Out[28]: 'Peter Parker'
```

```
In [29]: students[0].get("modules").get("CT5123").get("attendance")
```

```
Out[29]: [False, True, True, True, True]
```

```
In [30]: students[1].get("modules").get("CT5123").get("grades")
```

```
Out[30]: [90, 90, 90]
```

```
In [31]: students[0].get("modules").get("CT5234").get("grades")
```

```
Out[31]: [60, 74]
```

```
In [32]: students[1].get("modules").get("CT5234").get("attendance")
```

```
Out[32]: [False, True, True, True, True]
```


Exercise: Autonomous driving in a grid world

"...the most fundamental idea in programming: The evaluator, which determines the meaning of expressions in a programming language, is just another program ... we can regard almost any program as the evaluator for some language" -- SICP,

https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book-Z-H-25.html#_chap_4

In this exercise, we are going to write a function which, depending on your point of view, either:

- processes some input data
- acts as an interpreter of an input program.

- Grid world, `xsize` x `ysize` in size
- Car starts at (0, 0)
- Controlled by program `prog`, e.g. `prog = "NNNESS"`
- `simulate(prog, xsize, ysize)` : how many cells visited, how many time-steps used
- `plan(xsize, ysize)` : create a suitable `prog`.

This exercise is intended to help us get used to thinking about taking action in response to inputs, and also how to choose appropriate data structures for problems. By the way, despite the name this is just a simple game -- no connection to real-world self-driving cars.

Suppose we are in charge of running a mapping service like Google Maps and we want our car to travel around a city, taking photographs as it goes. We want it to cover the entire city.

To simplify the problem, let's suppose this is a city like Manhattan, arranged in a nice grid. Let's suppose the junction at the south-west corner is (0, 0), and the grid is `xsize` units from west to east, and `ysize` units from south to north. Our car will start at (0, 0).

Our car will be controlled by a simple program, consisting of a string of letters (N, S, E, W). For example, "NNNE" means to move north three times and then east once. If any step in the program would drive the car off the grid and into the sea, the car is smart enough to just ignore that step and not move.

We need two functions.

The first function will be called `simulate`. Given a program, it should simulate driving around the grid under control of the program, and tracking the number of junctions visited and the number of time-steps required.

Now for a given grid, for example a `3x4` grid, it would be easy to write out a program which efficiently traversed the entire grid in the minimum number of steps. However, for a large grid, this would be a pain. We need to write a function called `plan` which, given the "city limits" (ie `xsize` and `ysize`), will return the optimum program.

When we run `plan` and feed its output to `simulate`, we should find that our car has visited all junctions (`xsize` x `ysize`) and used no more steps than needed (`xsize * ysize - 1`).

Docstrings

To help us get started, templates for the two functions are provided below.

```
In [1]: def simulate(prog, xsize, ysize):
    """Given a "program" *prog*, run that program by "moving around" and
    tracking what happens.

    Return the number of junctions visited and the number of
    time-steps used.

    Examples of usage and results:

    >>> simulate("NNN", 100, 100)
    (4, 3)
    >>> simulate("NNNSSS", 100, 100)
    (4, 6)
    >>> simulate("NNNESS", 100, 100)
    (8, 7)
    >>> simulate("NNNNNNSSSS", 3, 3)
    (3, 10)
    >>> simulate(plan(10, 10), 10, 10)
    (100, 99)
    """

    return 0, 0 # TODO REPLACE WITH YOUR CODE
```

```
In [2]: def plan(xsize, ysize):
    """Given the city limits, return a program which moves around the
    entire city. The program should be a string, eg "NNESS".

    By the way, notice that doctest will insist that the "expected"
    string, below, must be in single-quotes (''), not double-quotes
    ("").

    Our plan is: starting from the southwest, we traverse all the way
    northwards, then one step east, then all the way southwards, then
    one step east, and so on.

    >>> plan(2, 2)
    'NES'
    >>> plan(2, 3)
    'NNESS'

    """

    return "" # TODO REPLACE WITH YOUR CODE
```

Notice that each function starts with a multi-line comment, enclosed with triple-quotes `""" ... [multiple lines] ... """`. This is called a *docstring*. We should write one for each function we create. Python documentation for libraries is usually created by automatically extracting the docstrings and formatting them for the web, e.g.:

<https://docs.python.org/3/library/collections.html#collections.defaultdict>.

Doctests and test-driven development

Notice we can add "doctests" to the docstring. Each doctest consists of a Python prompt `>>>`, then a function call, followed by the expected output on the next line.

When we run the cell with `doctest.testmod()` at the bottom of the notebook, the `doctest` module will run any doctests it detects in docstrings in the notebook. In this case, it will call `simulate` and `plan` several times as specified in their docstrings, and check that the output matches the expected output.

```
In [3]: import doctest
doctest.testmod()

*****
File "__main__", line 13, in __main__.plan
Failed example:
    plan(2, 2)
Expected:
    'NES'
Got:
    ''
*****
File "__main__", line 15, in __main__.plan
Failed example:
    plan(2, 3)
Expected:
    'NNESS'
Got:
    ''
*****
File "__main__", line 10, in __main__.simulate
Failed example:
    simulate("NNN", 100, 100)
Expected:
    (4, 3)
Got:
    (0, 0)
*****
File "__main__", line 12, in __main__.simulate
Failed example:
    simulate("NNNSSS", 100, 100)
Expected:
    (4, 6)
Got:
    (0, 0)
*****
File "__main__", line 14, in __main__.simulate
Failed example:
    simulate("NNNESS", 100, 100)
Expected:
    (8, 7)
Got:
    (0, 0)
*****
File "__main__", line 16, in __main__.simulate
Failed example:
    simulate("NNNNNNSSSS", 3, 3)
Expected:
    (3, 10)
Got:
    (0, 0)
*****
File "__main__", line 18, in __main__.simulate
Failed example:
    simulate(plan(10, 10), 10, 10)
Expected:
    (100, 99)
Got:
    (0, 0)
*****
2 items had failures:
  2 of   2 in __main__.plan
  5 of   5 in __main__.simulate
***Test Failed*** 7 failures.
```

Out[3]: `TestResults(failed=7, attempted=7)`

If all the outputs are correct, it won't print anything! If not, it will tell you, eg "1 items had failures" and give details. For more on doctests, see the official documentation: <https://docs.python.org/3/library/doctest.html>

In *test-driven development* (TDD), developers work by writing their tests first (with function names), expecting them to fail, and then working on making them pass one by one. In our case, if we run the doctests immediately, we'll see that all tests fail, because `simulate` and `pass` don't do anything correct.

Exercises:

- In your own notebook, execute all code cells, and observe the failing tests.
- Then write the functions `simulate` and `plan` and re-run the tests.

Functions

Let's take a quick look at some aspects where Python functions differ from some other languages.

Default parameters and keyword args

When defining a function we can supply a *default parameter*, that is a default value for a parameter which is used if the caller doesn't pass anything.

```
In [5]: def greet(name, greeting="Hello"):
    print("%s, %s" % (greeting, name))
greet("James", "Hi")
greet("James")
```

Hi, James

Hello, James

Default arguments are *keyword arguments*: they can be passed in any order (after the non-keyword arguments), but each must be passed together with the name ("keyword") that's used inside the function definition.

```
In [6]: def greet(name, greeting="Hello", exclamation="."):
    print("%s, %s%s" % (greeting, name, exclamation))
# notice the last two args are in the "wrong" order, but it works ok:
greet("James", exclamation="!!", greeting="Hi")
```

Hi, James!!

None

If a function doesn't return anything, then it implicitly returns the special null value called `None`:

```
In [9]: x = greet("James")
print("Here is x:", x)
type(x)
```

Hello, James.

Here is x: None

Out[9]: `NoneType`

A function can return multiple values, which is very handy:

```
In [12]: def max_argmax(L):
    maxv = -float("inf")
    maxi = -1
    for i, x in enumerate(L):
        if x > maxv:
            maxv = x
            maxi = i
    return maxv, maxi
a, b = max_argmax([4, 5, 6])
print(a, b)
```

6 2

```
In [13]: max_argmax([5, 6, 7, 10, 3, 4, 9])
```

Out[13]: (10, 3)

In fact, what is really happening here is that the function is returning a *compound data structure*, specifically a *tuple*.

Lambda expressions

`lambda` is a special syntax for defining small functions "inline" (eg inside another expression), and "anonymously" (without giving them names). The syntax is:

```
lambda x: x**2
```

That function has no name! If we assigned it to a variable `sq`:

```
sq = lambda x: x**2
```

then this would be exactly equivalent to:

```
def sq(x):
    return x**2
```

But using `lambda` in that way would be pointless -- it would be better to just use `def`.

```
In [15]: max(range(-20, 10), key=lambda x: x**2)
```

Out[15]: -20

However, sometimes we do want to write a new, anonymous function inside another expression. Functions like `sort`, `min` and `max` accept a `key` argument, which should be a function that returns a single number to be used for sorting:

Exercise: what does the following mean? Apply the substitution model!

```
(lambda x: x**2)(4)
```

Call-by-value, call-by-reference, and copies of lists

If we have studied other languages such as C and Java, we may be familiar with the terms *call-by-value* and *call-by-reference*. If not, we'll explain them now. Python uses a mixture of both.

In *call-by-value*, the idea is that when we pass something to a function, it is the *value* that goes in. If we pass in a variable and change its value inside the function, that doesn't affect its value outside the function:

```
In [16]: def f(x):
    x += 1
    print(x)
a = 3
f(a)
print(a)
```

4

3

By the way, if we want `f` to change the value of a parameter, then:

1. We should try *not to want things like that*

2. Usually better for `f` to *return* a new value:

```
In [17]: def f(x):
    x += 1
    print(x)
    return x
a = 3
a = f(a)
print(a)
```

4

4

Case closed?

No: immutable objects are (effectively) call-by-value, mutable objects are call-by-reference.

Ok, so Python is *call-by-value* as shown. Case closed? No: in fact, this is true only for *immutable* objects, which include the primitive types `int`, `float`, `str`, and `tuple`. Other objects including `list`, `dict`, and objects created by `class` (which we will see later), are mutable and are *call-by-reference*.

```
In [18]: def f(L):
    L.append(1)
    print(L)
M = [4, 5, 6]
f(M)
print("M", M)
```

L [4, 5, 6, 1]

M [4, 5, 6, 1]

As we can see, `L` has been changed inside `f` and the change has propagated to the `M` outside `f`. (We are using distinct variable names `L` and `M` to emphasise that they don't have to have the same name for this to happen.)

User-created objects are mutable too. We'll demonstrate this even though we haven't learned how classes work yet.

```
In [19]: def f(c):
    c.a += 1
class C:
    def __init__(self, a):
        self.a = a
    def __str__(self):
        return "C(%d)" % self.a
c = C(3)
f(c)
print(c)
```

C(4)

It might seem strange that a container like `str` is immutable, but it's true:

```
In [20]: s = "abc"
s[1] = "z"
```

```
----- Traceback (most recent call last)
<ipython-input-20-1e8d6214a068> in <module>
      1 s = "abc"
----> 2 s[1] = "z"

TypeError: 'str' object does not support item assignment
```

Functions which look like they change a string actually make and return a new one. E.g. here, the `replace` does not change `s`.

```
In [22]: s = "abc"
s.replace("a", "z")
print(s)
t = s.replace("a", "z")
print(t)
```

abc

zbc

What we have seen about mutable objects passed to functions is also important when it comes to *copying*.

```
In [23]: L = [4, 5, 6]
M = L
M.append(7)
print(L)
```

[4, 5, 6, 7]

The code `M = L` did not copy `L`. It just made a new name `M` and pointed it to the existing list object.

Example: histograms

As we know, a *histogram* is a graph representing the frequency of occurrence of data, segregated into categories. If the data are real numbers, the categories are *bins*, e.g. [0, 10], [10, 20], [20, 30]. If the data are discrete, then the data values may be the categories, e.g. A, B, C.

In fact, a histogram is really the underlying data structure -- a number representing frequency of occurrence, for each category -- not the graphical representation of it.

In this notebook we'll develop a histogram and refine it a bit, illustrating several useful Python features and good Python style.

```
In [1]: def histogram(s):
    h = {} # represent histogram as a dict
    for c in s: # assume s is iterable
        if c in h:
            h[c] += 1 # increment
        else:
            h[c] = 1 # create key
    return h

In [2]: # test our histogram out
histogram([31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31])

Out[2]: {31: 7, 28: 1, 30: 4}
```

Duck typing

One nice feature of Python is *duck typing*. "If it looks like a duck, and walks like a duck, and quacks like a duck, then treat it like a duck".

Contrary to some misunderstandings, Python objects do have types and these types never change. But a variable is really a *name* which points to an object. A variable can point to one object, and later on in the same program, to a different object of a different type. So *objects* have permanent types but *names* don't.

This becomes very useful when we write functions like `histogram(s)`. Here, the object referred to by the name `s` can be any iterable type. We don't care whether it's a string, list, tuple, or some other type we've never even heard of. As long as it knows how to iterate ("quacks like a duck"), then our code `for c in s` will work ok.

```
In [3]: histogram("mississippi") # can iterate over a string

Out[3]: {'m': 1, 'i': 4, 's': 4, 'p': 2}
```

`collections.defaultdict`

A common pattern occurred above: we had to use an `if-else` (four lines) to take care of the special case where the key didn't already exist.

```
if c in h:
    h[c] += 1 # increment
else:
    h[c] = 1 # create key
```

This pattern is so common that they created a special `dict`-like type, `collections.defaultdict`, which can make our definition a bit nicer. When creating it, we pass in a function (of no arguments) which will be called to create a default when needed.

```
In [4]: from collections import defaultdict
def histogram(s):
    h = defaultdict(int) # int() -> 0
    for c in s: # assume s is iterable
        h[c] += 1
    return h
```

```
In [5]: int() # what does int mean as a function?
```

```
Out[5]: 0
```

Canonicalisation

Suppose we wanted to use our histogram to count letter frequencies in a large corpus.

```
In [6]: s = "It was the best of times, it was the worst of times...""
print(histogram(s))

defaultdict(<class 'int'>, {'I': 1, 't': 8, ' ': 11, 'w': 3, 'a': 2, 's': 6, 'h': 2, 'e': 5, 'b': 1, 'o': 3,
'f': 2, 'i': 3, 'm': 2, ',': 1, 'r': 1, '.': 3})
```

Look what happened: the count for `I` and for `i` are separate!

How to fix this? One solution is to *canonicalise* the data, that is for any data which can occur in multiple forms, map them all to a single, "canonical", form.

```
In [7]: def histogram(s):
    h = defaultdict(int) # int() -> 0
    for c in s: # assume s is iterable
        c = c.lower() # canonicalise. doesn't change s
        h[c] += 1
    return h
```

```
In [8]: print(histogram(s))
```

```
defaultdict(<class 'int'>, {'i': 4, 't': 8, ' ': 11, 'w': 3, 'a': 2, 's': 6, 'h': 2, 'e': 5, 'b': 1, 'o': 3,
'f': 2, 'm': 2, ',': 1, 'r': 1, '.': 3})
```

This works, but now our function is less general than before: it assumes that the elements of `s` are strings, or at least they quack like strings (they have a `.lower()` method). We can fix our problem and still retain generality like this:

```
In [9]: def histogram(s, canonicalise=None):
    h = defaultdict(int) # int() -> 0
    for c in s: # assume s is iterable
        if canonicalise:
            c = canonicalise(c)
        h[c] += 1
    return h
```

```
In [10]: def canonicalise_case(s): return s.lower()
print(histogram(s, canonicalise=canonicalise_case))

defaultdict(<class 'int'>, {'i': 4, 't': 8, ' ': 11, 'w': 3, 'a': 2, 's': 6, 'h': 2, 'e': 5, 'b': 1, 'o': 3,
'f': 2, 'm': 2, ',': 1, 'r': 1, '.': 3})
```

This is neat, because our function becomes much more *combinable* -- more *lego-like*. For example:

```
In [11]: histogram([17.3, 17.4, 19.1, 19.2, 20.5, 20.6, 20.7],
                 canonicalise=round)
```

```
Out[11]: defaultdict(int, {'17': 2, '19': 2, '20': 1, '21': 2})
```

Normalisation

Mapping numerical data from one range to another:

- 0-1 Normalisation: map $[a, b]$ to $[0, 1]$
- -1 to 1: map $[a, b]$ to $[-1, 1]$
- z-scores: map so that data has mean 0 and variance 1
- vector-length normalisation: map so that resulting data, treated as vector, has length 1
- map to probabilities: map so that sum of resulting vector is 1

Normalisation means a few different things (sometimes the same as canonicalisation!), but one possible meaning is to map numerical data from by dividing by the sum, so that the new sum is 1. In a histogram, that means that instead of counting *occurrences*, we will count *frequencies*. We can implement this as below.

```
In [12]: def histogram(s, normalise=False, canonicalise=None):
    h = defaultdict(int) # int() -> 0
    for c in s: # assume s is iterable
        if canonicalise:
            c = canonicalise(c)
        h[c] += 1
    if normalise:
        total = len(s)
        for c in h:
            h[c] /= total # normalise
    return h
```

```
In [13]: histogram("mississippi", normalise=True)
```

```
Out[13]: defaultdict(int,
{'m': 0.09090909090909091,
'i': 0.36363636363636365,
's': 0.36363636363636365,
'p': 0.18181818181818182})
```

As before, we are able to add new functionality but retain generality, because this normalisation is optional. We are taking advantage of Python's optional keyword arguments.

By the way, it is the common idiom to use a `bool` argument to switch behaviours on or off (like `normalise`), but to provide `None` as the default value for an optional *function* (like `canonicalise`).

Sampling from a histogram

To sample from a histogram means to choose one of the keys with probability weighted by the count.

A common idea in AI is to learn a distribution from data and then sample from it. If the data is purely numerical, then of course that is familiar to us in statistics. The data could also be characters, words, tuples, or something else. Our histogram function can "learn" the distribution of any of these. But how can we then sample from it?

Recall that in a (normalised) histogram, the frequencies will sum to 1. Imagine the interval from 0 to 1 divided up into slots of different lengths. In the following algorithm, we choose a random value $r \in [0, 1]$ and see which slot it falls into:

```
[ m |   s   |   i   |   p   ]
  0           1
  [           r           ]
```

```
In [14]: import random
def hist_sample(h):
    # we will assume h is normalised, and so sum(h.values()) == 1
    r = random.random() # in [0, 1]
    accum = 0
    for c in h:
        accum += h[c]
        if accum >= r:
            return c
    raise ValueError
```

```
In [15]: h = histogram("mississippi", normalise=True)
for i in range(10):
    print(hist_sample(h))
```

```
s
s
m
s
i
s
p
i
i
i
```

We can now generate individual letters and they'll be in the right frequencies. If we "learn" from a large corpus of English text, we'll see `e` as the most common letter.

But we might prefer to generate text at the *word level*. We can do it easily! We need to split our input sequence up into tokens and get rid of any punctuation.

```
In [16]: s = "It was the best of times, it was the worst of times..."
```

```
def canonicalise_word(w):
    return w.lower().strip(".,?")
```

```
h = histogram(s.split(), normalise=True, canonicalise=canonicalise_word)
```

```
for i in range(10):
    print(hist_sample(h), end=" ")
```

```
the times of of times was worst times of
```

The text is still nonsense, of course. *n*-grams is one technique which could be used to make it a bit more realistic.

File input/output

So far we have processed very small amounts of data. Let's process a whole book. We can get one in plain text from Project Gutenberg.

We'll use some shell commands to do so. We can execute shell commands directly in a Jupyter Notebook, using the `! !` prefix. If you don't have `wget` on your system, don't worry -- just download the file manually and put it in the current directory (if you are running in Spyder, you might need to tell Spyder to change the current directory also).

```
In [17]: !wget https://www.gutenberg.org/files/98/98-0.txt
!mv 98-0.txt data/tale.txt
```

```
--2019-09-19 08:23:28-- https://www.gutenberg.org/files/98/98-0.txt
Resolving www.gutenberg.org... 152.19.134.47
Connecting to www.gutenberg.org|152.19.134.47|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 804335 (785K) [text/plain]
Saving to: '98-0.txt'
```

```
98-0.txt      100%[=====] 785.48K  1.10MB/s    in 0.7s
```

```
2019-09-19 08:23:29 (1.10 MB/s) - '98-0.txt' saved [804335/804335]
```

```
In [18]: fname = "data/tale.txt"
s = open(fname, "r").read() # notice utf8 addition!
```

```
h = histogram(s.split(), normalise=True, canonicalise=canonicalise_word)
```

```
for i in range(10):
    print(hist_sample(h), end=" ")
```

```
imposing passed of the paris i three be being it
```

Above, we used `open()` to open the file for reading. We then used `.read()` to actually read it -- all at once. An alternative is to use:

```
f = open(fname) # returns a File object
```

```
for line in f: # iterate over its lines
```

```
    # process line somehow...
```

To illustrate file *output*, let's generate some text and write it out. We have to pass the `w` (write) flag to `open`:

```
In [19]: fname = "data/tale_generated.txt"
g = open(fname, "w")
```

```
for i in range(100): # 100 lines of 50 words each
```

```
    output = []
```

```
    for j in range(50):
```

```
        output.append(hist_sample(h))
```

```
        g.write(" ".join(output) + "\n") # write some text, full-stop, and newline
```

```
g.close()
```

Of course, there are many more options for file input/output, and of course file output can be combined with the usual Python

string formatting.

Unpacking

Tuple unpacking

We have already seen that when we return multiple items from a function, they are packed up into a tuple, and then unpacked by the caller, for example:

```
In [43]: def count_punct(s):
    # notice single-quote!
    return s.count("."), s.count(","), s.count('\'')
s = "It was the best of times, it was the blurst of times."
np, nc, nq = count_punct(s)
```

This is actually a special case of a generic mechanism in Python called *unpacking*. It works with tuples and dictionaries.

```
In [44]: print(np, nc, nq)
```

```
1 1 0
```

It doesn't only work in `return`. Here's an example showing how to swap two items using unpacking:

```
In [45]: a = 10
b = 20
a, b = b, a
```

```
In [46]: print(a, b)
```

```
20 10
```

We can use a "wild-card" `*` when unpacking, as follows. This is a common pattern in list-processing - equivalent to `car` and `cons`, for Lisp fans.

```
In [7]: L = [5, 6, 7, 8, 9]
head, *rest = L
print(head)
print(rest)
```

```
5
[6, 7, 8, 9]
```

Notice that `head` is a single item, the first item or the "head" of the list, whereas `rest` gets everything else, so it's a list. Notice that `*rest` is used when unpacking, but the variable that is created is just called `rest`.

Exercise: suppose `L = [5]`. What values will `head` and `rest` have?

Tuple packing in function arguments

The opposite also exists. This "packing" is the basis of two mechanisms for variable-length argument lists in Python functions, called `*args` and `**kwargs`.

First, notice that - surprisingly? - this works:

```
In [10]: print(max(4, 5))
print(max(4, 5, 6))
print(max(4, 5, 6, 7))
```

```
5
6
7
```

Here, `max` takes a variable number of arguments. How can we program a function like that?

```
In [48]: def max(*args): # override the builtin `max`
    # will raise error if len(args) == 0
    result = args[0]
    for arg in args:
        if arg > result:
            result = arg
    return result
```

```
In [32]: print(max(4, 5))
print(max(4, 5, 6))
print(max(4, 5, 6, 7))
```

```
5
6
7
```

What we have seen is that `*` attached to a function parameter name allows multiple arguments to be packed into that parameter. It becomes a tuple:

```
In [13]: def type_test(*args):
    print(type(args))
type_test(4, 5, 6)

<class 'tuple'>
```

Dict packing in function arguments

A similar mechanism is available for keyword arguments. In this case, multiple parameter name-value pairs are packed into a `dict`. Here's a contrived example:

```
In [14]: def f(**kwargs):
    for k in kwargs:
        if k.startswith("_"):
            print(k, kwargs[k])
f(a=1, b=2, _c=3)
```

```
_c 3
```

This mechanism is used a lot in large libraries like Matplotlib (for plotting). See, e.g., https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.scatter.html.

```
matplotlib.pyplot.scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, edgecolors=None, *, plotnonfinite=False, data=None, **kwargs)
```

This API already has a lot of arguments, but there are many more argument relating to `Container` which the `scatter` function will not use explicitly, but will only pass on to `Container` sub-functions. So instead of writing `scatter()` with all those extra arguments, they are just anonymised and shortened as `**kwargs`. This is great: if the `Container` API changes, we don't have to update our `scatter` function at all.

By the way, above we also see a bare `*`. That says that everything after `*` can only be passed as keyword arguments. There is a similar `/` also. It marks the end of positional-only arguments, ie arguments which must not be passed as keyword arguments. There is quite a bit of complexity here!

Tuple and dict unpacking at call time

Unpacking can also be useful when *calling* a function. In this example, we have a function which expects individual arguments, but our data is already packed up. We unpack on the fly. This is just one example where Python makes it easy to "plug in" to an API which doesn't quite fit our data.

```
In [24]: def f(a, b, c, d):
    print(a + b + c + d)
ab = (1, 2)
cd = {"c": 3, "d": 4}
f(*ab, **cd) # unpack on the fly
```

```
10
```

Once again, we can also use the unpacking syntax in other contexts, not just function APIs. This example merges two dictionaries by unpacking them into a new dictionary:

```
In [19]: d1 = {"a": 1, "b": 2}
d2 = {"a": 7, "c": 3, "d": 4}
d = {**d1, **d2}
```

think about it first, then confirm by trying it.

Comprehensions and Generators

Python 3 aims to allow quite lego-like programming - small objects, flexibly combinable. Comprehensions and generators are two nice tools for this.

Comprehensions

A comprehension is a very handy syntax for transforming and slicing-and-dicing lists, sets and dicts to make new ones.

List comprehensions

The list comprehension syntax is like an inverted for-loop inside square brackets:

```
In [1]: # make a list which we will use in following examples.
L = [0, 1, 2, 3, 4]
Out[1]: [0, 1, 2, 3, 4]
In [2]: M = [x**2 for x in L]
Out[2]: [0, 1, 4, 9, 16]
Here, the x**2 is an expression which will be evaluated for each value of x, and the results will be collected to make the new list.

```

Now `M` is a list

```
In [3]: type(M)
Out[3]: list
In [4]: print(M)
Out[4]: [0, 1, 4, 9, 16]
The above code is exactly equivalent to:
```

```
In [5]: M = []
for x in L:
    M.append(x**2)
However, the list comprehension is shorter, faster, and (subjectively) more readable.
```

The list comprehension syntax is intended to parallel the "set builder" notation which is familiar in mathematics, eg

```
{ $x^2 \mid x = 0, \dots, 4\}$ }
```

Filtering

We can filter out items using if-statements inside the comprehension:

```
In [6]: M = [x**2 for x in L if x % 2 == 0]
print(M)
Out[6]: [0, 4, 16]
The above code is exactly equivalent to:
```

```
In [7]: M = []
for x in L:
    if x % 2 == 0:
        M.append(x**2)
You can have double for-loops, or even more -- but it becomes less readable than the equivalent for-loop, and should be used sparingly.
```

```
In [8]: N = [[1, 2, 3], [4, 5, 6]] # nested lists
P = [x**2 for M in N for x in M]
```

Note that `for x in M` in `N` instead (ie putting the `for` parts the other way around) will crash. To see why, try writing out the equivalent double for-loop. Which has to go first, the `for M in N` or the `for x in M`?

Set and dict comprehensions

There are also set comprehensions and dict comprehensions that do what you expect. The set syntax is `{}` instead of `[]`:

```
In [9]: s = {x**2 for x in L}
type(s)
print(s)
Out[9]: {0, 1, 4, 9, 16}
The dict syntax is again {}, but with key: value pairs inside it instead of single items:
```

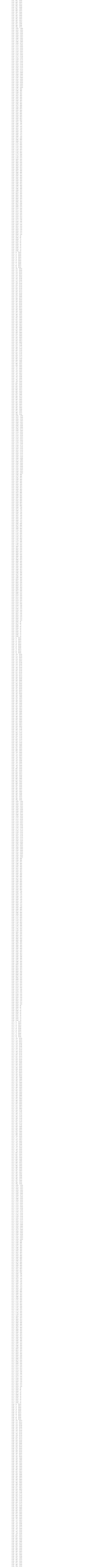
```
In [10]: d = {x: x**2 for x in L}
type(d)
print(d)
Out[10]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
Reminder: you can loop over the keys of a dict with for:
```

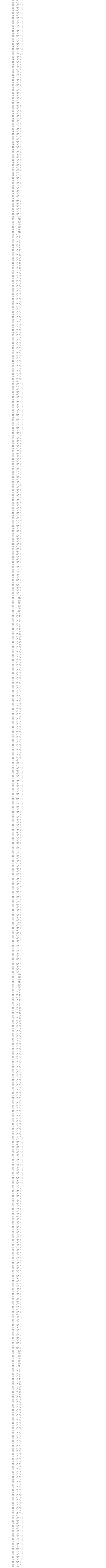
```
In [11]: for k in d:
    print(d[k])
Out[11]: 0
1
4
9
16
121
144
169
196
225
256
289
324
361
400
441
484
529
576
625
676
729
784
841
900
961
1024
1089
1156
1225
1296
1369
1444
1521
1600
1681
1764
1849
1936
2025
2116
2209
2304
2401
2500
2601
2704
2809
2916
3025
3136
3249
3364
3481
3600
3721
3844
3969
4096
4225
4356
4489
4624
4761
4900
5041
5184
5329
5476
5625
5776
5929
6081
6236
6396
6556
6716
6876
7036
7196
7356
7516
7676
7836
7996
8156
8316
8476
8636
8796
8956
9116
9276
9436
9596
9756
9916
10081
10244
10409
10576
10744
10912
11081
11250
11420
11590
11761
11932
12103
12274
12445
12616
12787
12958
13129
13299
13470
13641
13812
14103
14274
14445
14616
14787
14958
15129
15299
15470
15641
15812
16083
16254
16425
16596
16767
16938
17109
17280
17451
17622
17793
17964
18135
18306
18477
18648
18819
19090
19261
19432
19603
19774
19945
20116
20287
20458
20629
20799
20970
21141
21312
21483
21654
21825
22000
22171
22342
22513
22684
22855
23026
23197
23368
23539
23710
23881
24052
24223
24394
24565
24736
24907
25078
25249
25420
25591
25762
25933
26104
26275
26446
26617
26788
26959
27130
27301
27472
27643
27814
27985
28156
28327
28498
28669
28840
29011
29182
29353
29524
29695
29866
29937
30108
30279
30450
30621
30792
30963
31134
31305
31476
31647
31818
31989
32160
32331
32502
32673
32844
33015
33186
33357
33528
33699
33870
34041
34212
34383
34554
34725
34896
35067
35238
35409
35580
35751
35922
36093
36264
36435
36606
36777
36948
37119
37290
37461
37632
37803
37974
38145
38316
38487
38658
38829
39000
39171
39342
39513
39684
39855
39926
40097
40268
40439
40610
40781
40952
41123
41294
41465
41636
41807
41978
42149
42320
42491
42662
42833
42904
43075
43246
43417
43588
43759
43930
44101
44272
44443
44614
44785
44956
45127
45298
45469
45640
45811
45982
46153
46324
46495
46666
46837
46908
47079
47250
47421
47592
47763
47934
48105
48276
48447
48618
48789
48960
49131
49302
49473
49644
49815
49986
50157
50328
50499
50670
50841
51012
51183
51354
51525
51696
51867
52038
52209
52380
52551
52722
52893
53064
53235
53406
53577
53748
53919
54090
54261
54432
54603
54774
54945
55116
55287
55458
55629
55799
55970
56141
56312
56483
56654
56825
56996
57167
57338
57509
57680
57851
58022
58193
58364
58535
58706
58877
59048
59219
59390
59561
59732
59903
60074
60245
60416
60587
60758
60929
61099
61270
61441
61612
61783
61954
62125
62296
62467
62638
62809
62980
63151
63322
63493
63664
63835
64006
64177
64348
64519
64690
64861
65032
65203
65374
65545
65716
65887
66058
66229
66399
66570
66741
66912
67083
67254
67425
67596
67767
67938
68109
68280
68451
68622
68793
68964
69135
69306
69477
69648
69819
69990
70161
70332
70503
70674
70845
71016
71187
71358
71529
71699
71870
72041
72212
72383
72554
72725
72896
73067
73238
73409
73580
73751
73922
74093
74264
74435
74606
74777
74948
75119
75290
75461
75632
75803
75974
76145
76316
76487
76658
76829
76999
77170
77341
77512
77683
77854
78025
78196
78367
78538
78709
78880
79051
79222
79393
79564
79735
79906
80077
80248
80419
80589
80760
80931
81102
81273
81444
81615
81786
81957
82128
82299
82470
82641
82812
82983
83154
83325
83496
83667
83838
83999
84170
84341
84512
84683
84854
85025
85196
85367
85538
85709
85880
86051
86222
86393
86564
86735
86906
87077
87248
87419
87589
87760
87931
88102
88273
88444
88615
88786
88957
89128
89299
89470
89641
89812
89983
90154
90325
90496
90667
90838
91009
91180
91351
91522
91693
91864
92035
92206
92377
92548
92719
92890
93061
93232
93403
93574
93745
93916
94087
94258
94429
94599
94770
94941
95112
95283
95454
95625
95796
95967
96138
96309
96480
96651
96822
96993
97164
97335
97506
97677
97848
98019
98190
98361
98532
98703
98874
99045
99216
99387
99558
99729
99900
100071
100242
100413
100584
100755
100926
101097
101268
101439
101610
101781
101952
102123
102294
102465
102636
102807
102978
103149
103320
103491
103662
103833
104004
104175
104346
104517
104688
104859
105030
105201
105372
105543
105714
105885
106056
106227
106398
106569
106740
106911
107082
107253
107424
107595
107766
107937
108108
108279
108450
108621
108792
108963
109134
109305
109476
109647
109818
109989
110160
110331
110502
110673
110844
111015
111186
111357
111528
111699
111870
112041
112212
112383
112554
112725
112896
113067
113238
113409
113580
113751
113922
114093
114264
114435
114606
114777
114948
115119
115290
115461
115632
115803
115974
116145
116316
116487
116658
116829
116999
117170
117341
117512
117683
117854
118025
118196
118367
118538
118709
118880
119051
119222
119393
119564
119735
119906
120077
120248
120419
120589
120760
120931
121102
121273
121444
121615
121786
121957
122128
122299
122470
122641
122812
122983
123154
123325
123496
123667
123838
123999
124170
124341
124512
124683
124854
125025
125196
125367
125538
125709
125880
126051
126222
126393
126564
126735
126906
127077
127248
127419
127590
127761
127932
128103
128274
128445
128616
128787
128958
129129
129300
129471
129642
129813
129984
130155
130326
130497
130668
130839
131010
131181
131352
131523
131694
131865
132036
132207
132378
132549
132720
132891
133062
133233
133404
133575
133746
133917
134088
134259
134430
134601
134772
134943
135114
135285
135456
135627
135808
135979
136150
136321
136492
136663
136834
137005
137176
137347
137518
137689
137860
138031
138202
138373
138544
138715
138886
139057
139228
139399
139570
139741
139912
140083
140254
140425
140596
140767
140938
141109
141280
141451
141622
141793
141964
142135
142306
142477
142648
142819
142990
143161
143332
143503
143674
143845
144016
144187
144358
144529
144699
144870
145041
145212
145383
145554
145725
145896
146067
146238
146409
146580
146751
146922
147093
147264
147435
147606
147777
147948
148119
148290
148461
148632
148803
148974
149145
149316
149487
149658
149829
149999
150170
150341
150512
150683
150854
151025
151196
151367
151538
151709
151880
152051
152222
152393
152564
152735
152906
153077
153248
153419
153590
153761
153932
154103
154274
154445
154616
154787
154958
155129
155300
155471
155642
155813
155984
156155
156326
156497
156668
156839
156999
157170
157341
157512
157683
157854
158025
158196
158367
158538
158709
158880
159051
159222
159393
159564
159735
159906
160077
160248
160419
160590
160761
160932
161103
161274
161445
161616
161787
161958
162129
162300
162471
162642
162813
162984
163155
163326
163507
163678
163849
164020
164191
164362
164533
164704
164875
165046
165217
165388
165559
165730
165901
166
```

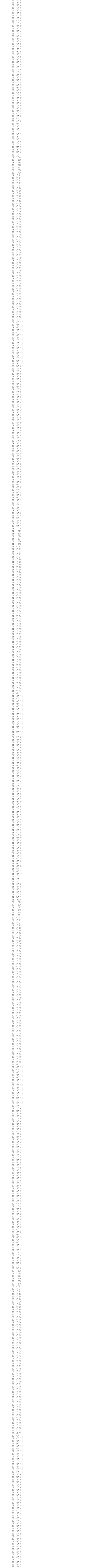


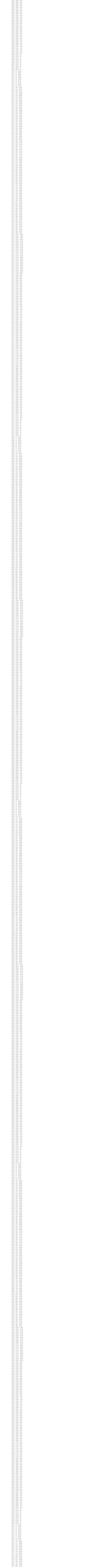


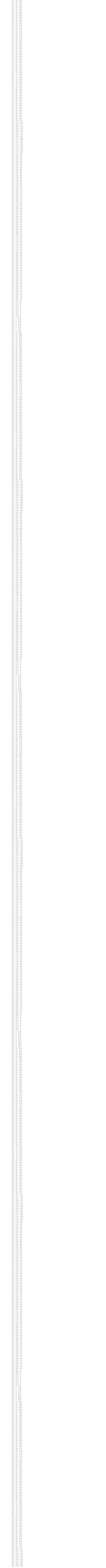




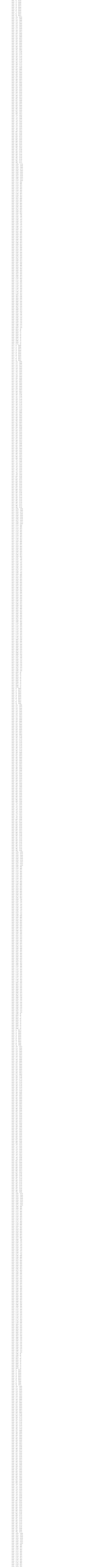






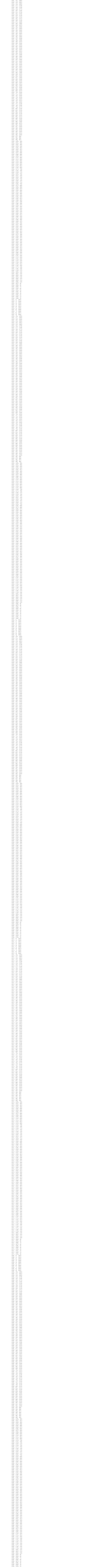


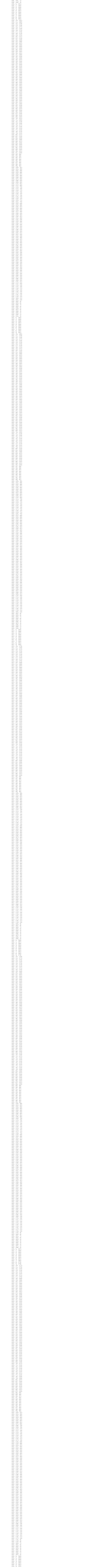


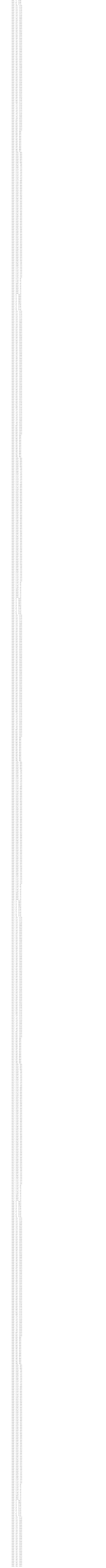


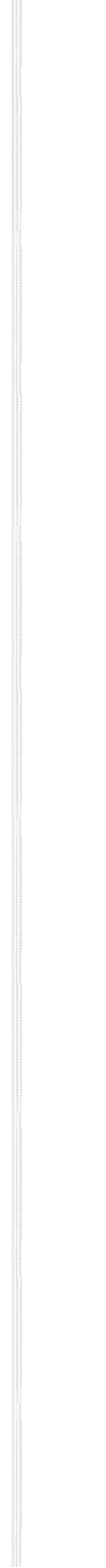


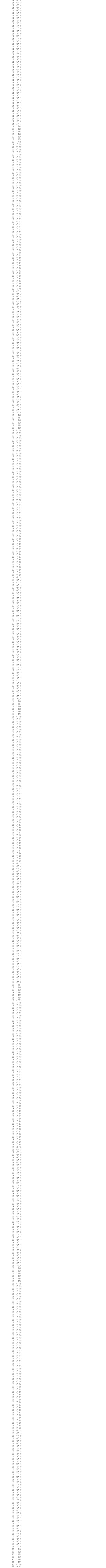




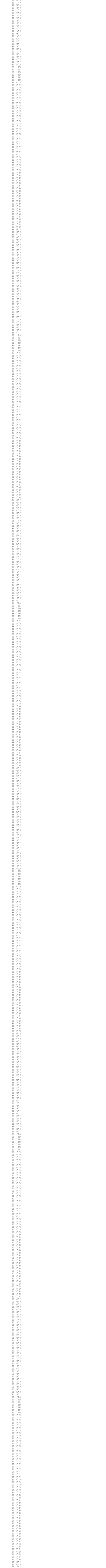


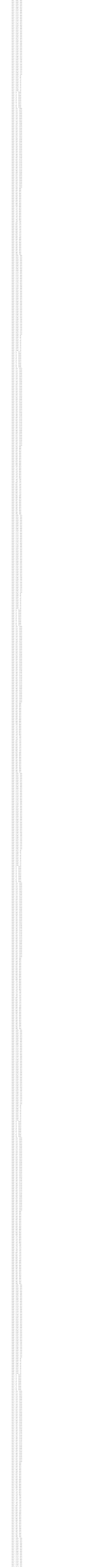


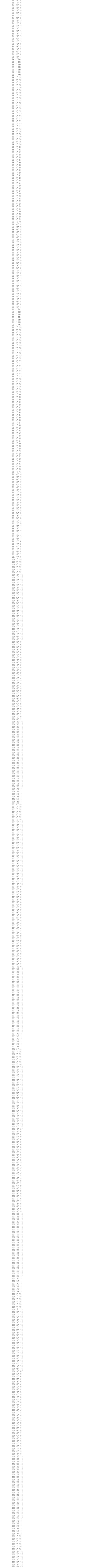


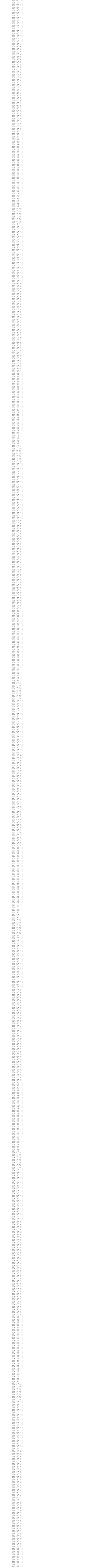


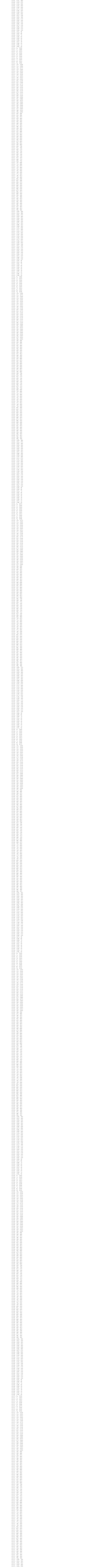


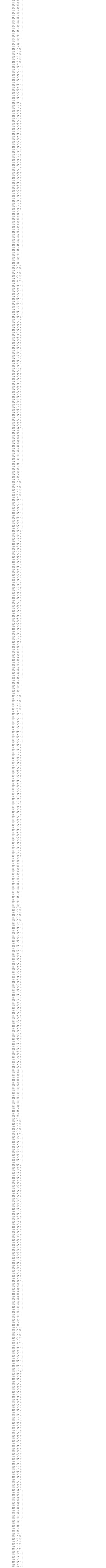


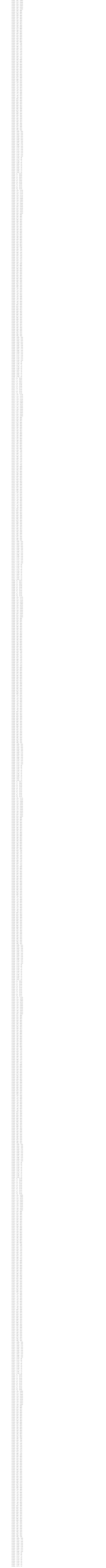


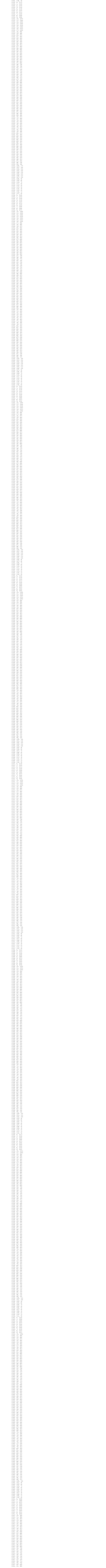


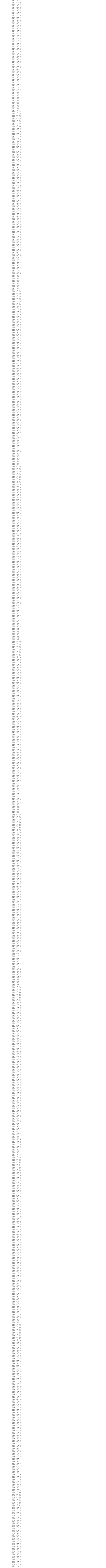


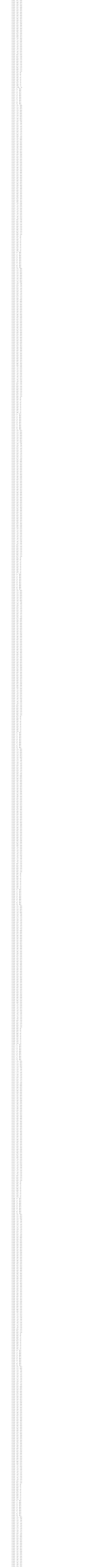


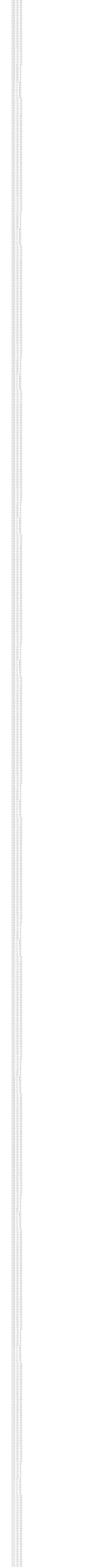


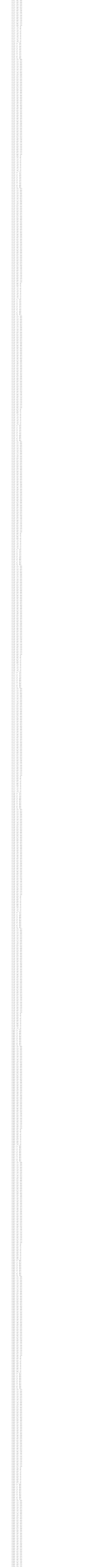














```
In [28]:  
def volume(c): return c[0] * c[1] * c[2]  
max([c for c in cuboids()  
     if c[0] + c[1] + c[2] == top], # another generator comprehension  
key=volume)  
  
Out[28]: (83, 83, 84)  
  
In [29]:  
def volume(c): return c[0] * c[1] * c[2]  
max([c for c in cuboids()  
     if c[0] + c[1] + c[2] == top], # another generator comprehension  
key=volume)  
  
Out[29]: (83, 83, 84)
```

Functional Programming

"Functional Programming" means programming with an emphasis on functions as the basic building blocks. It is particularly characterised by the use of higher-order functions and functions without side-effects, which we'll introduce below. In fact, we've already started doing functional programming. We'll start with the easiest concept, programming without side effects.

Functions without side effects

A *side effect* is something that a function does *other* than calculate and return a value. For example, printing to the terminal and writing to a file are side effects. Also, changing the values of an argument is a side effect. Example:

```
In [2]: def concat(a, b):
    a += b
    return a
a = [1, 2, 3]
b = [4, 5, 6]
c = concat(a, b)
print(a)
print(b)
print(c)
```

```
[1, 2, 3, 4, 5, 6]
[4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

As we see above, although the *return value* `c` is correct, this function has a side effect: it changes the first argument `a`. It's often better to avoid such effects. Programming without side-effects tends to allow for "pure", clean program design, and makes testing much easier.

```
In [3]: def concat(a, b):
    return a + b
a = [1, 2, 3]
b = [4, 5, 6]
c = concat(a, b)
print(a)
print(b)
print(c)
```

```
[1, 2, 3]
[4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

any and all

It is common to want to check whether all items in a list satisfy some *predicate*, e.g. to check whether all items are odd:

```
In [12]: flag = True
L = [17, 19, 23, 31, 32]
for x in L:
    if x % 2 == 0: # x even?
        flag = False
        break
print(flag)
```

```
False
```

In functional programming, we would try to improve this. First, we extract the predicate:

```
In [10]: # notice, no need for an if-else here!
def even(n): return n % 2 == 0
```

Next, Python provides a nice function `all` which reduces the whole thing to one line. The argument to `all` can be list, or another iterable type such as a generator comprehension. Note, we are not using a *higher-order* function here, as we are passing in Boolean values, not a function.

```
In [11]: all(not even(x) for x in L)
```

```
Out[11]: False
```

map

"Of course someone has to write loops. It doesn't have to be you." -- [Jenny Bryan](#)

 Adapted from Wickham, *The Joy of Functional Programming*

The `map` function is a central example of functional programming. It takes two arguments, a function `f` and a list `L`, and returns a new list created by applying `f` to each element of `L`. For example:

```
In [13]: list(map(len, ["a", "cat", "and", "a", "dog"]))
```

```
Out[13]: [1, 3, 3, 1, 3]
```

Note that we are passing in `len`, not `len()`. We are passing in a function, not the return value of a function.

`map` doesn't return a list: it returns an iterator, a bit like the generators we saw previously:

```
In [14]: map(len, ["a", "cat", "and", "a", "dog"])
```

```
Out[14]: <map at 0x111bebc50>
```

That's why, if we just want to see the results immediately (as opposed to iterate over them), we just enclose the `map` in a `list`:

```
In [15]: list(map(len, ["a", "cat", "and", "a", "dog"]))
```

```
Out[15]: [1, 3, 3, 1, 3]
```

`map` is an example of functional programming because our attention is on the transformation represented by `len` and the higher-order structure represented by `map`. We are not distracted with details of `for`-loops and initialisations.

Exercise: make your own implementation of `map` (but call it `mymap` to avoid overriding the builtin `map`) and show that it works using the example above.

`lambda` is common in combination with `map`, e.g.:

```
In [16]: list(map(lambda x: x**2, [4, 5, 6]))
```

```
Out[16]: [16, 25, 36]
```

Exercise: calculate e^x for every value of $x \in [0.0, 0.1, 0.2, \dots, 1.0]$. Use `range` to make a range of integers, then `lambda` and `map` to transform it.

 Adapted from Wickham, *The Joy of Functional Programming*

So far, with `map` we've only seen functions `f` which take just *one* argument. What if we want to work with a function `f` which takes more than one? For example, what if we had lists of numbers `x` and `y` and we wanted to calculate `x * y`. Well, it turns out that this works just fine with `map`: it accepts any number of lists:

```
In [17]: list(map(lambda x, y: x * y, [1, 2, 3, 4], [5, 1, 5, 1]))
```

```
Out[17]: [5, 2, 15, 4]
```

The above won't work with our `mymap`. We'd have to go and learn some extra concepts for it so we won't do that here.

Higher-order functions

Notice that when using `map`, we are passing-in a *function*. Specifically, we are passing-in a function such as `lambda x: x**2`, or the name of a function, such as `sq`. We don't pass in a function *call*, i.e. we don't write `map(sq(10), ...)`.

Any function which treats other functions as "just another data type", e.g. by taking functions as arguments or by returning functions, is called a *higher-order function* (HOF). `map` is the best-known HOF.

`max`, `min`, `L.sort`, `sorted`, and some other functions are also higher-order because as we know we can pass a *key function* to them, e.g.:

```
In [1]: sorted([-10, -5, 0, 5], key=lambda x: x**2)
```

```
Out[1]: [0, -5, 5, -10]
```

Callbacks

A *callback* is a function you supply to some other piece of code, in the knowledge that it will be called sometime later, not under your control. We have already seen some callbacks, e.g. when we passed a key function to `sorted` above, we knew that it would be called many times with various arguments.

Callbacks are common in two particular situations:

- In GUI programming, e.g. if we create a button then we also create a function `on_click` which will be called by the GUI framework every time the user clicks it. Example: <https://blog.kivy.org/2014/03/kivys-bind-method/>

- In long-running algorithms, such as optimisation algorithms including neural network training algorithms, we can often customize the output that is printed during the run by passing in a callback. Example:

https://keras.io/models/sequential/#fit_generator

Don't Repeat Yourself

The "don't repeat yourself" (DRY) principle is "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" -- Hunt and Thomas, *The Pragmatic Programmer*.

For a simple example, what could go wrong here?

```
def customer_id(L, x):
    if x in L:
        return L.index(x)
    else:
        return -1 # default value
```

```
# 1000 lines later...
```

```
if customer_id(L, x) == -1:
    print("Customer does not exist")
```

It is not obvious, but `-1` is duplicated. The danger here is that someone editing `customer_id()` might decide on a different default value, e.g. `None`, but not update the other use. DRY tells us to eliminate the duplication -- represent the default value in only one place -- to avoid that danger.

DRY, re-use and functional programming

"if you find yourself copying and pasting a block of code, you have probably found an opportunity for functional abstraction" -- <http://wla.berkeley.edu/~cs61a/fa11/lectures/functions.html>

Functional programming is especially suitable for DRY. The ability to pass one function to another allows us to write very reusable functions so that we don't have to copy and paste.

Overall, in functional programming the aim is again to write very *lego-like* programs: small pieces which can fit together in many different ways. This allows for maximum *re-use* of code.

Further reading (advanced): recall again our code for Newton's method for square roots. Walk through <http://wla.berkeley.edu/~cs61a/fa11/lectures/functions.html#example-newton-s-method> to see how it can be generalised in a highly DRY way, using functional programming ideas, to find logarithms and other real functions.



itertools and Factorial Design of Experiments

Itertools

`itertools` is one of my favourite modules in the Python standard library. It provides a lot of functions for creating iterators, especially for things like permutations, combinations, and Cartesian products. See <https://docs.python.org/3/library/itertools.html> for the full documentation including great examples.

Example: Factorial Design of Experiments

Let's pretend we are working with neural networks. Let's pretend we have programmed a neural network training function `NN(alpha, beta)` where the two parameters are numbers that control how it behaves. Suppose it returns one number representing performance (higher is better).

```
In [1]: import random
def NN(alpha, beta):
    # warning, this is just a placeholder.
    return alpha * random.random() + (
        1 - beta) * random.randrange(2)
```

We want to decide what are the best values for these parameters, i.e. what values gives the highest performance. Suppose we have a few likely values for each, already. We'll use nested for loops as follows.

```
In [2]: alphas = [0.0, 0.1, 0.2]
betas = [0, 1]
for alpha in alphas:
    for beta in betas:
        print(alpha, beta, NN(alpha, beta))

0.0 0 0.0
0.0 1 0.0
0.1 0 1.0638748038774313
0.1 1 0.09007449459279822
0.2 0 1.1301372412978836
0.2 1 0.015563778529370187
```

This is called a *factorial design of experiments*. We had two *factors* and we tried every possible value and we can judge which is the best.

However, the program design is bad in one way. What if our colleagues suggest to add another hyperparameter `gamma` ?

```
In [3]: def NN(alpha, beta, gamma):
    # warning, this is just a placeholder.
    return (alpha * random.random() +
           (1 - beta) *
           random.randrange(2) / gamma)
```

```
In [ ]: alphas = [0.0, 0.1, 0.2]
betas = [0, 1]
gammas = [0.9, 0.99, 0.999, 0.9999]
for alpha in alphas:
    for beta in betas:
        for gamma in gammas:
            print(alpha, beta, gamma,
                  NN(alpha, beta, gamma))
```

The addition of the extra hyperparameter means we have to change the structure, adding a for-loop. If we have a lot of hyperparameters, our code will look awful and will be a pain to work with.

```
In [4]: import itertools

alphas = [0.0, 0.1, 0.2]
betas = [0, 1]
gammas = [0.9, 0.99, 0.999, 0.9999]

# "Cartesian product", ie a grid over
# alphas x betas x gammas
for alpha, beta, gamma in itertools.product(
    alphas, betas, gammas):
    print(alpha, beta, gamma,
          NN(alpha, beta, gamma))
```

```
0.0 0 0.9 0.0
0.0 0 0.99 1.0101010101010102
0.0 0 0.999 0.0
0.0 0 0.9999 1.000100010001
0.0 1 0.9 0.0
0.0 1 0.99 0.0
0.0 1 0.999 0.0
0.0 1 0.9999 0.0
0.1 0 0.9 1.1558396000819633
0.1 0 0.99 0.05966784325368675
0.1 0 0.999 1.0717814645906805
0.1 0 0.9999 0.08348786761663249
0.1 1 0.9 0.026967774875449504
0.1 1 0.99 0.03109578891139805
0.1 1 0.999 0.033408642342887165
0.1 1 0.9999 0.014099061754399622
0.2 0 0.9 0.013180929834195077
0.2 0 0.99 1.0932624835597278
0.2 0 0.999 0.010007774240233958
0.2 0 0.9999 0.154025250487095
0.2 1 0.9 0.009814100695902961
0.2 1 0.99 0.13053908579954676
0.2 1 0.999 0.08986472883750807
0.2 1 0.9999 0.10043362111024019
```

`itertools.product` gives us an iterator over all the possible hyperparameter settings. Highly neat!

Further reading

- The `itertools` docs are good: <https://docs.python.org/3/library/itertools.html>
- There are some great recipes too: <https://docs.python.org/3/library/itertools.html#itertools-recipes>
- A study in algorithmic thinking and clear, simple Python from Peter Norvig, examples of comprehensions and generators, and more `itertools` : <http://nbviewer.jupyter.org/url/norvig.com/ipython/Golomb-Puzzle.ipynb> (By the way, you don't need to understand the parts about coloured rectangles and animations.)

Errors, Exceptions, and Tracebacks

Exceptions

An exception is a way of signalling that something unusual ("exceptional") has happened while the code is running.

- How to raise an exception
- How to handle one.

That might mean an error, for example. If the code doesn't know how to recover from the error, it can raise (or "throw") an exception. That changes the control flow of the program. If the exception is not handled (or "caught") inside the currently-executing function, then the function exits -- immediately, and without returning any value. The function that called this one then has a chance to handle (or "catch") it. If not, it continues to the function that called *that* one, and so on. If it is never caught, it propagates "all the way up", the program exits ("crashes"), and you see a Traceback on your screen.

For unrecoverable errors, that is probably just fine. Other times, we will want to handle an exception before the program exits, either within the current function, or in an enclosing one.

So, there are two main mechanisms we need to know about: how to raise an exception, and how to handle one.

```
In [1]: def get_mobile_number(operator, user):
    """Given an operator and a user number,
    return the full mobile number. For example:
    >>> get_mobile_number("meteor", "1234567")
    '0851234567'

    """
    # We have a mapping from operators to prefixes.
    d = {"virgin": "087", "three": "086",
         "meteor": "085"}
    return d[operator] + user
```

This works as follows:

```
In [2]: get_mobile_number("virgin", "1234567")
```

```
Out[2]: '0871234567'
```

But if we run something like the following, we'll get a `KeyError`, because T-Mobile doesn't exist:

```
In [3]: get_mobile_number("T-Mobile", "1234567")
```

```
-----  
KeyError                                     Traceback (most recent call last)  
<ipython-input-3-5c616191561f> in <module>  
----> 1 get_mobile_number("T-Mobile", "1234567")  
  
<ipython-input-1-390f2cbf7649> in get_mobile_number(operator, user)  
    10     # We have a mapping from operators to prefixes.  
    11     d = {"virgin": "087", "three": "086", "meteor": "085"}  
----> 12     return d[operator] + user  
  
KeyError: 'T-Mobile'
```

That `KeyError` is an exception. It was raised by the code that implements dictionaries to indicate the requested key wasn't found. Our variable `d` is a dictionary. When we tried to look up the value of "T-Mobile", it found there was no such key in `d`. It (the dict) didn't print out an error message, or return a special error value, or anything like that. All the dict did was raise an exception, one called `KeyError`. Because our code didn't handle the exception, it propagated all the way to the console and caused a crash.

Catching exceptions

But what if we wanted to do something else with a bad operator, instead of crashing the program? Let's say we're printing out mobile numbers, and anytime we get an unrecognised operator, we should just print nothing, instead. We can do that by "catching" the exception, using a try-except block.

A try-except block looks a bit like an if-statement. That reflects the fact that it affects control flow, just like an if statement does. It means something like: run this, and if there's an exception, clean up by running something else instead.

We can either catch the error from outside the function:

```
In [4]: user = "1234567"
oper = "T-Mobile"
try:
    # try this code...
    print(get_mobile_number(oper, user))
except KeyError:
    # ... and if a KeyError happens
    # ... do this instead.
    print("Operator " + oper + " not found")
```

Operator T-Mobile not found

Or we can catch it inside the function. It depends what we want to achieve. The following new version won't crash. If there is a `KeyError` while we're "trying" the first piece of code, then we'll just return `None` instead.

```
In [5]: def get_mobile_number(operator, user):
    # We have a mapping from operators to prefixes. Note that we use a
    # string for the number and the prefix -- why?
    d = {"virgin": "087", "three": "086", "meteor": "085"}
    try:
        return d[operator] + user # try this code...
    except KeyError: # ... and if a KeyError happens...
        return None # ... do this instead.

print(get_mobile_number(oper, user))
```

None

Exercise: Write a function which accepts two numbers and returns the first divided by the second. Observe the exception that is thrown if you try to divide by zero. Now use a try-except block to catch that exception and print a sensible error message, instead of crashing the program.

We can also use multiple `except` clauses, e.g.:

```
try:
    something
except KeyError:
    deal_with_KeyError
except ZeroDivisionError:
    deal_with_ZeroDivisionError
```

Exercise: Observe the exceptions that are thrown in the below cases, and then repair the function using multiple `except` clauses to do something sensible that doesn't crash.

```
In [6]: def subtract_one(L):
    L[0] -= 1
    return L
```

```
In [7]: subtract_one([0, 1, 2, 3, 4]) # should work ok
```

```
Out[7]: [-1, 1, 2, 3, 4]
```

```
In [8]: subtract_one((0, 1, 2, 3, 4))
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-8-377662191a46> in <module>
----> 1 subtract_one((0, 1, 2, 3, 4))

<ipython-input-6-3439f8f0c97e> in subtract_one(L)
    1 def subtract_one(L):
----> 2     L[0] -= 1
    3     return L

TypeError: 'tuple' object does not support item assignment
```

```
In [9]: subtract_one([])
```

```
-----  
IndexError                                     Traceback (most recent call last)  
<ipython-input-9-9ea726af39b> in <module>
----> 1 subtract_one([])
      2
      3     return L

IndexError: list index out of range
```

Raising exceptions

So far, we've seen how to handle ("catch") exceptions. What about raising ("throwing") them? The most common scenario is when our function receives arguments which are "impossible" in some way, or when processing data that doesn't make sense.

```
In [31]: from math import sqrt
def triangle_area(x, y, z):
    """Calculate the area of a triangle (not necessarily right-angled)
    given its three sides. Heron's formula tells us the area. However,
    not all values of x, y, and z are valid. If the two smallest of
    them add up to less than the largest, no such triangle exists.
    """
    # Order so that x <= y <= z
    x, y, z = sorted((x, y, z))
    # Check that the values are valid
    if x + y < z:
        # ... and if not, raise an exception.
        raise ValueError("No such triangle exists")

    # Heron's formula [https://www.mathsisfun.com/geometry/herons-formula.html]
    s = (x + y + z)/2.0
    area = sqrt(s * (s-x) * (s-y) * (s-z))
    return area
```

```
In [32]: print(triangle_area(5, 5, 5)) # should work ok
```

```
10.825317547305483
```

```
In [33]: print(triangle_area(10, 2, 2)) # we detect an impossible triangle and raise an exception
```

```
-----  
ValueError                                     Traceback (most recent call last)  
<ipython-input-33-86d7a7cf954e> in <module>
----> 1 print(triangle_area(10, 2, 2)) # we detect an impossible triangle and raise an exception

<ipython-input-31-b45acaec40c> in triangle_area(x, y, z)
    11     if x + y < z:
    12         # ... and if not, raise an exception.
----> 13         raise ValueError("No such triangle exists")
    14
    15     # Heron's formula [https://www.mathsisfun.com/geometry/herons-formula.html]
```

```
ValueError: No such triangle exists
```

Exceptions and doctests

We can take account of exceptions in our doctests. That is, we can write a doctest where the expected result is a Traceback and exception. For example, we can write a doctest like the following. Note, we can put `...`` to indicate some extra text that the doctest can ignore.

```
In [34]: def triangle_area(x, y, z):
    """
    A normal, working call:
    >>> triangle_area(5, 5, 5)
    10.825317547305483
    """

    An impossible triangle: we hope to see a Traceback:
    >>> triangle_area(10, 2, 2)
    Traceback (most recent call last):
    ...
    ValueError: No such triangle exists
    """

    # Order so that x <= y <= z
    x, y, z = sorted((x, y, z))
    # Check that the values are valid
    if x + y < z:
        # ... and if not, raise an exception.
        raise ValueError("No such triangle exists")
```

```
# Heron's formula [https://www.mathsisfun.com/geometry/herons-formula.html]
```

```
s = (x + y + z)/2.0
```

```
area = sqrt(s * (s-x) * (s-y) * (s-z))
```

```
return area
```

```
In [35]: import doctest
doctest.testmod()
```

```
*****
```

```
File "__main__", line 5, in __main__.get_last_n_elements
```

```
Failed example:
```

```
get_last_n_elements("abcde", 8)
```

```
Expected:
```

```
Traceback (most recent call last):
```

```
...
ValueError: Requested too many elements (8 versus 5)
```

```
Got:
```

```
Traceback (most recent call last):
```

```
File "/Users/jmmcdonald/anaconda3/lib/python3.7/doctest.py", line 1329, in __run
```

```
compileflags, 1), test_globs)
```

```
File "<doctest __main__.get_last_n_elements[1]>", line 1, in <module>
```

```
    get_last_n_elements("abcde", 8)
```

```
File "<ipython-input-27-500f12525>", line 11, in get_last_n_elements
```

```
    raise ValueError(f"Requested too many elements ({n} versus {len(s)})")
```

```
ValueError: Requested too many elements (8 versus 5)
```

```
*****
```

```
1 item had failures:
  1 of  2 in __main__.get_last_n_elements
***Test Failed*** 1 failures.
```

```
Out[35]: TestResults(failed=1, attempted=4)
```

Exercise: What happens here if `s` is shorter than `n` elements? Try it.

```
In [15]: def get_last_n_elements(s, n):
    return s[-n:]
```

```
In [16]: get_last_n_elements("abcde", 2)
```

```
Out[16]: 'de'
```

Now, change the code to raise a `ValueError` in that case. What if `n` was negative? Again, raise a `ValueError` in that case. Test it works correctly. Finally, add some doctests. Add both positive ones (normal operation) and negative ones (`ValueError` tracebacks)

```
In [38]: def get_last_n_elements(s, n):
    """
    >>> get_last_n_elements("abcde", 2)
    'de'
    >>> get_last_n_elements("abcde", 8)
    Traceback (most recent call last):
    ...
    ValueError: Requested too many elements (8 versus 5)
    """

    if n > len(s):
        raise ValueError(f"Requested too many elements ({n} versus {len(s)})")
    elif n < 0:
        raise ValueError("Requested negative number of elements")
    return s[-n:]
```

```
In [39]: import doctest
doctest.testmod()
```

```
*****
```

```
File "__main__", line 5, in __main__.get_last_n_elements
```

```
Failed example:
```

```
get_last_n_elements("abcde", -1)
```

```
Expected:
```

```
Traceback (most recent call last):
```

```
...
ValueError: Requested negative number of elements
```

```
Got:
```

```
Traceback (most recent call last):
```

```
File "/Users/jmmcdonald/anaconda3/lib/python3.7/doctest.py", line 1329, in __run
```

```
compileflags, 1), test_globs)
```

```
File "<doctest __main__.get_last_n_elements[1]>", line 1, in <module>
```

```
    get_last_n_elements("abcde", -1)
```

```
File "<ipython-input-27-500f12525>", line 11, in get_last_n_elements
```

```
    raise ValueError(f"Requested negative number of elements")
```

```
ValueError: Requested negative number of elements
```

```
*****
```

```
1 item had failures:
  1 of  1 in __main__.get_last_n_elements
***Test Failed*** 1 failures.
```

```
Out[39]: TestResults(failed=0, attempted=4)
```

Exercise: What happens here if `s` is shorter than `n` elements? Try it.

```
In [14]: def get_last_n_elements(s, n):
    return s[-n:]
```

```
In [16]: get_last_n_elements("abcde", 2) # ok
```

```
Out[16]: 'de'
```

```
In [17]: get_last_n_elements("abcde", 8) # no crash, but misleading for the user I think
```

```
Out[17]: 'abcde'
```

Now, change the code to raise a `ValueError` in that case. What if `n` was negative? Again, raise a `ValueError` in that case. Test it works correctly. Finally, add some doctests. Add both positive ones (normal operation) and negative ones (`ValueError` tracebacks)

```
In [38]: def get_last_n_elements(s, n):
    """
    >>> get_last_n_elements("abcde", 2)
    'de'
    >>> get_last_n_elements("abcde", 8)
    Traceback (most recent call last):
    ...
    ValueError: Requested too many elements (8 versus 5)
    """

    if n > len(s):
        raise ValueError(f"Requested too many elements ({n} versus {len(s)})")
    elif n < 0:
        raise ValueError("Requested negative number of elements")
    return s[-n:]
```

```
In [39]: import doctest
doctest.testmod()
```

```
*****
```

```
File "__main__", line 5, in __main__.get_last_n_elements
```

```
Failed example:
```

```
get_last_n_elements("abcde", -1)
```

```
Expected:
```

```
Traceback (most recent call last):
```

```
...
ValueError: Requested negative number of elements
```

```
Got:
```

```
Traceback (most recent call last):
```

```
File "/Users/jmmcdonald/anaconda3/lib/python3.7/doctest.py", line 1329, in __run
```

```
compileflags, 1), test_globs)
```

```
File "<doctest __main__.get_last_n_elements[1]>", line 1, in <module>
```

```
    get_last_n_elements("abcde", -1)
```

```
File "<ipython-input-27-500f12525>", line 11, in get_last_n_elements
```




Strings and Formatted Output

Reminder

```
In [1]: s = "abc" # equivalent  
       s = 'abc' # but not s = 'abc"
```

```
In [2]: s.startswith("ab")
```

```
Out[2]: True
```

Formatting strings

There are several ways of formatting strings in Python. The resulting strings can be printed to the screen or written to a file, or used for some other purpose -- the formatting works the same regardless.

Formatting strings using `%`

The string interpolation operator `%` is probably still the most common in the wild.

The `%` operator is used *twice* in formatting.

```
In [3]: for i in range(5):  
        if i % 2 == 0: # arithmetic % (mod)  
            # %d is a format specifier  
            # bare % is string interpolation  
            s = "i = %d" % i  
            print(s)  
  
i = 0  
i = 2  
i = 4
```

We can use different `%` format strings to format values of different types:

```
In [4]: import math  
"%d, %f, %.2f: %s" % (17, math.pi,  
                         math.pi, "hello")
```

```
Out[4]: '17, 3.141593, 3.14: hello'
```

Formatting strings using `.format()`

```
In [7]: for i in range(10):  
        if i % 2 == 0:  
            print("i = {}".format(i))  
  
i = 0  
i = 2  
i = 4  
i = 6  
i = 8
```

Formatting strings using `f`-strings

I think modern Python authors prefer `f`-strings.

```
In [6]: for i in range(10):  
        if i % 2 == 0:  
            s = f"i = {i}" # special syntax f""  
            print(s)  
  
i = 0  
i = 2  
i = 4  
i = 6  
i = 8
```

Further reading

Numpy

Why Numpy?

- Speed
- Abstraction
- A library of pre-written functions

```
In [3]: import numpy as np
```

A cheat sheet: <https://www.dataquest.io/blog/numpy-cheat-sheet/>

Numpy ("Numerical Python", pronounced NUM-pie, not NUM-pee) is a library used in Python for numerical computing. Most scientific computing work in Python relies on Numpy as a base.

But we can already do numerical calculations in Python, so why does Numpy exist?

1. **Speed.** Numpy makes many numerical calculations much faster.
2. **Abstraction.** It is very handy to be able to think of our equations as (e.g.) $y = \beta x$ as opposed to $y_0 = \beta x_0, y_1 = \beta x_1$, etc., even though they mean the same thing.
3. **A library.** Numpy provides many common functions. "Batteries included."

In this notebook, we'll start by seeing a nice 3rd-party tutorial which emphasises abstraction, and then fill in some extra details. In later notebooks, we'll see examples, input/output, plotting, and more.

A nice introduction to Numpy basics which focusses on the benefit of abstraction: <https://jalammar.github.io/visual-numpy/>. We will look at an example calculation of mean-square-error.

Numpy array

"NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers" - <https://numpy.org/devdocs/user/quickstart.html>

```
In [29]: a = np.array([4, 5, 6.0]) # make an array, passing in a list
print(a)
```

```
[4. 5. 6.]
```

```
In [30]: a.shape
```

```
Out[30]: (3,)
```

```
In [31]: a.dtype # data type
```

```
Out[31]: dtype('float64')
```

Vectorisation

A typical use-case for Numpy: we have a for-loop processing a list of numerical values, and we replace it with a single Numpy line. This is called *vectorisation*. The same concept is essential to good performance in Matlab and R.

```
In [27]: L = [4, 5, 6]
for i in range(len(L)):
    L[i] = L[i]**2 # L a Python list

a = a**2 # a a Numpy array
```

From the point of view of *abstraction*, the for-loop is hidden. From the point of view of *speed*, the for-loop is moved from pure Python into an underlying function written in C or Fortran.

- Abstraction/brevity
- Homogeneous operations/no flexibility -> speed

When dealing with large data, Python can be slow. If we have a list of 10 numbers and we calculate the mean, it is instantaneous. But if we have 10 million numbers, it will be slow. The reason for this is Python's *flexibility*. Python allows a list to contain any type of value, eg we can have a mixed list of `int`s, `float`s, `string`s, other `list`s, and so on. Python has to check what type each value is before deciding how to add it (or whether it even *can* add it).

In a Numpy array, all elements are of the same type, e.g. all `float`. Thus there is no need for Python to waste time checking what type each value is. The saving is probably a factor of 100, depending on the workload.

Vectorisation is also happening when, e.g., we add two arrays:

```
In [5]: b = np.array([1, 2, 3])
print(a + b)
```

```
[17 27 39]
```

We should become comfortable with the difference between *element-wise arithmetic* and *aggregation*. These look similar, but the result is an `array` in one case and a single value in the other.

```
In [4]: x = np.array([4, 9, 16])
print(np.sqrt(x))
print(np.max(x))
```

```
[2. 3. 4.]
```

```
16
```

More ways of making arrays

- `np.zeros(shape)` and `np.ones(shape)`
- Random numbers: we can use `np.random.random(shape)` for uniform values in $[0, 1]$. We can also generate from other distributions, e.g. using `np.random.normal(shape)`.

`np.sum(a)` versus `a.sum()`

In several cases, one can write either style.

```
In [6]: a = np.array([4, 9, 16])
print(np.sum(a))
print(a.sum())
```

```
29
```

```
29
```

But for arithmetic functions, most are not available as methods of the array, e.g.:

```
In [7]: print(np.sqrt(a))
try:
    print(a.sqrt())
except:
    print("That doesn't exist!")
```

```
[2. 3. 4.]
```

```
That doesn't exist!
```

`argmax` and friends are often overlooked (and eventually re-implemented) by beginners.

```
In [8]: x = np.array([4, 5, 6, 1])
# the index where the largest element is
print(x.argmax())
```

```
2
```

We have seen how to create Numpy arrays by passing in lists. Of course, another way is to read from a file. We'll cover file input/output in a later notebook/video.

`np.linspace()`

Another handy way to create an array is to create evenly-spaced values. We use `np.linspace`. We have to say where the values start and stop, and how many there should be. `np.linspace` works out the rest:

```
In [28]: # start, stop, n_values
grid = np.linspace(0, 10, 11)
print(grid)
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

Exercises

Exercise: Using `linspace`, make this array:

```
[ 0.  0.25 0.5 0.75 1.  1.25 1.5 1.75 2.]
```

Exercise: `np.logspace` does something similar. Use it to make this array:

```
[1.e-06 1.e-05 1.e-04 1.e-03 1.e-02 1.e-01 1.e+00]
```

Exercise: Generate a list of 20 numbers in a Gaussian (normal) distribution, with mean 10 and standard deviation 10. Confirm the statistics are correct using `np.mean` and `np.std`.

Exercise: What happens if we try to add two arrays of different lengths?

Further reading

- Here is a nice textbook reference on Numpy with several longer worked examples: <http://www.labri.fr/perso/nrougier/from-python-to-numpy/>
- If you're already good at Matlab: <https://www.numpy.org/devdocs/user/numpy-for-matlab-users.html>
- If you're already good at R: <http://mathesaurus.sourceforge.net/r-numpy.html>

Solutions to exercises

```
In [11]: import numpy as np
np.linspace(0, 2, 8)
```

```
Out[11]: array([0.          , 0.28571429, 0.57142857, 0.85714286, 1.14285714,
```

```
           1.42857143, 1.71428571, 2.          ])
```

```
In [15]: np.logspace(-6, 0, 7)
```

```
Out[15]: array([1.e-06, 1.e-05, 1.e-04, 1.e-03, 1.e-02, 1.e-01, 1.e+00])
```

```
In [21]: x = np.random.normal(10, 10, 20)
print(x)
print(x.mean(), x.std())
```

```
[ 8.8834325 19.4112422 33.90154357 11.6035311  4.256217  11.32856614
```

```
13.18182866 -7.69053737 2.7394066 12.62946496 -0.94995881 18.00138806
```

```
-4.38183108 2.64972255 5.87981063 25.06102658 2.77559316 10.2004984
```

```
-7.17798373 -3.68822081]
```

```
7.930737116979188 10.53276469422564
```

```
In [14]: np.array([4, 5, 6]) + np.array([1, 2, 3, 4])
```

```
ValueError: Traceback (most recent call last)
<ipython-input-14-11b3836befd7> in <module>
----> 1 np.array([4, 5, 6]) + np.array([1, 2, 3, 4])
```

```
ValueError: operands could not be broadcast together with shapes (3,) (4,)
```


Numpy example: Heart Rate

<https://my.clevelandclinic.org/health/diagnostics/16953-electrocardiogram-ekg>

Example: suppose we are in a hospital setting. We are programming an *electrocardiogram* device, that is a device which measures the electrical activity near the heart at a rate of 360Hz.

<https://www.onlinebiologynotes.com/electrocardiogram-ecg-working-principle-normal-ecg-wave-application-of-ecg/>

The result is the familiar heart-beat signal. How can we calculate the *heart rate* in beats per minute (bpm)?

We'll answer this using basic signal-processing methods in Numpy.

Reading and extracting data

```
In [2]: import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline
```

We'll use a sample data file downloaded from <https://data.mendeley.com/datasets/7dybx7wyfn/3>. (I have copied it to the `data/` directory in `W03.zip`.)

It is in Matlab format, which we can load using the [input-output module of the Scipy library](#).

```
In [3]: import scipy.io as sio  
m = sio.loadmat("data/ECG_MLII_1_NSR_100m_0.mat")
```

```
In [3]: # let's inspect:  
type(m)
```

```
Out[3]: dict
```

```
In [4]: list(m.keys())
```

```
Out[4]: ['val']
```

```
In [5]: type(m['val'])
```

```
Out[5]: numpy.ndarray
```

```
In [4]: # it seems m['val'] is what we want.  
x = m['val']
```

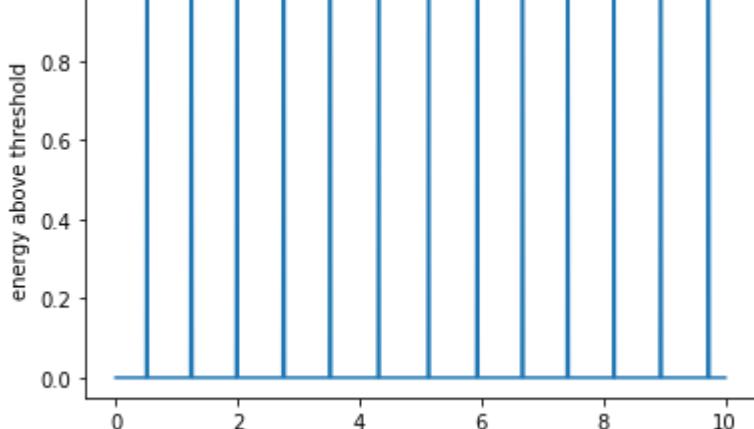
```
In [7]: # let's inspect further  
x.shape
```

```
Out[7]: (1, 3600)
```

```
In [5]: # we would prefer a simple 1D array  
x = x.flatten()
```

Now we are ready to take a look at the signal. It looks familiar, so we feel confident that everything is ok so far.

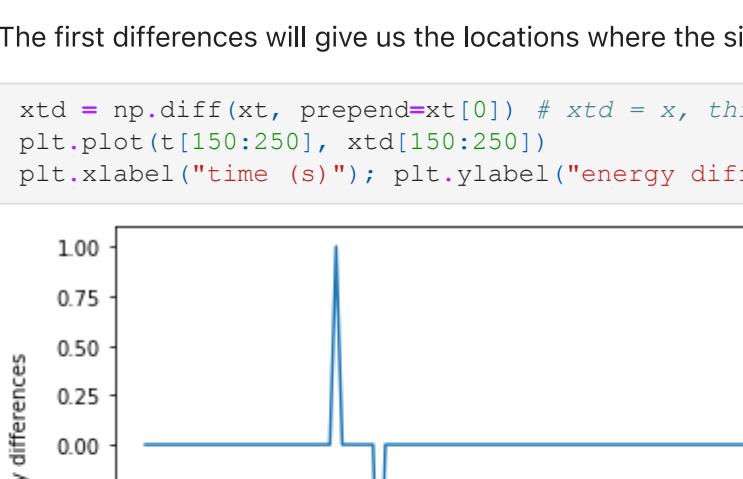
```
In [6]: fs = 360 # sampling frequency in Hz, see https://data.mendeley.com/datasets/7dybx7wyfn/3  
L = 10 # length of signal in seconds  
t = np.linspace(0, L, fs*L)  
plt.plot(t, x); plt.xlabel("time (s)"); plt.ylabel("energy");
```



Signal processing to find heart rate: overall plan

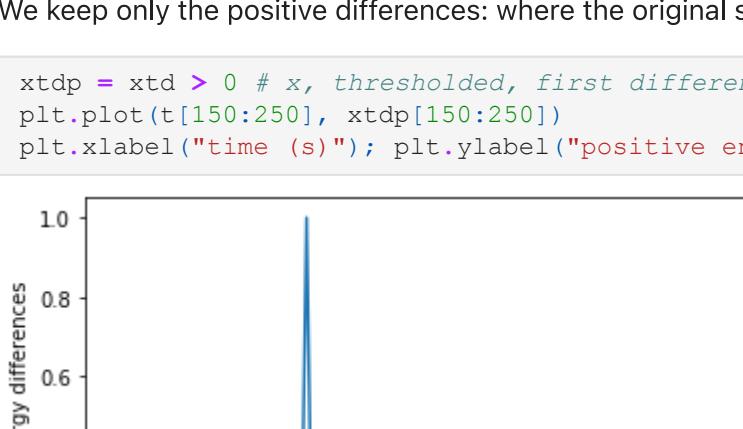
- Find the *peaks*, one per heart-beat, as follows:
 - Apply a threshold to make signal binary, setting $x_t = 1$ for any very high value and 0 otherwise
 - Take first differences, ie change from point to point
 - Find points where the first difference is positive, ie change from non-peak to peak
- Calculate how many peaks per unit time.

```
In [7]: # apply a threshold and plot  
xt = x > 1050 # xt = x thresholded  
plt.plot(t, xt); plt.xlabel("time (s)"); plt.ylabel("energy above threshold");
```



Next we zoom in. Notice that each peak is *not* a single sample, but several successive ones.

```
In [8]: plt.plot(t[150:250], xt[150:250])  
plt.xlabel("time (s)"); plt.ylabel("energy above threshold");
```



The first differences will give us the locations where the signal *changes* from 0 to 1 or 1 to 0.

```
In [9]: xtd = np.diff(xt, prepend=xt[0]) # xtd = x, thresholded, first differences  
plt.plot(t[150:250], xtd[150:250])  
plt.xlabel("time (s)"); plt.ylabel("energy differences");
```



We keep only the positive differences: where the original signal goes from "not very high" to "very high"

```
In [10]: xtdp = xtd > 0 # x, thresholded, first differences, positive  
plt.plot(t[150:250], xtdp[150:250])  
plt.xlabel("time (s)"); plt.ylabel("positive energy differences");
```


Now each heart beat is represented by a single value 1 surrounded by zeros. The heart rate is just the number of these, divided by total time.

```
In [14]: np.sum(xtdp) / L * 60 # heart rate in bpm
```

```
Out[14]: 78.0
```


Floating-point numbers

As we know, computer languages can't represent *real* numbers perfectly, because a real number can take on an infinite number of possible values (even in $[0, 1]$), but for any particular number only finite storage is available in the computer.

Let's say we are using binary strings of 64 bits to represent real numbers. Consider just a tiny subset of the number line as shown. Let's say it starts at x represented by some bitstring of length 64. There has to be some number which is the next largest, out of all possible bitstrings of length 64. Call that $\text{successor}(x)$. Well, in between there are still infinitely many (in fact, uncountably many) other numbers which can't be represented in our 64-bit system! Any of them can only be rounded off to either x or $\text{successor}(x)$.

Floating-point, specified by IEEE 754, is the most common scheme for representing real numbers. Single-precision numbers (e.g. `float` in the C language) use 32 bits each, and double-precision (`double`) use 64. In Python all floating-point numbers are double-precision but are just called `float`.

FP is inexact

Floating-point calculations can be inexact:

```
In [1]: 2 / 3
Out[1]: 0.6666666666666666
```

By the way, if you want C-style integer division, it is available using `//`:

```
In [6]: 2 // 3
Out[6]: 0
```

A consequence of this inexactness is that `x == y` is not meaningful for `float` values.

```
In [3]: import math
x = math.sqrt(2) ** 2
y = 2.0
x == y
```

```
Out[3]: False
```

Instead we usually use a construction like this:

```
In [4]: eps = 10**-8 # epsilon, a small constant which we take as "close enough"
abs(x - y) < eps
Out[4]: True
```

FP has upper and lower bounds on magnitude

Some calculations just become too large, and we can get an `OverflowError`:

```
In [5]: print("ok:", (10.0**10.0)**10.0)
print("not ok:", 10.0**10.0**10.0)
ok: 1e+100
-----
OverflowError                                     Traceback (most recent call last)
<ipython-input-5-dc137f252e61> in <module>
      1 print("ok:", (10.0**10.0)**10.0)
----> 2 print("not ok:", 10.0**10.0**10.0)

OverflowError: (34, 'Result too large')
```

Machine epsilon is the academic term for the *smallest magnitude that can be represented in a floating-point system*.

Exercise: what is machine epsilon in Python?

```
In [ ]: # HINT
print(10 ** -10)
print(10 ** -350)
```

Special numbers

IEEE 754 allows for three special numbers: infinity (`inf`), negative infinity (`-inf`), and *not a number* (`nan`). In principle these can arise through division by zero and similar operations, but in practice Python just throws an error instead:

```
In [7]: z = 15 / 0 # would be "inf"
-----
ZeroDivisionError                               Traceback (most recent call last)
<ipython-input-7-af5a1b8b55a3> in <module>
----> 1 z = 15 / 0 # would be "inf"

ZeroDivisionError: division by zero
```

```
In [8]: 0 / 0 # would be "nan"
-----
ZeroDivisionError                               Traceback (most recent call last)
<ipython-input-8-914a11a0ba07> in <module>
----> 1 0 / 0 # would be "nan"

ZeroDivisionError: division by zero
```

However, we can create these by hand and look at their properties:

```
In [9]: inf = float('inf')
neg_inf = -inf
nan = float('nan')
```

```
In [10]: inf > 3
```

```
Out[10]: True
```

```
In [11]: -inf < inf
```

```
Out[11]: True
```

```
In [12]: inf + -inf # don't expect zero, here!
```

```
Out[12]: nan
```

```
In [13]: inf + 3
```

```
Out[13]: inf
```

```
In [18]: nan < 3
```

```
Out[18]: False
```

Floating-point errors in Numpy

All of the above is using pure Python. The story becomes slightly different in Numpy. Numpy uses the same floating-point system as pure Python. However, its method of handling errors is different, because in Numpy it is common to process an entire array at once, and an error (e.g. division by zero) might occur only for some elements of the array. So, Numpy allows you to *choose* how errors are handled.

```
In [15]: import numpy as np
np.seterr(divide="warn") # see https://docs.scipy.org/doc/numpy/reference/generated/numpy.seterr.html
```

```
Out[15]: {'divide': 'warn', 'over': 'warn', 'under': 'ignore', 'invalid': 'warn'}
```

Here, `divide="warn"` says: if a divide-by-zero error happens, please warn me, but allow the result to be calculated, so I can see it.

```
In [16]: x = -1.0
y = np.array([0.0, 1.0, 2.0]) # -1 / 0 -> -inf
x / y
```

```
/Users/jmmcd/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3: RuntimeWarning: divide by zero encountered in true_divide
```

```
This is separate from the ipykernel package so we can avoid doing imports until
```

```
Out[16]: array([-inf, -1., -0.5])
```

```
In [17]: z = np.array([0.0, 1.0, 2.0]) # 0 / 0 -> nan
y / z
```

```
/Users/jmmcd/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in true_divide
```


Numpy: Multidimensional Arrays and Fancy Indexing

Multidimensional arrays

A 2-dimensional array is like a matrix in maths. We can again make a 2-dimensional array in several ways, e.g. using functions like `np.ones`:

```
In [3]: import numpy as np
M = np.ones((3, 3))
print(M)
```

```
[1. 1. 1.]
[1. 1. 1.]
[1. 1. 1.]
```

Here, we must pass in the `shape` (not just the length). The shape is a tuple of integers.

By the way, we can write `M` on its own in IPython or Jupyter Notebook, and we'll see the output:

```
In [7]: M
```

```
Out[7]: array([[1., 1., 1.],
               [1., 1., 1.],
               [1., 1., 1.]])
```

But writing `print(M)` gives a slightly nicer output:

```
In [8]: print(M)
```

```
[1. 1. 1.]
[1. 1. 1.]
[1. 1. 1.]
```

We can also pass in a list of lists to make a two-dimensional array composed of the elements of those lists:

```
In [9]: X = [[1.0, 2.0],
          [3.0, 4.0]] # just a list of lists
X = np.array(X)
print(X)
```

```
[1. 2.]
[3. 4.]
```

Again, we can do operations on each element:

```
In [10]: X + 10
```

```
Out[10]: array([[11., 12.],
               [13., 14.]])
```

Of course, `X` has not been altered by this:

```
In [11]: X
```

```
Out[11]: array([[1., 2.],
               [3., 4.]])
```

We can find out the shape of an array:

```
In [12]: X.shape
```

```
Out[12]: (2, 2)
```

Fancy Indexing

When we studied lists, they were always one-dimensional:

```
L = [4, 5, 6]
```

It requires just one integer to index `L`.

(Ok, we can make *lists of lists*, but by the substitution model we only deal with one dimension at a time: `L = [[4, 5, 6], [7, 8, 9]]` and `L[0][0] == 4`.)

We also saw list *slices*.

As we have seen, we can use indexing and slicing on 1D Numpy arrays, just like with lists. But for multidimensional arrays, we access elements using not a single number but a *tuple*.

Operation	Python list	Numpy array
Index	<code>L[int]</code>	<code>a[tuple]</code>
Shape	<code>len(L)</code>	<code>a.shape</code>

This reflects the move from a single number (length of a list) to a *tuple* (*shape* of an array). We can combine indexing and slicing in that tuple arbitrarily. That is called *fancy indexing*.

```
In [13]: print(X)
X[0, 1] # equivalent to X[(0, 1)]
```

```
[1. 2.]
[3. 4.]
```

```
Out[13]: 2.0
```

We can also extract a single row:

```
In [14]: X[0, :]
```

```
Out[14]: array([1., 2.])
```

Here `0` means "the first row" and `:` means "all columns" -- like writing `[:]` to mean "all elements" in a list slice, which we saw when studying lists. So we get all elements of the first row.

We get a single column in a similar way, and notice that it now "looks like a row".

```
In [15]: X[:, 0]
```

```
Out[15]: array([1., 3.])
```

As the logical conclusion of this, we'll see expressions such as `X[:, :]`. You'll say wait, that's not a *tuple*! But it is `:, :` effectively a *tuple* of slices, where each slice omits both start and end values, so they take on default values.

```
In [16]: print(X[:, :])
```

```
[1. 2.]
[3. 4.]
```

```
Out[16]: print(X[0:X.shape[0], 0:X.shape[1]])
```

```
[1. 2.]
[3. 4.]
```

Exercise: what does `X[0]` mean and why?

Reshaping

If we have, say an array of 10 numbers, we can ask for it to be reshaped as 5×2 or 2×5 . The numbers aren't changed -- it's just the *shape* of the table they are presented in:

```
In [11]: X = np.array(range(10))
print(X)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
In [12]: print(X.reshape((5, 2)))
```

```
[0 1]
[2 3]
[4 5]
[6 7]
[8 9]
```

```
In [13]: print(X.reshape((2, 5)))
```

```
[0 1 2 3 4]
[5 6 7 8 9]
```

```
In [14]: print(X.reshape((10, 1)))
```

```
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
```

```
In [15]: print(X.reshape((1, 10)))
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
In [16]: print(X.reshape((10, 1))) # a tuple of one element: (10,)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

Notice that `X` itself has not been altered:

```
In [17]: print(X)
```

```
Out[17]: [0 1 2 3 4 5 6 7 8 9]
```

A related idea is transposing, done using `.T`:

```
In [28]: X = np.array([[1.0, 2.0, 3.0],
                  [4.0, 5.0, 6.0]])
print(X)
print("") # put a blank line for clarity
print(X.T)
```

```
[[1. 2. 3.]
 [4. 5. 6.]]
```

```
[1. 4.]
[2. 5.]
[3. 6.]]
```

Exercise: If we have a 2D array and transpose it twice, what happens?

Matrix multiplication

As we know, many Numpy operations work *per-element*. A binary operation like `*` is straightforward when the two inputs are the same shape:

```
In [4]: X = np.array([[1.0, 2.0, 3.0],
                  [4.0, 5.0, 6.0]])
Y = np.array([[0.01, 0.1, 1.0],
                  [10.0, 100.0, 1000.0]])
print(X * Y)
```

```
[[1.e-02 2.e-01 3.e+00]
 [4.e+01 5.e+02 6.e+03]]
```

That is sometimes called the *element-wise product* or *Hadamard product*. It requires the arrays to have the same size and shape.

Broadcasting

Numpy also allows multiplication or other functions to work when the two array shapes are *broadcastable*. "the smaller array is 'broadcast' (reused) across the larger array so that they have compatible shapes". This works as long as corresponding dimensions are *equal*, or one of them is equal to 1, or not present. We line the shapes up "right-aligned":

A (2d array): 2 x 4
B (1d array): 4
Result (2d array): 2 x 4

```
In [35]: A = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]); print(A.shape)
```

```
B = np.array([10, 11, 12, 13]); print(B.shape)
```

```
C = A + B
print(C); print(C.shape)
```

```
(2, 4)
```

```
(4,)
```

```
[11 13 15 17]
```

```
[15 17 19 21]
```

```
(2, 4)
```

A contrived example:

A (4d array): 8 x 1 x 6 x 1
B (3d array): 7 x 1 x 5
Result (4d array): 8 x 7 x 6 x 5

```
In [36]: A = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]); print(A.shape)
```

```
B = np.array([10, 11, 12, 13]); print(B.shape)
```

```
C = A + B
print(C); print(C.shape)
```

```
(2, 4)
```

```
(2,)
```

```
-----
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-36-fd5e775c3a0> in <module>
      1 A = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
      2 B = np.array([10, 11, 12, 13])
      3 C = A + B
      4
      5 C
      6
      7
      8
      9
      10
      11
      12
      13
      14
      15
      16
      17
      18
      19
      20
      21
      22
      23
      24
      25
      26
      27
      28
      29
      30
      31
      32
      33
      34
      35
      36
      37
      38
      39
      40
      41
      42
      43
      44
      45
      46
      47
      48
      49
      50
      51
      52
      53
      54
      55
      56
      57
      58
      59
      60
      61
      62
      63
      64
      65
      66
      67
      68
      69
      70
      71
      72
      73
      74
      75
      76
      77
      78
      79
      80
      81
      82
      83
      84
      85
      86
      87
      88
      89
      90
      91
      92
      93
      94
      95
      96
      97
      98
      99
      100
      101
      102
      103
      104
      105
      106
      107
      108
      109
      110
      111
      112
      113
      114
      115
      116
      117
      118
      119
      120
      121
      122
      123
      124
      125
      126
      127
      128
      129
      130
      131
      132
      133
      134
      135
      136
      137
      138
      139
      140
      141
      142
      143
      144
      145
      146
      147
      148
      149
      150
      151
      152
      153
      154
      155
      156
      157
      158
      159
      160
      161
      162
      163
      164
      165
      166
      167
      168
      169
      170
      171
      172
      173
      174
      175
      176
      177
      178
      179
      180
      181
      182
      183
      184
      185
      186
      187
      188
      189
      190
      191
      192
      193
      194
      195
      196
      197
      198
      199
      200
      201
      202
      203
      204
      205
      206
      207
      208
      209
      210
      211
      212
      213
      214
      215
      216
      217
      218
      219
      220
      221
      222
      223
      224
      225
      226
      227
      228
      229
      230
      231
      232
      233
      234
      235
      236
      237
      238
      239
      240
      241
      242
      243
      244
      245
      246
      247
      248
      249
      250
      251
      252
      253
      254
      255
      256
      257
      258
      259
      260
      261
      262
      263
      264
      265
      266
      267
      268
      269
      270
      271
      272
      273
      274
      275
      276
      277
      278
      279
      280
      281
      282
      283
      284
      285
      286
      287
      288
      289
      290
      291
      292
      293
      294
      295
      296
      297
      298
      299
      300
      301
      302
      303
      304
      305
      306
      307
      308
      309
      310
      311
      312
      313
      314
      315
      316
      317
      318
      319
      320
      321
      322
      323
      324
      325
      326
      327
      328
      329
      330
      331
      332
      333
      334
      335
      336
      337
      338
      339
      340
      341
      342
      343
      344
      345
      346
      347
      348
      349
      350
      351
      352
      353
      354
      355
      356
      357
      358
      359
      360
      361
      362
      363
      364
      365
      366
      367
      368
      369
      370
      371
      372
      373
      374
      375
      376
      377
      378
      379
      380
      381
      382
      383
      384
      385
      386
      387
      388
      389
      390
      391
      392
      393
      394
      395
      396
      397
      398
      399
      400
      401
      402
      403
      404
      405
      406
      407
      408
      409
      410
      411
      412
      413
      414
      415
      416
      417
      418
      419
      420
      421
      422
      423
      424
      425
      426
      427
      428
      429
      430
      431
      432
      433
      434
      435
      436
      437
      438
      439
      440
      441
      442
      443
      444
      445
      446
      447
      448
      449
      450
      451
      452
      453
      454
      455
      456
      457
      458
      459
      460
      461
      462
      463
      464
      465
      466
      467
      468
      469
      470
      471
      472
      473
      474
      475
      476
     
```


(This notebook is partly based on Bauckhage, *N*,
from <https://www.researchgate.net/publication/2>

The image above is the *Julia set*, a well-known *fractal*. As we may know, a fractal is a mathematical object which has some properties of *self-similarity* at different scales. If we zoom in to the image, we'll see similar objects re-appearing at a smaller scale.

The Julia set is defined as follows. For each point $z_0 = a + bi$ in the complex plane, we apply an iterated map $z_{n+1} = z_n^2 + c$.

At each step, we look at the *modulus* (size)

```
c = -0.065 + 0.66j # Julia constant: using 'j'  
def f(z):  
    return z**2 + c # complex arithmetic is built-in  
def escape_time(z0, its=20):
```

```
    z = z0
    for i in range(its):
        if abs(z) > 2:
            return i
        else:
            z = f(z)
```

```
    return its # deem escape time equal to number of iterations, as a default  
escape_time(0.4 + -0.1j)
```

11

Now, some points might take a long time to escape (if ever). We can't keep iterating forever.

We use a maximum of 20 iterations above. Any point z_0 which doesn't have an escape time escape time of 20, for the purposes of plotting.

```
In [2]: def test_escape_and_return(z0, its=20):
    z = z0
    for i in range(its):
        if abs(z) > 2:
            print("outside")
            z = f(z)
        else:
            print("inside")
            z = f(z)
    return its # deem escape time equal to number of iterations, as a default
test_escape_and_return(0.4 + 0.5j, 20)
```

inside
inside
inside
inside
inside
outside
outside
outside
outside

outside
inside
inside
inside
inside

Wait a minute, that's not what we expected! The point escaped, then came back in. **Exercise:** debug this to understand what happened. Hint: print `abs(z)` at each iteration. What is the special case in which the `if`-test becomes False?

our materials.

So far, we have looked at one point at a time. The beautiful images emerge when we consider escape times. We'll use Numpy and Matplotlib.

```
In [3]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
```

```
In [4]: def JuliaNaive(zmin, zmax, m, n, tmax=256):
    xs = np.linspace(zmin.real, zmax.real, n)
    ys = np.linspace(zmin.imag, zmax.imag, m)
    J = np.zeros((m, n))
    for r, v in enumerate(ys):
```

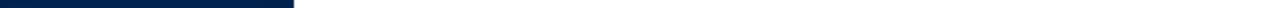
```
    for c, x in enumerate(xs):
        z = x + 1j * y
        t[r, c] = escape_time(z, tmax)
```

```
return J
```

For the Julia set, the following points give suitable boundaries for the grid:

```
zmin = -1.3 - 1j * 1.3 # ie -1.3 - 1.3i  
zmax = 1.3 + 1j * 1.3 # ie 1.3 + 1.3i
```

```
# warning, this will be slow for large values of m and n
J = JuliaNaive(zmin, zmax, m=30, n=30)
```



```
def Julia(zmin, zmax, m, n, tmax=256):
    xs = np.linspace(zmin.real, zmax.real, n)
    vs = np.linspace(zmin.imag, zmax.imag, m)
```

```
J = np.ones(Z.shape) * tmax # initialise all points with max escape time
for t in range(tmax):
    mask = np.abs(Z) <= 2.
    Z[mask] = f(Z[mask])
    J[~mask] -= 1 # decrement escape time of points which have escaped
```

- `J` starts with value `tmax` everywhere

- set `mask` where z has not escaped
- decrement `J` wherever z has escaped
- mask runs `f` only where z has not escaped

operator. The way to think of this is: all points start with escape time `tmax=256`. A point which escapes straight away will be decremented 256 times, so will end up with escape time 0, which is correct.

```
plt.imshow(J, cmap=cm.cividis);
```



The image shows a vertical decorative border on the left side of a page. The border is a solid dark blue color and features a repeating pattern of stylized yellow floral or mandala motifs. These motifs are arranged in a staggered, overlapping fashion along the edge. The design is intricate, with multiple layers of petals or petals-like shapes forming a complex, organic pattern.

Exercise: try out different values of c to see what happens. Zoom in by changing `zmin` and `zmax`. Try different colourmaps using `cmap` : see <https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html>.

```
In [20]: def Mandelbrot(zmin, zmax, m, n, tmax=256):
    xs = np.linspace(zmin.real, zmax.real, n)
    ys = np.linspace(zmin.imag, zmax.imag, m)
    X, Y = np.meshgrid(xs, ys)
```

```
M = np.ones(Z.shape) * tmax
for t in range(tmax):
    mask = np.abs(Z) <= 2.
    Z[mask] = Z[mask]**2 + C[mask]
    M[mask] = 1
```

```
In [21]: zmin = -2.3 - 1.5j
        zmax = 0.7 + 1.5j
        M = Mandelbrot(zmin, zmax, 1024, 1024)
```

A grayscale heatmap visualization of a 2D matrix. The matrix has a dark background with a single, sharp, localized peak in red color located at the center. This red area represents the maximum value in the matrix, which is 1. The surrounding matrix elements have values ranging from 0 to 1, with higher values represented by lighter shades of gray.

Where will we use this?

- Numberphile on Feigenbaum's constant (related to iterated functions in \mathbb{R} instead of \mathbb{C}) and eventually mentions

- Silja/Mandelbrot fractals. <https://www.youtube.com/watch?v=LHTE4Mg8EQ>



Numpy: Input/Output

Reading and writing data with Numpy

To read and write Numpy arrays, we use `np.load()` and `np.save()` like this:

```
In [1]: import numpy as np
X = np.random.random((5, 2))
# .npy is a convention for the suffix
np.save("data/mytest.npy", X)
Y = np.load("data/mytest.npy")
print(Y)
```

[[0.41738414 0.50075328]
[0.22414113 0.54842202]
[0.64288255 0.01291776]
[0.39759689 0.31183326]
[0.264635 0.17048049]]

The above stores the data in a compressed format, not human-readable. Sometimes it's nice to store in a CSV or similar format instead. `np.savetxt()` and `np.loadtxt()` have lots of options for controlling formatting.

```
In [2]: np.savetxt("data/mytest.csv", X,
               delimiter=',', header="a, b",
               fmt=".3f")
```

```
In [3]: # let's look at the file itself:
# if this doesn't work, just open
# data/mytest.csv in a text editor
# the ! syntax allows us to type a
# shell command in IPython.
# 'cat' is not a Python command
!cat "data/mytest.csv"
```

a, b
0.417,0.501
0.224,0.548
0.643,0.013
0.398,0.312
0.265,0.170

```
In [4]: Y = np.loadtxt("data/mytest.csv",
                  delimiter=',', comments="#") # now load it
print(Y)
```

[[0.417 0.501]
[0.224 0.548]
[0.643 0.013]
[0.398 0.312]
[0.265 0.17]]

Exercises: Here are 100 Numpy exercises. Many of these are far too difficult for our purposes here.

<https://github.com/rougier/numpy-100>

Pandas

- Reading and writing data with Pandas
- Manipulating data with Pandas
- Prerequisites: Numpy

Pandas is a library for data manipulation in Python.

```
In [1]: import pandas as pd
```

The central object is the Pandas `DataFrame`, but it is built on top of the Pandas `Series` so we'll look at that first.

A `Series` is like a 1D Numpy `array`:

```
In [2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
```

```
Out[2]: 0    0.25  
1    0.50  
2    0.75  
3    1.00  
dtype: float64
```

Also, the values can be any type (but usually homogeneous):

```
In [3]: data = pd.Series(["a", "b", "c", "d"])
```

```
Out[3]: 0    a  
1    b  
2    c  
3    d  
dtype: object
```

The indices can also be anything -- in contrast to a Numpy array -- not just contiguous integers starting from 0.

```
In [4]: data = pd.Series([0.25, 0.5, 0.75, 1.0],  
                     index=[4, 2, 4/3, 1])
```

```
In [5]: data
```

```
Out[5]: 4    0.25  
2    0.50  
0.75  
1    1.00  
dtype: float64
```

The indices don't have to be unique because location is maintained:

```
In [6]: data = pd.Series([0.25, 0.5, 0.75, 1.0],  
                     index=[4, 2, 2, 1])
```

```
Out[6]: 4    0.25  
2    0.50  
0.75  
1    1.00  
dtype: float64
```

The indices will usually be of homogeneous types but it's not strictly required.

DataFrames

A good way to think of a DataFrame is: it's like a `dict` where the keys are strings (column headers) and the values are Series with a common index.

```
In [7]: df = pd.DataFrame({"a": [4, 5, 6, 7],  
                      "b": [0.4, 0.5, 0.6, 0.7]},  
                      index=[10, 11, 12, 13])
```

```
Out[7]:   a    b  
10  4  0.4  
11  5  0.5  
12  6  0.6  
13  7  0.7
```

```
In [8]: type(df["a"])
```

```
Out[8]: pandas.core.series.Series
```

```
In [9]: df["a"].index
```

```
Out[9]: Int64Index([10, 11, 12, 13], dtype='int64')
```

The `id()` function gets the location of an object in memory. So this proves that the entire DataFrame shares a single `index`:

```
In [10]: id(df["a"].index) == id(df["b"].index)
```

```
Out[10]: True
```

"NumPy arrays have one dtype for the entire array, while pandas DataFrames have one dtype per column" (from https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html). So, DataFrames are suitable for "rectangular", spreadsheet-like data with potentially a different data type in each column. So, we'll now see how to create a DataFrame by reading a file in a spreadsheet-like format.

Reading data with Pandas

The file we're going to use as an example is `data/irish_cities.txt`, available in Blackboard. The columns are separated by tab characters. Also, the file has a `header` which is not part of the data proper.

```
In [11]: # ! introduces a shell command  
!head data/irish_cities.txt
```

```
# Copied from http://www.tageo.com/index-e-ei-cities-IE.htm -- James McDermott  
#Rank  City      Population (2000)  Latitude (DD)  Longitude (DD)  
1     Dublin    1027900    53.330    -6.250  
2     Cork      195400     51.900    -8.470  
3     Limerick   85700     52.670    -8.630  
4     Galway    53.280    -9.060  
5     Waterford 48300     52.260    -7.130  
6     Dundalk   34400     51.020    -6.420  
7     Bray      32300     53.210    -6.110  
8     Drogheda  28800     53.720    -6.360
```

Pandas `read_csv` makes it easy to deal with all this. We use `sep="\t"` to say the file is `tab`-separated. And we tell it to skip the first row, which is just a comment. It then correctly uses the second row to guess the names for the columns.

```
In [12]: # "\t" -> tab-separated  
df = pd.read_csv("data/irish_cities.txt",  
                 sep="\t",  
                 skiprows=1)
```

```
In [13]: df.head() # handy for a quick look, nice formatting in Notebook
```

```
Out[13]:   #Rank  City  Population (2000)  Latitude (DD)  Longitude (DD)
```

```
Out[13]: 0     1  Dublin    1027900    53.33    -6.25  
1     2  Cork      195400     51.90    -8.47  
2     3  Limerick   85700     52.67    -8.63  
3     4  Galway    53.28    -9.06  
4     5  Waterford 48300     52.26    -7.13
```

Useful arguments for `read_csv`:

- The filename can be a file on disk or a URL.
- `sep="\t"`, `sep=" "`, and `sep=","` tell Pandas what character is used to separate the columns.
- `skiprows=3` means to ignore the first rows entirely.
- `names=["a", "b", "c"]` tells Pandas the names of the columns.
- But if the names of the columns are written at the top of the file, Pandas can figure them out.

Recall that a DataFrame is a bit like a dictionary mapping column headers to columns. We can *alter* the values in the way that you might expect, for example if we want our data to be in millions:

```
In [14]: df["Population (2000)"] /= 1000000.0
```

Then we should rename the column to reflect that change, and while we're at it we'll rename a few others.

```
In [15]: df = df.rename(columns={"#Rank": "Rank",  
                           "Population (2000)": "Population (M)",  
                           "Latitude (DD)": "Latitude",  
                           "Longitude (DD)": "Longitude"})
```

```
Out[15]:   Rank  City  Population (M)  Latitude  Longitude
```

```
Out[15]: 0     1  Dublin    1.0279    53.33    -6.25  
1     2  Cork      0.1954    51.90    -8.47  
2     3  Limerick   0.0857    52.67    -8.63  
3     4  Galway    0.0686    53.28    -9.06  
4     5  Waterford 0.0483    52.26    -7.13
```

Now, we can access the columns and treat them like Numpy arrays.

```
In [16]: df["Population (M)"].sum()
```

```
Out[16]: 2.0336000000000004
```

Aggregation functions like `mean()` on the whole DataFrame work across all numerical arguments -- which might not be meaningful, for all data:

```
In [17]: df.mean()
```

```
Out[17]: Bank      25.000000  
Population (M)  0.041563  
Latitude        53.042449  
Longitude       -7.391224  
dtype: float64
```

```
In [18]: df.describe() # handy, general description
```

```
Out[18]:   Rank  Population (M)  Latitude  Longitude
```

```
Out[18]:  count  49.000000  49.000000  49.000000  
  mean   25.000000  0.041563  53.042449  -7.391224  
  std    14.28869  0.146841  0.692864  1.083562  
  min    1.000000  0.006700  51.810000  -9.720000  
  25%   13.000000  0.008800  52.500000  -8.300000  
  50%   25.000000  0.013100  53.140000  -7.160000  
  75%   37.000000  0.019500  53.450000  -6.480000  
  max   49.000000  1.027900  54.940000  -6.050000
```

We can take subsets of the data, e.g.:

```
In [19]: df[df["Population (M)"] > 0.05]
```

```
Out[19]:   Rank  City  Population (M)  Latitude  Longitude
```

```
Out[19]: 0     1  Dublin    1.0279    53.33    -6.25  
1     2  Cork      0.1954    51.90    -8.47  
2     3  Limerick   0.0857    52.67    -8.63  
3     4  Galway    0.0686    53.28    -9.06
```

This works because `df["Population (M)"]` returns a Boolean array, which is then used to select a subset of `df`.

We can use Boolean operations for more powerful subsetting. Notice that we use a `single &` symbol here for `and`, and the round brackets are needed.

```
In [20]: df[(df["Population (M)"] > 0.05) &  
         (df["Longitude"] < -8)]
```

```
Out[20]:   Rank  City  Population (M)  Latitude  Longitude
```

```
Out[20]: 1     2  Cork      0.1954    51.90    -8.47  
2     3  Limerick   0.0857    52.67    -8.63  
3     4  Galway    0.0686    53.28    -9.06
```

Let's save to disk, and check what it looks like on disk.

```
In [21]: df.to_csv("data/sort_times.csv")
```

```
In [22]: !cat data/sort_times.csv
```

```
,#Rank  City  Population (M)  Latitude  Longitude
```

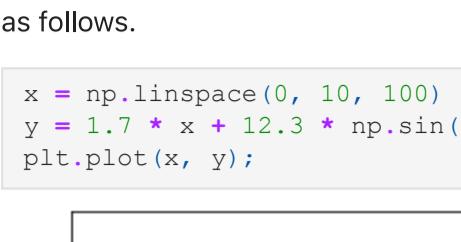
```
,0     1  Dublin    1.0279    53.33    -6.25  
1     2  Cork      0.1954    51.90    -8.47  
2     3  Limerick   0.0857    52.67    -8.63  
3     4  Galway    0.0686    53.28    -9.06
```

```
,4     5  Waterford 0.0483    52.26    -7.13
```

```
0.0000000000000001, 0.0000000000000002, 0.0000000000000003, 0.0000000000000004, 0.0000000000000005, 0.0000000000000006, 0.0000000000000007, 0.0000000000000008, 0.0000000000000009, 0.0000000000000010, 0.0000000000000011, 0.0000000000000012, 0.0000000000000013, 0.0000000000000014, 0.0000000000000015, 0.0000000000000016, 0.0000000000000017, 0.0000000000000018, 0.0000000000000019, 0.0000000000000020, 0.0000000000000021, 0.0000000000000022, 0.0000000000000023, 0.0000000000000024, 0.0000000000000025, 0.0000000000000026, 0.0000000000000027, 0.0000000000000028, 0.0000000000000029, 0.0000000000000030, 0.0000000000000031, 0.0000000000000032, 0.0000000000000033, 0.0000000000000034, 0.0000000000000035, 0.0000000000000036, 0.0000000000000037, 0.0000000000000038, 0.0000000000000039, 0.0000000000000040, 0.0000000000000041, 0.0000000000000042, 0.0000000000000043, 0.0000000000000044, 0.0000000000000045, 0.0000000000000046, 0.0000000000000047, 0.0000000000000048, 0.0000000000000049, 0.0000000000000050, 0.0000000000000051, 0.0000000000000052, 0.0000000000000053, 0.0000000000000054, 0.0000000000000055, 0.0000000000000056, 0.0000000000000057, 0.0000000000000058, 0.0000000000000059, 0.0000000000000060, 0.0000000000000061, 0.0000000000000062, 0.0000000000000063, 0.0000000000000064, 0.0000000000000065, 0.0000000000000066, 0.0000000000000067, 0.0000000000000068, 0.0000000000000069, 0.0000000000000070, 0.0000000000000071, 0.0000000000000072, 0.0000000000000073, 0.0000000000000074, 0.0000000000000075, 0.0000000000000076, 0.0000000000000077, 0.0000000000000078, 0.0000000000000079, 0.0000000000000080, 0.0000000000000081, 0.0000000000000082, 0.0000000000000083, 0.0000000000000084, 0.0000000000000085, 0.0000000000000086, 0.0000000000000087, 0.0000000000000088, 0.0000000000000089, 0.0000000000000090, 0.0000000000000091, 0.0000000000000092, 0.0000000000000093, 0.0000000000000094, 0.0000000000000095, 0.0000000000000096, 0.0000000000000097, 0.0000000000000098, 0.0000000000000099, 0.0000000000000100, 0.0000000000000101, 0.0000000000000102, 0.0000000000000103, 0.0000000000000104, 0.0000000000000105, 0.0000000000000106, 0.0000000000000107, 0.0000000000000108, 0.0000000000000109, 0.0000000000000110, 0.0000000000000111, 0.0000000000000112, 0.0000000000000113, 0
```


Plotting with Matplotlib

Data visualisation is an important step in almost every data analysis. In Python there are many plotting libraries, which all accept Numpy arrays as input data. The main plotting library is called Matplotlib, and we will later see another built on top of it, called Seaborn.

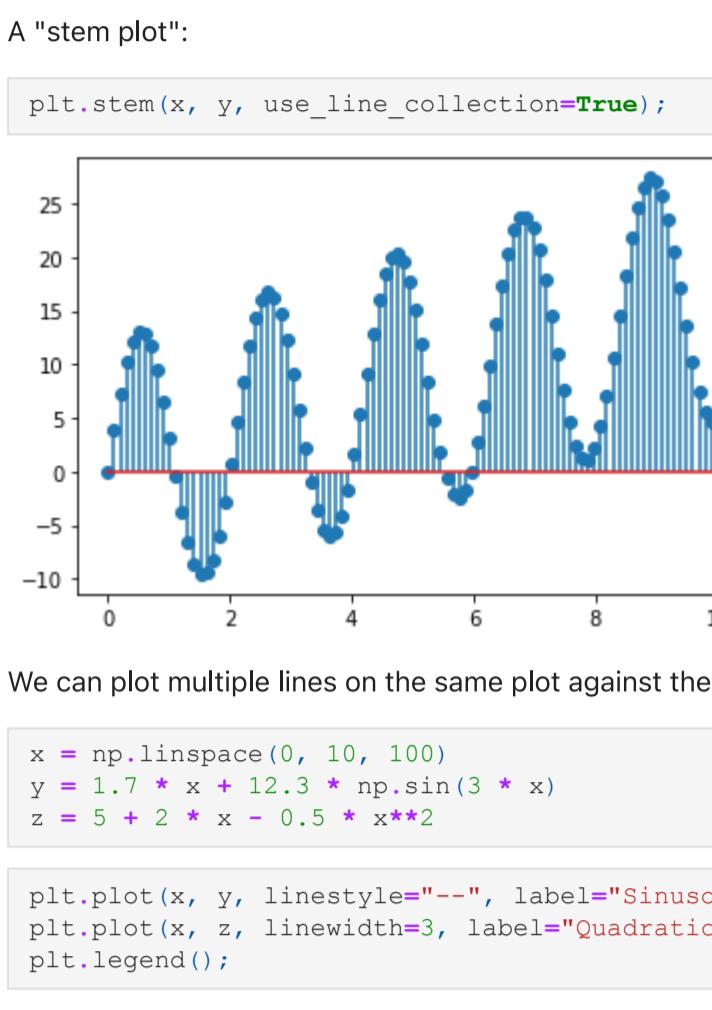


We can plot data directly inside the Jupyter Notebook or inside the IPython console in Spyder. We import and also use a Jupyter magic command (starts with %) to enable the inline plotting.

```
In [1]: import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
# this line tells the notebook to show plots "inline"  
%matplotlib inline
```

Let's say we want to plot the formula $y = 1.7x + 12.3 \sin(3x)$ for all values of x from 0 to 10. Numpy and Matplotlib make it easy as follows.

```
In [2]: x = np.linspace(0, 10, 100)  
y = 1.7 * x + 12.3 * np.sin(3 * x)  
plt.plot(x, y);
```

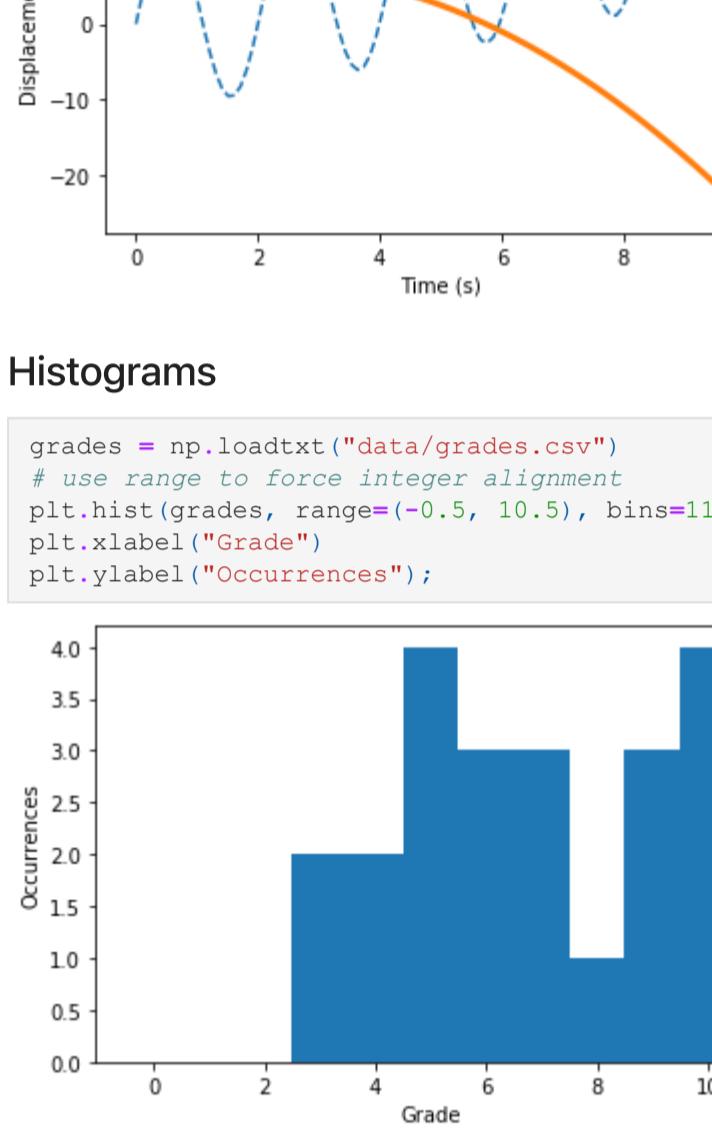


Exercise: why did we put a semi-colon at the end of the last line?

Exercise: Plot the formula $0.2 * (0.7^x)$ for $x \in [0, 20]$.

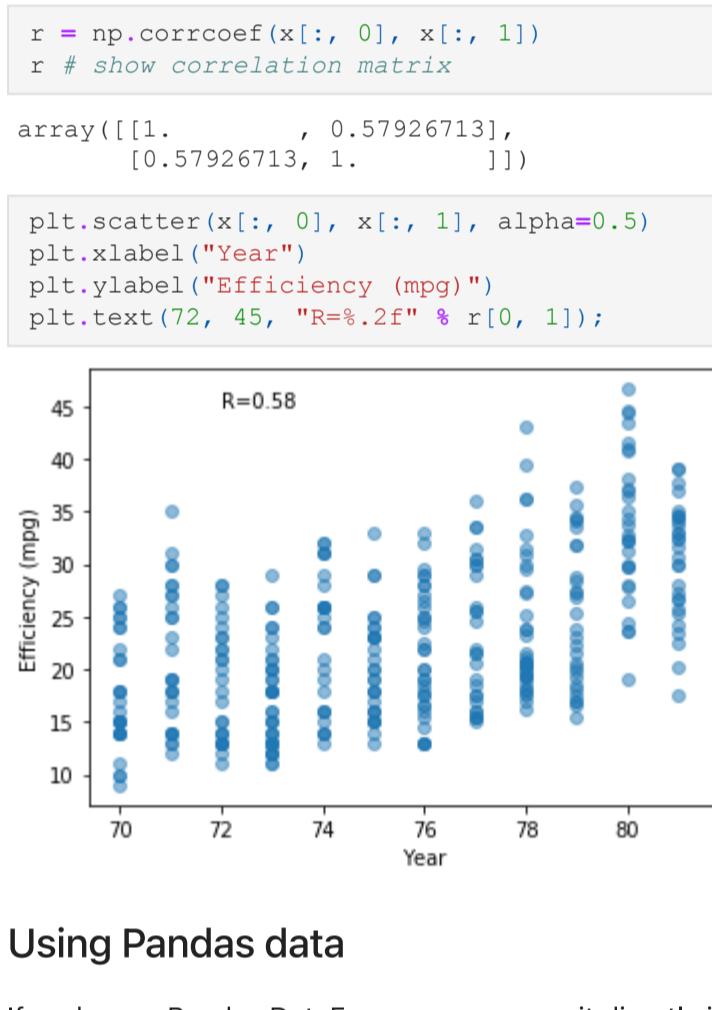
Of course, we can change the plotting style:

```
In [3]: # 's' -> squares instead of lines  
# alpha -> transparency  
# markersize -> size  
# markeredgewidth=0 -> remove borders on markers  
plt.plot(x, y, "s", color="g", alpha=0.5,  
         markersize=15, markeredgewidth=0);
```



A "stem plot":

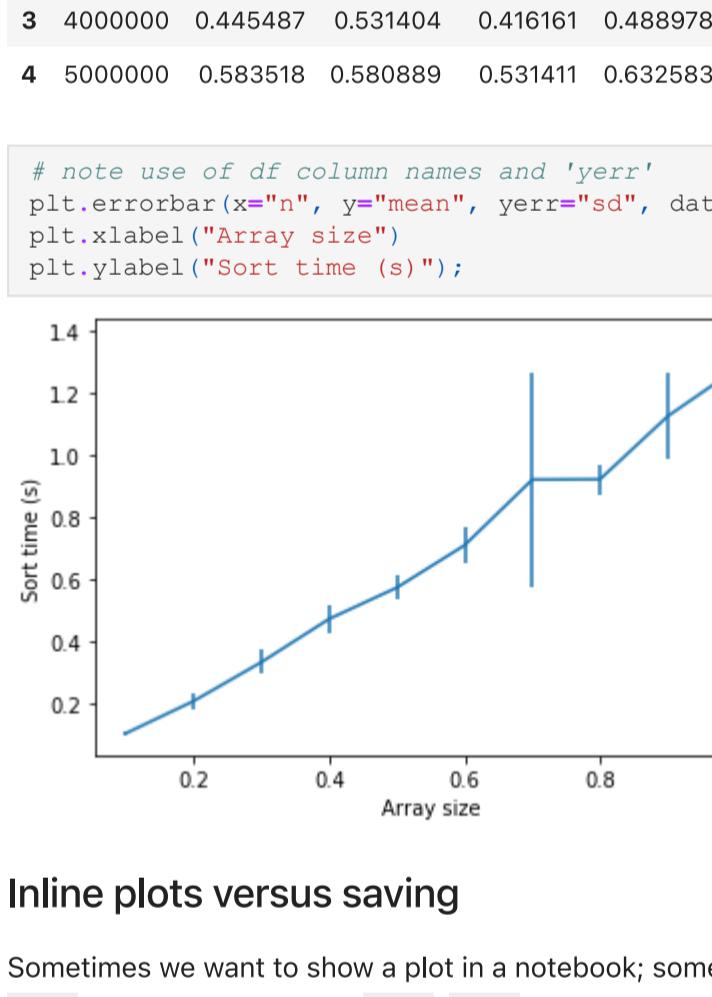
```
In [4]: plt.stem(x, y, use_line_collection=True);
```



We can plot multiple lines on the same plot against the same x -axis, using different styles for each, and add a legend:

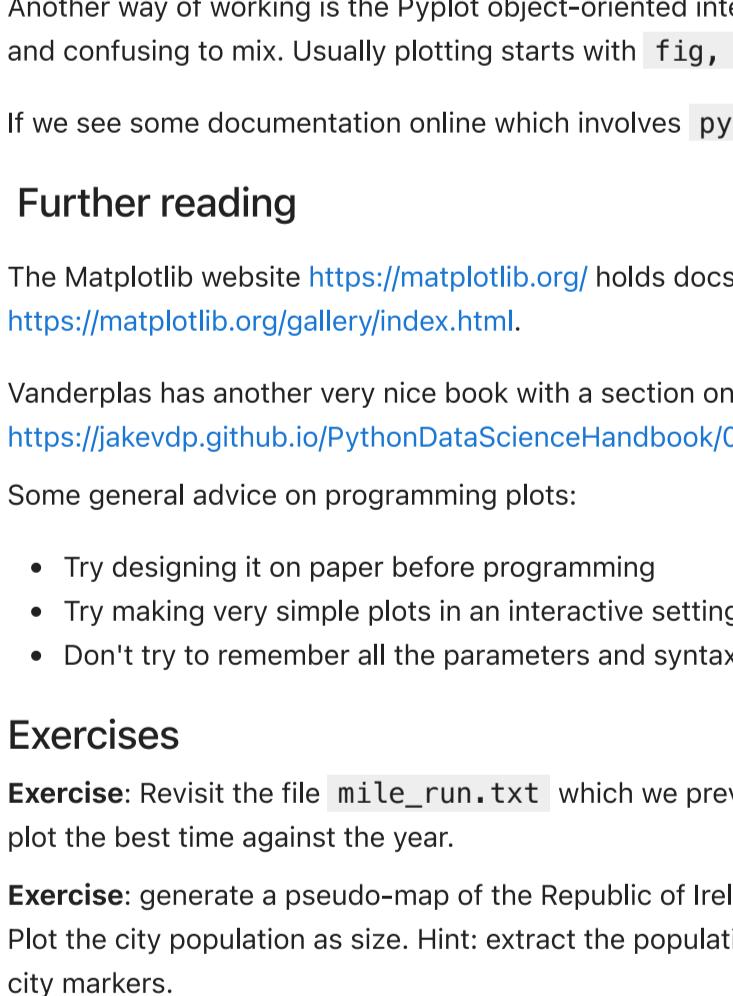
```
In [5]: x = np.linspace(0, 10, 100)  
y = 1.7 * x + 12.3 * np.sin(3 * x)  
z = 5 + 2 * x - 0.5 * x**2
```

```
In [6]: plt.plot(x, y, linestyle="--", label="Sinusoid")  
plt.plot(x, z, linewidth=3, label="Quadratic")  
plt.legend();
```



And we should always label our axes.

```
In [7]: plt.plot(x, y, linestyle="--", label="Sinusoid")  
plt.plot(x, z, linewidth=3, label="Quadratic")  
plt.legend()  
plt.xlabel("Time (s)")  
plt.ylabel("Displacement (m)");
```



Histograms

```
In [8]: grades = np.loadtxt("data/grades.csv")  
# use range to force integer alignment  
plt.hist(grades, range=(-0.5, 10.5), bins=11)  
plt.xlabel("Grade")  
plt.ylabel("Occurrences");
```



Scatter plots

To understand the relationship between a pair of variables, we often start with scatterplots and correlation values. We'll take an example extracted from a dataset with efficiency of cars (miles per gallon). We'll use `np.corrcoef` and then `plt.scatter`.

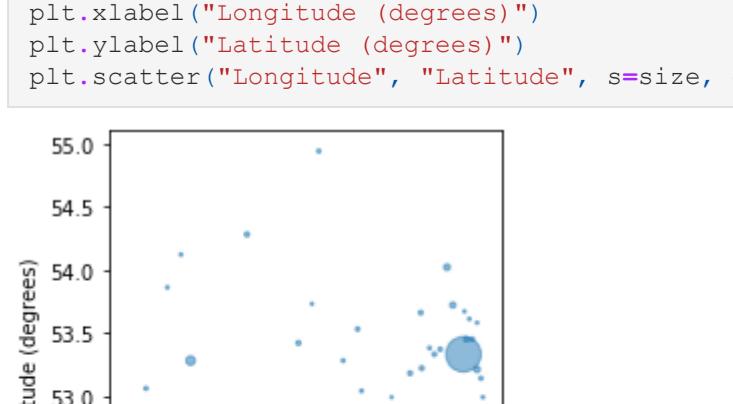
```
In [9]: x = np.loadtxt("data/mpg_extract.csv", delimiter=",")  
x.shape # 398 cars, 2 variables: year and efficiency
```

```
Out[9]: (398, 2)
```

```
In [10]: r = np.corrcoef(x[:, 0], x[:, 1])  
r # show correlation matrix
```

```
Out[10]: array([[1.0, 0.57926713],  
                 [0.57926713, 1.0]])
```

```
In [11]: plt.scatter(x[:, 0], x[:, 1], alpha=0.5)  
plt.xlabel("Year")  
plt.ylabel("Efficiency (mpg)")  
plt.text(72, 45, "R=% .2f" % r[0, 1]);
```



Using Pandas data

If we have a Pandas DataFrame, we can use it directly instead of extracting columns. Compare:

```
plt.plot("xvar", "yvar", data=df)
```

```
plt.plot(df["xvar"], df["yvar"])
```

Example: recall the data file `data/sort_times.csv`, which we created in the notebook about Pandas. The first column was the independent variable `n` (the size of the array being sorted), and the other columns were multiple observations of the dependent variable `y`, i.e. the `elapsed time` in seconds. We have already calculated the mean and standard deviation across observations, and stored them as new columns. Now, using `plt.errorbar`: plot the mean of `y` against `n`, with errorbars given by the standard deviation of `y`.

```
In [12]: # index_col=0 tells Pandas that the first column is an index  
# not a data column  
df = pd.read_csv("data/sort_times.csv", index_col=0)  
df.head() # take a look
```

```
Out[12]: n      run0      run1      run2      run3      run4      mean      sd
```

0	1000000	0.099246	0.109996	0.101855	0.100453	0.114092	0.105128	0.006543
1	2000000	0.197066	0.194505	0.203326	0.193344	0.256533	0.208955	0.026876
2	3000000	0.302803	0.300889	0.316597	0.365341	0.385361	0.334198	0.038706
3	4000000	0.445487	0.531404	0.416161	0.488978	0.485014	0.473409	0.044152
4	5000000	0.583518	0.580889	0.531411	0.632583	0.543272	0.574335	0.039773

```
In [13]: # note use of df column names and 'yerr'  
plt.errorbar(n, mean, yerr=sd, data=df)
```

```
plt.xlabel("Array size")  
plt.ylabel("Sort time (s)");
```


Warning

There are at least two different ways of using Matplotlib:

- Pyplot stateful interface (what we've been using)
- Pyplot object-oriented interface

We've been using the Pyplot stateful interface. A command like `plt.xlabel` changes the `x` label of whatever figure `plt` is currently working on.

Another way of working is the Pyplot object-oriented interface, which is a bit more powerful than the interface we've been using and confusing to mix. Usually plotting starts with `fig, ax = plt.subplots()`. See [discussion here](#).

If we see some documentation online which involves `pylab`, keep away!

Further reading

The Matplotlib website <https://matplotlib.org/> holds docs, and the gallery works as a cookbook: <https://matplotlib.org/gallery/index.html>

Vanderplas has another very nice book with a section on Matplotlib: <https://jakevdp.github.io/PythonDataScienceHandbook/04.00-introduction-to-matplotlib.html>

Some general advice on programming plots:

- Try designing it on paper before programming
- Try making very simple plots in an interactive setting (Jupyter Notebook or Spyder) and then iterating to refine it
- Don't try to remember all the parameters and syntax -- just know how to look for plots e.g. in the galleries.

Exercises

Exercise: Revisit the file `mile_run.txt` which we previously saw in the Pandas notebook. Read it in to a Pandas DataFrame and plot the best time against the year.

Exercise: generate a pseudo-map of the Republic of Ireland by plotting the longitude and latitude of `data/irish_cities.txt`. Plot the city population as size. Hint: extract the population column to an array `size` and manipulate it to give good sizes for the city markers.

```
In [15]: head -n 15 data/mile_run.txt # show the first 15 lines
```

```
# The following gives the winning time, in seconds, for the mile run.  
# The year in which occurred, the name of the runner, and the runner's  
# nationality are also given.
```

```
# Copied from http://www.math.hope.edu/swanson/data/mile_run.txt and  
# edited format -- James McDermott.
```

0	1913	John Paul Jones	USA	254.4
1	1915	Norman Taber	USA	252.6
2	1923	Paavo Nurmi	Finland	250.4
3	1931	Jules Ladoumegue	France	249.2
4	1933	Jack Lovelock	New Zealand	247.6
5	1934	Glen Cunningham	USA	246.8
6	1937	Sydney Wooderson	UK	246.4
7	1948	Frank Shorter	USA	245.8
8	1952	Frank Shorter	USA	245.2
9	1956	Frank Shorter	USA	244.8
10	1960	Frank Shorter	USA	244.4
11	1964	Frank Shorter	USA	244.0
12	1968	Frank Shorter	USA	243.6
13	1972	Frank Shorter	USA	243.2
14	1976	Frank Shorter	USA	242.8
15	1980	Frank Shorter	USA	242.4

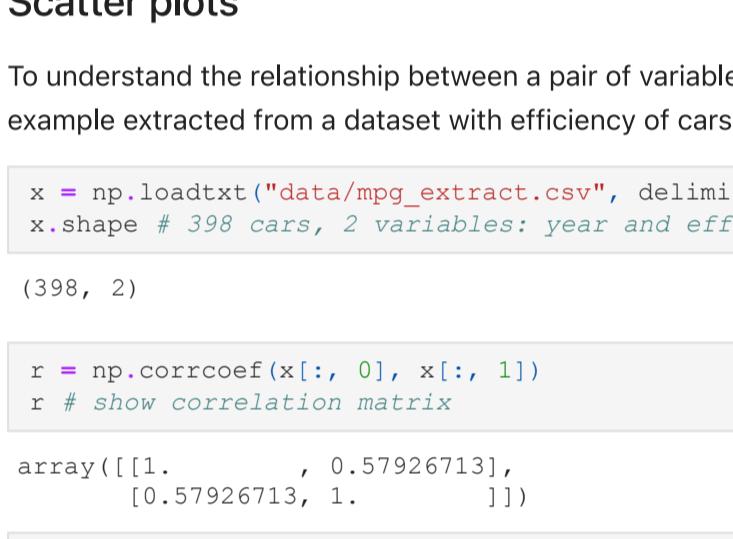


Solutions

```
In [16]: df = pd.read_csv("data/mile_run.txt", delimiter=" ", skiprows=7)
```

```
df.head()
```

Year	Athlete	Nationality	Time(seconds)
1913	John Paul Jones	USA	254.4
1915	Norman Taber	USA	252.6
1923	Paavo Nurmi	Finland	250.4
1931	Jules Ladoumegue	France	249.2
1933	Jack Lovelock	New Zealand	247.6
1934	Glen Cunningham	USA	246.8
1937	Sydney Wooderson	UK	246.4
1948	Frank Shorter	USA	245.8
1952	Frank Shorter	USA	245.2
1956	Frank Shorter	USA	244.8
1960	Frank Shorter	USA	244.4
1964	Frank Shorter	USA	244.0
1968	Frank Shorter	USA	243.6
1972	Frank Shorter	USA	243.2
1976	Frank Shorter	USA	242.8
1980	Frank Shorter	USA	242.4

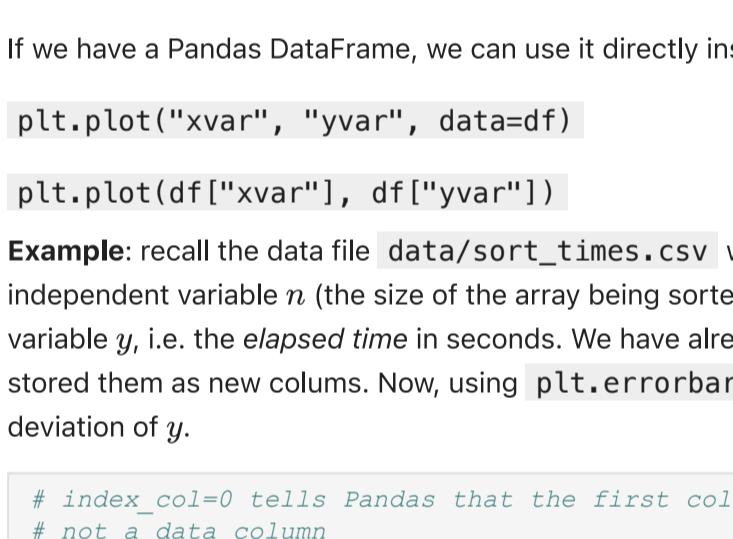


Inline plots versus saving

Sometimes we want to show a plot in a notebook; sometimes we want to save to disk. The following code will save the plot to a `.png` file. You can also use `.pdf`, `.svg`, and other formats. As we may know, `.png` is a bitmap format suitable for the web; `.pdf` is a vector format suitable for documents.

```
In [14]: plt.savefig("data/sort_times.png")
```

```
plt.close()
```



Plotting with Seaborn

Seaborn

"Seaborn is a library for making statistical graphics in Python. It is built on top of matplotlib and closely integrated with pandas data structures." -- from Seaborn docs.

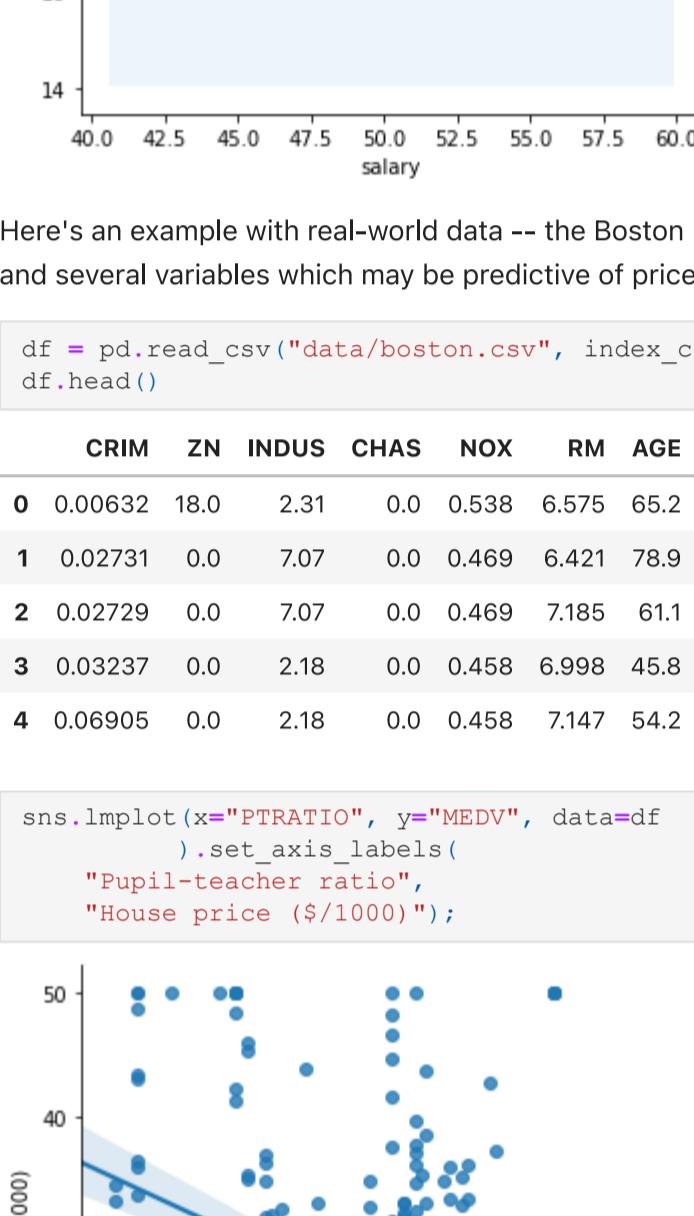
- Default styles are nicer
- Some "higher-level" plots are builtin which would take a lot of work in Matplotlib
- Better integration with Pandas

(But Matplotlib has caught up in some ways)

```
In [2]: import numpy as np
import pandas as pd
import seaborn as sns
# this line tells the notebook to show plots
# "inline". seaborn calls matplotlib behind
# the scenes, so this line still just says
# %matplotlib, not %seaborn!
%matplotlib inline
```

In Seaborn, as in Matplotlib, we can pass a Pandas DataFrame as `data`, and choose which columns to plot with kwargs `x` and `y`:

```
In [3]: df = np.random.multivariate_normal(
    [50, 20], [[5, 2], [2, 2]], size=2000) # some fake data
df = pd.DataFrame(df,
                   columns=["salary", "spending"])
# kde = kernel density estimation
sns.jointplot(x="salary", y="spending",
               data=df, kind="kde");
```

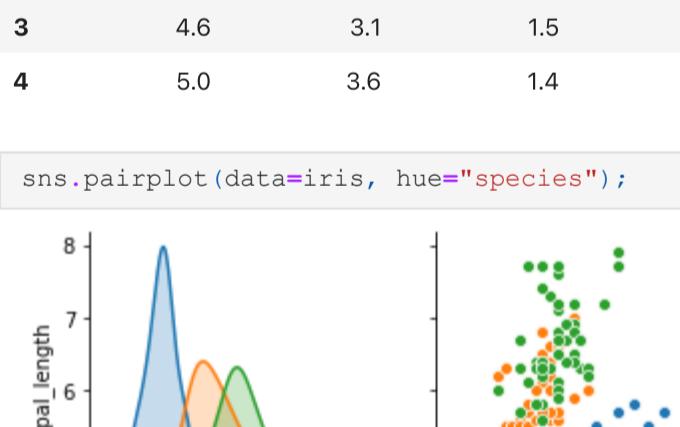


Here's an example with real-world data -- the Boston Housing dataset, which has house prices ("MEDV", in thousands of dollars) and several variables which may be predictive of prices.

```
In [4]: df = pd.read_csv("data/boston.csv", index_col=0)
df.head()
```

```
Out[4]:   CRIM      ZN     INDUS    CHAS      NOX      RM      AGE      DIS      RAD      TAX      PTRATIO      B      LSTAT      MEDV
0  0.00632  18.0    2.31    0.0  0.538  6.575  65.2  4.0900  1.0  296.0  15.3  396.90  4.98  24.0
1  0.02731  0.0    7.07    0.0  0.469  6.421  78.9  4.9671  2.0  242.0  17.8  396.90  9.14  21.6
2  0.02729  0.0    7.07    0.0  0.469  7.185  61.1  4.9671  2.0  242.0  17.8  392.83  4.03  34.7
3  0.03237  0.0    2.18    0.0  0.458  6.998  45.8  6.0622  3.0  222.0  18.7  394.63  2.94  33.4
4  0.06905  0.0    2.18    0.0  0.458  7.147  54.2  6.0622  3.0  222.0  18.7  396.90  5.33  36.2
```

```
In [5]: sns.lmplot(x="PTRATIO", y="MEDV", data=df
                 ).set_axis_labels(
    "Pupil-teacher ratio",
    "House price ($/1000)");
```



`lmplot` (linear model plot) gives us:

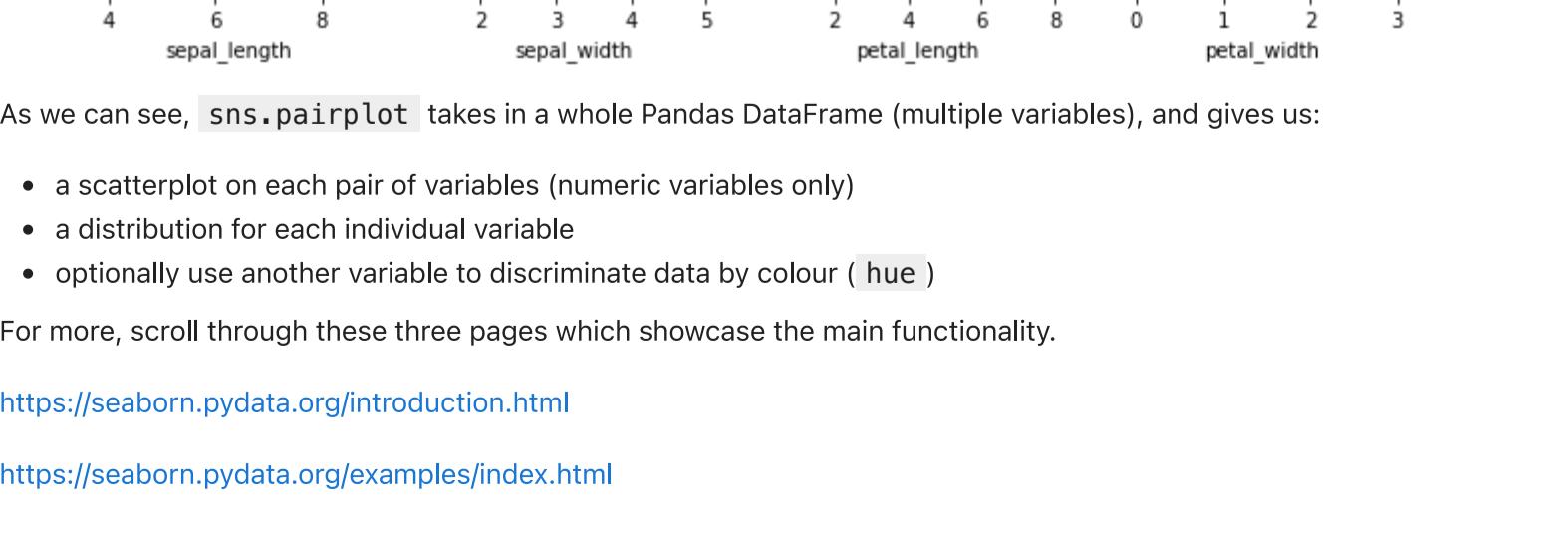
- data as a scatter plot
- a linear regression model, i.e. a trendline (linear regression is studied in other modules)
- a shaded confidence interval on the linear model

One more example, this time the Iris dataset which has measurements from a few species of Iris (a type of flower).

```
In [6]: # some example datasets are built-in
iris = sns.load_dataset("iris")
iris.head()
```

```
Out[6]:   sepal_length  sepal_width  petal_length  petal_width  species
0          5.1         3.5          1.4          0.2   setosa
1          4.9         3.0          1.4          0.2   setosa
2          4.7         3.2          1.3          0.2   setosa
3          4.6         3.1          1.5          0.2   setosa
4          5.0         3.6          1.4          0.2   setosa
```

```
In [9]: sns.pairplot(data=iris, hue="species");
```



As we can see, `sns.pairplot` takes in a whole Pandas DataFrame (multiple variables), and gives us:

- a scatterplot on each pair of variables (numeric variables only)
- a distribution for each individual variable
- optionally use another variable to discriminate data by colour (hue)

For more, scroll through these three pages which showcase the main functionality.

<https://seaborn.pydata.org/introduction.html>

<https://seaborn.pydata.org/examples/index.html>

<https://jakevdp.github.io/PythonDataScienceHandbook/04.14-visualization-with-seaborn.html>

By the way, there is another module in data visualisation which goes much farther than we will here:

- CT5100 Data Visualisation
- CT5136 Data Visualisation - Online

code into your notebook and run it. Then try to achieve the same type of plot on any other dataset.

Computational Complexity

This notebook will start by reviewing the main idea of computational complexity, and then try to make it practical in the Python context. We will try to explain a lot of things that are usually left unsaid.

Computational efficiency is important!

Computer scientists realised long ago that the efficiency of an algorithm isn't really well-measured by measuring the "wall clock time" it takes to execute.

We can reduce runtime by buying a faster computer, faster disk, or a better network connection, or rewriting (the same algorithm) in a different language, but that doesn't change the inherent complexity of the algorithm.

Instead, what is important is how quickly the runtime increases as the size of the input increases (as input gets arbitrarily large).

A typical scenario: we run our algorithm with a small input file. It takes 10s to read in the data and 10s to process it. Then we run it on a larger input file. It takes 10m to read in the data, and 3 weeks to process it. The real issue here is the growth in processing from 10s to 3 weeks. We want to ignore the time spent reading the data.

Assumptions

- A program consists of idealised "instructions"
- There may be several types of instructions, e.g. multiplying two numbers, or checking for integer equality
- Some may be slower than others, but only by a **constant factor**;
- If one instruction is slower than another by a factor that depends on program input size, it is not an instruction -- we have to see it as composed of multiple instructions.

With these assumptions, we can write the runtime as the number of instructions -- a formula in n , the size of the input.

- To calculate the row-sum (sum of each row, as a vector) of a square matrix (n rows and n columns):

```
In [11]: def row_sum(M):
    R = []
    for L in M:
        R.append(sum(L))
    return R
```

We have to create an empty list, do n look-ups to get the rows and n^2 look-ups to get the individual items, and n^2 additions, and append n times to the list. That is a total of $n^2 + 3n + 1$ instructions.

Notice that `sum(L)` is not an instruction, because its runtime depends on the size of the `row_sum` input `M`. Instead, `sum` is composed of multiple instructions.

(But `sum([4, 5, 6])` does not depend on the size of `M`, so it counts as 1.)

Worst-case

In computational complexity, if the number of instructions could vary (for fixed input size), we usually consider the **worst-case**. This takes care of situations where the algorithm can exit early for a special case or an error: we don't want to count those when measuring complexity.

(There are situations where we'll consider average-case as well, see later.)

- E.g. to check that a given element exists in a list of length n :

```
In [45]: def element_exists(L, y):
    for x in L:
        if x == y: return True
    return False
```

We have to look at all n elements (worst-case). We'll have to look up n items and make n comparisons, so we'll execute $2n$ instructions.

Asymptotic runtime

"Asymptotic behaviour" means something like "behaviour for arbitrarily large values". In computational complexity, we only care about the number of instructions executed for large n .

How large?

"Arbitrarily large". For small n , the number is probably dominated by setup or reading input, and anyway for small n the program will be fast anyway, so we just don't care about runtime in the first place.

So the runtime is the number of instructions -- a formula in n . Now, we are going to argue that in that formula, for large n , only the **fastest-growing term** in n matters.

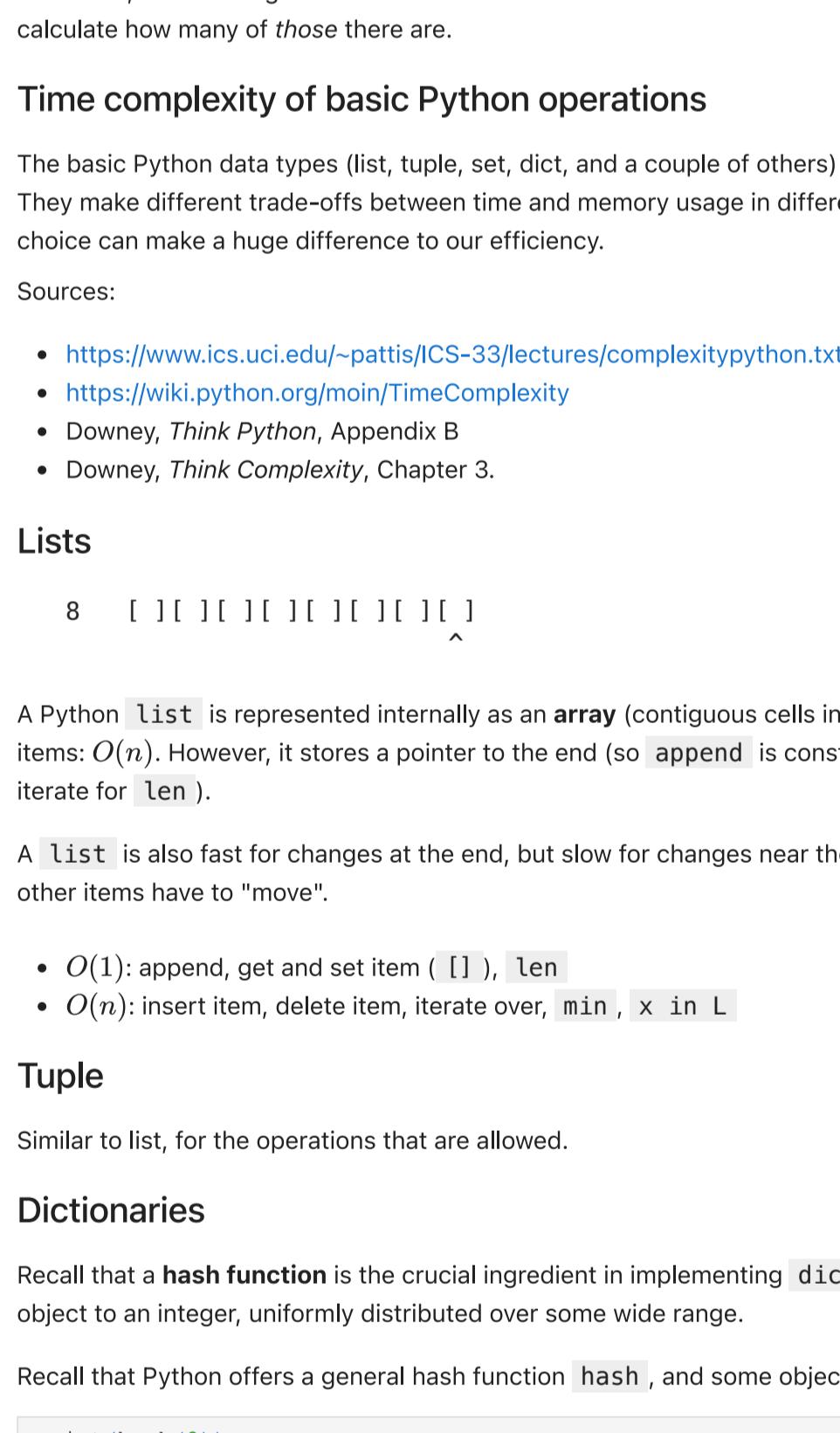
Let's look at the growth of a few simple terms.

```
In [7]: import math
import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [40]: plt.rcParams["figure.figsize"] = (8, 5)
plt.rcParams.update({'font.size': 14})
```

```
In [41]: ns = range(2, 21)
growths = [
    "constant": lambda n: 1,
    "logarithmic": lambda n: math.log(n),
    "linear": lambda n: n,
    "log-linear": lambda n: n * math.log(n),
    "quadratic": lambda n: n ** 2
]
```

```
In [42]: for growth in growths:
    plt.plot(ns, [growths[growth](n) for n in ns],
             label=growth)
plt.legend(); plt.xlabel("n");
plt.ylabel("n instructions");
```



What we see is a huge discrepancy between terms even for small n .

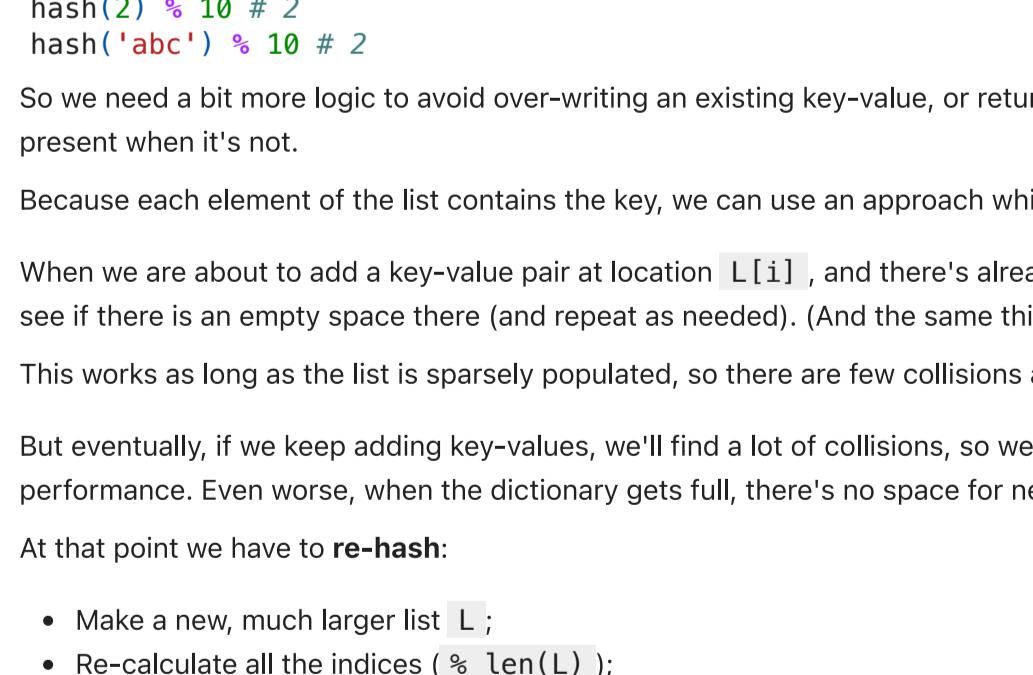
It doesn't matter if we execute 1000× more instructions in one algorithm than another. Consider `1000*linear`, below.

It also doesn't matter if one algorithm does a lot of extra setup that another doesn't. Consider `100000+linear`, below.

The `quadratic` function is still far worse than both even for this small $n = 2000$.

```
In [43]: ns = range(2, 2001)
growths = [
    "linear": lambda n: n,
    "100000+linear": lambda n: n + 100000,
    "1000*linear": lambda n: n * 1000,
    "quadratic": lambda n: n ** 2
]
```

```
In [44]: for growth in growths:
    plt.plot(ns, [growths[growth](n) for n in ns],
             label=growth)
plt.legend(); plt.xlabel("n");
plt.ylabel("n instructions");
```



It also doesn't matter if one type of instruction is (say) 1000× slower than another: again, consider `1000*linear` above. This justifies our (surprising) assumption that we can treat all instructions as equal.

And, e.g., `sum(range(1000))` counts as one instruction.

Therefore, for large n , the **number of instructions** of the **fastest-growing term** in n is effectively **proportional** to runtime.

"Big O" notation

We ignore all slow-growing terms. The "Big O" notation ("the order of complexity") does just this:

- Write the total number of instructions as a function of n , e.g. $10n^2 + 23n \log(n) + 1000000$
- Drop all but the fastest-growing term and drop its coefficient, and write in "big O" notation: $O(n^2)$.

Exercise: consider again the `element_exists(L, x)` function. We saw that its worst-case complexity is $O(n)$. What is its average-case complexity?

Answer: the average-case complexity is still $O(n)$. On average we will have to compare $n/2$ elements, so we'll do $n/2$ lookups and $n/2$ comparisons, giving $O(n)$.

There are a few common "Big O" values. Here they are from slowest- to fastest-growing.

- constant, $O(1)$
- logarithmic, $O(\log(n))$
- linear, $O(n)$
- log-linear aka quasi-linear, $O(n \log(n))$
- quadratic, $O(n^2)$

Examples

- We already argued that to check a particular item exists in a list of length n , we need $2n$ instructions. That is linear, $O(n)$.
- Calculating the row-sum a square matrix of size n is quadratic, $O(n^2)$.
- Sorting a list of length n takes $O(n \log(n))$.

This is a classic example. It doesn't matter what algorithm you use, it can't be better than $O(n \log(n))$. (There are algorithms which are much worse, e.g. **bubble sort**) Thus, it is justified to talk about the complexity of the **problem**, not just the complexity of the **algorithm**.

Rules of thumb

- Every for-loop multiplies by a factor of n , e.g. this is $O(n^3)$:

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            # do something
```

Reading input data can usually be ignored. We usually just assume that reading an input file is $O(n)$, i.e. linear in the number of lines or file size, and so we ignore it. We know that processing won't be sub-linear, so the time spent reading data won't exceed it, so it can be ignored.

People say things like "comparison-based sorting algorithms like Merge sort take only Big O $O(n \log(n))$ and so you can sort just about any amount of data in a feasible time" (<https://www.gwern.net/Complexity-vs-AI>). Why does $O(n \log(n))$ imply that it is feasible?

It's because $O(n \log(n))$ is only a little slower than $O(n)$ itself. We already assume that we can read the data in feasible time, otherwise the question of sorting it would not arise. Reading it is $O(n)$. So if we can read it into RAM, we can sort it. If our RAM is not big enough, the sorting algorithm will be bottlenecked by **thrashing**.

Sometimes we might have multiple inputs, e.g. two arrays of sizes n and m , or a 2D array of size $n \times m$. In these cases we might need $O()$ expressions in both n and m . Alternatively we might just (conservatively) take n to be the larger of n and m .

The most important difference in practice is between linear and log-linear algorithms and quadratic.

In a linear algorithm, "you took some input and did something simple to each element" (Accidentally Quadratic, <http://accidentallyquadratic.tumblr.com/post/113840433022/why-accidentally-quadratic>).

A quadratic algorithm can occur if we need to traverse the whole input once for every element of the input.

$O(n \log n)$ can occur if we take something like that and improve it with a tree structure, because the cost of traversing a tree from root to leaf is often $O(\log n)$.

Another place $O(n \log n)$ occurs is in sorting algorithms, and one "reason" why it occurs there is that some algorithms work recursively, and so induce an implicit binary tree structure of function calls.

Exponential and factorial algorithms seem to occur mostly as algorithms that our common sense would probably tell us not to write in the first place.

Situations not governed by computational complexity

Suppose we are running a webserver. Every request that comes in is approximately the same, small, size. We don't care about asymptotic complexity. We **do** care about the setup time and the coefficients!

We only care about computational complexity if we're going to have large inputs. Someone once criticised me for using an $O(n^2)$ algorithm instead of $O(n)$ in a situation where we knew $n < 7$. You might say "well $6 < 6^2$, so it will be faster".

But remember that for small n the assumptions break down: we can't ignore the coefficient or the "slow-growing" terms. E.g. for $n < 7$, $n^2 < 20 + 3n$.

Complexity in Python data structures

We have been discussing idealised instructions. How do they correspond to Python source code?

Well, it turns out that many of the operations of the Python compound datatypes (`list`, `set`, `dict`, `tuple`) take time which depends on the size of the object (e.g. `list.find` takes time proportional to the length of the `list`).

Therefore, we have to see them as composed of multiple instructions and calculate how many of those there are.

Time complexity of basic Python operations

The basic Python data types (`list`, `tuple`, `set`, `dict`, and a couple of others) offer various operations, with various time complexities.

They make different trade-offs between time and memory usage in different common cases and worst cases. Sometimes the right choice can make a huge difference to our efficiency.

Exercise: consider again the `element_exists(L, x)` function. We saw that its worst-case complexity is $O(n)$. What is its average-case complexity?

Answer: the average-case complexity is still $O(n)$. On average we will have to compare $n/2$ elements, so we'll do $n/2$ lookups and $n/2$ comparisons, giving $O(n)$.

There are a few common "Big O" values. Here they are from slowest- to fastest-growing.

- constant, $O(1)$
- logarithmic, $O(\log(n))$
- linear, $O(n)$
- log-linear aka quasi-linear, $O(n \log(n))$
- quadratic, $O(n^2)$

Lists

```
8 [ ] [ ] [ ] [ ] [ ] [ ] ^
```

A Python `list` is represented internally as an **array** (contiguous cells in memory). Many operations involve iterating over the items or the length, and so we ignore it. We know that processing won't be sub-linear, so the time spent reading data won't exceed it, so it can be ignored.

People say things like "comparison-based sorting algorithms like Merge sort take only Big O $O(n \log(n))$ and so you can sort just about any amount of data in a feasible time" (<https://www.gwern.net/Complexity-vs-AI>). Why does $O(n \log(n))$ imply that it is feasible?

It's because $O(n \log(n))$ is only a little slower than $O(n)$ itself. We already assume that we can read the data in feasible time, otherwise the question of sorting it would not arise. Reading it is $O(n)$. So if we can read it into RAM, we can sort it. If our RAM is not big enough, the sorting algorithm will be bottlenecked by **thrashing**.

Sometimes we might have multiple inputs, e.g. two arrays of sizes n and m , or a 2D array of size $n \times m$. In these cases we might need $O()$ expressions in both n and m . Alternatively we might just (conservatively) take n to be the larger of n and m .

The most important difference in practice is between linear and log-linear algorithms and quadratic. This is called **linear probing**:

When we are about to add a key-value pair at location `L[i]`, and there's already a different key there, just go to `L[i+1]` and see if there is an empty space there (and repeat as needed). (And the same thing when retrieving.)

This works as long as the list is sparsely populated, so there are few collisions and most linear probes end quickly.

But eventually, if we keep adding key-values, we'll find a lot of collisions, so we'll spend forever linear probing, and get bad performance. Even worse, when the dictionary gets full, there's no space for new elements!

At that point we have to re-hash:

- Make a new, much larger list `L`;
- Re-calculate all the indices (% `len(L)`);
- Copy each old element to its new location in the new list.

That will take a long time, so our **worst-case time** for item addition is $O(n)$. But it happens **rarely**.

Dictionary performance

We ignore all slow-growing terms. The "Big O" notation ("the order of complexity") does just this:

- Write the total number of instructions as a function of n , e.g. $10n^2 + 23n \log(n) + 1000000$
- Drop all but the fastest-growing term and drop its coefficient, and write in "big O" notation: $O(n^2)$.

Exercise: consider again the `element_exists(L, x)` function. We saw that its worst-case complexity is $O(n)$. What is its average-case complexity?

Answer: the average-case complexity is still $O(n)$. On average we will have to compare $n/2$ elements, so we'll do $n/2$ lookups and $n/2$ comparisons, giving $O(n)$.

For many situations, this trade-off is well worth it.

High-Performance Computing

The purpose of this notebook is to provide the basic terminology and concepts of high-end/high-performance computing. If you already know what these are, you can skip it!

- Threads versus processes
- Distributed versus parallel computing
- GPUs
- Taskfarming
- Profilers

Python and HPC

Python is slow, but it's still used for (some) HPC! Why?

- Numpy can move many inner loops to C or Fortran
- "Just-in-time" compilation with Numba, Cython, PyPy, etc.
- Deep learning uses GPUs, Python uses GPU libraries
- Python is slower than C only by a (multiplicative) constant. Getting the right algorithm remains more important.

GPUs

A **graphics processing unit** is a specialised processor for graphics, especially video games. A GPU is specialised for processing large numerical matrices. Many AI and numerical computing problems run very effectively on GPUs!

- Deep learning is always on GPUs, except for toy problems
- Cryptocurrency :-)
- Weather simulations, nuclear simulations, and more.

Distributed versus parallel computing

Parallel computing: running multiple things at once on a single machine (especially with multiple CPUs on a single machine)

Distributed computing: running one thing across multiple machines, usually with message-passing between them.

Parallel computing

<https://www.explainthatstuff.com/how-supercomputers-work.html>

Parallel computing with threads and processes

Threads and processes are two similar concepts. They both involve multiple parts of a program running in parallel (on a single machine).

- Every program, when it runs, becomes a **process**
- A program may **fork** to become multiple processes
- A program may also create multiple **threads**

Threads are more "lightweight" (easier/quicker to start and stop threads than processes).

If we have multiple CPUs, then multiple processes or multiple threads can take advantage of that.

Even if we have a single CPU, multiple processes or threads can be useful. (They don't really run in **parallel**, but the OS gives that illusion by interleaving their work, and it can still help a lot!)

Examples

- Any program with a GUI will have at least a back-end process and a GUI process or thread. Even when the back-end process is busy processing, the GUI remains responsive.
- A web server typically launches a new thread for every incoming request.
- Some optimisation algorithms can run multiple processes or threads, e.g. all running in parallel, to take advantage of multiple CPUs, e.g. `GridSearchCV` has a parameter `n_jobs`.

Communication between threads and/or processes is a complex topic and causes many bugs :-/

Using multiple cores -- the hard way

Take our program, and rewrite it to use multiple processes and/or threads with communication between them. When it runs, the OS will put each process/thread on separate cores in a fairly smart/adaptive way. We'll get a speedup, but probably sublinear :-)

Using multiple cores -- the easy way

If we need to run our program **multiple times** and we have **multiple cores** (and if each run won't use up too much RAM), then don't rewrite the program. Just run it multiple times at once. E.g. if we have 4 cores, open up 4 terminals and type `python myprog.py` in each one. We'll probably get a 4x speedup :-)

This is a bit too **manual** for large-scale tasks (e.g. many runs on multiple machines). **Taskfarming** takes this to a logical conclusion: we put our commands into a file and hand it to a specialised taskfarming program. It decides how many to run in parallel, and takes care of starting the next one whenever a task finishes (to ensure CPU utilisation) etc.

It can even use multiple machines if available.

In the next notebook/video we'll see how this is used on ICHEC.

Distributed computing

In distributed computing, we have multiple machines running a single job, so usually they need to communicate during the job. This is outside our scope.

Understanding performance in terms of bottlenecks

When a program is too slow, there are many possible reasons:

- Slow CPU
- Not enough RAM causes the machine to move data from RAM to disk and back often ("thrashing")
- Wrong algorithm (e.g. "accidentally quadratic")

And many possible solutions:

- Buy a bigger computer or a GPU
- Use parallel and/or distributed computing
- Find a better algorithm.

<https://www.kissclipart.com/manufacturing-bottleneck-clipart-bottleneck-business-0xtdag/>

A **bottleneck** model is useful for thinking about performance. Usually when a program is slow, there's one main reason why it's slow, e.g. just one function or one inner loop. That's the only place where an optimisation effort can yield any improvements.

Before trying to optimise a function, we should find out **which functions** are using most of the time in our program!

That's the job of the **profiler**. A profiler is a tool that runs your program and tells you where it spends most of its time. It gives a report like this, telling us how often each function is called and the total time spent in each function.

197 function calls (192 primitive calls)

in 0.002 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall
--------	---------	---------	---------	---------

filename:lineno(function)				
---------------------------	--	--	--	--

1	0.000	0.000	0.001	0.001
---	-------	-------	-------	-------

<string>:1(<module>)				
----------------------	--	--	--	--

1	0.000	0.000	0.001	0.001
---	-------	-------	-------	-------

re.py:212(compile)				
--------------------	--	--	--	--

1	0.000	0.000	0.001	0.001
---	-------	-------	-------	-------

re.py:268(_compile)				
---------------------	--	--	--	--

1	0.000	0.000	0.000	0.000
---	-------	-------	-------	-------

sre_compile.py:172(_compile_charset)				
--------------------------------------	--	--	--	--

1	0.000	0.000	0.000	0.000
---	-------	-------	-------	-------

sre_compile.py:201(_optimize_charset)				
---------------------------------------	--	--	--	--

4	0.000	0.000	0.000	0.000
---	-------	-------	-------	-------

sre_compile.py:25(_identityfunction)				
--------------------------------------	--	--	--	--

3/1	0.000	0.000	0.000	0.000
-----	-------	-------	-------	-------

sre_compile.py:33(_compile)				
-----------------------------	--	--	--	--

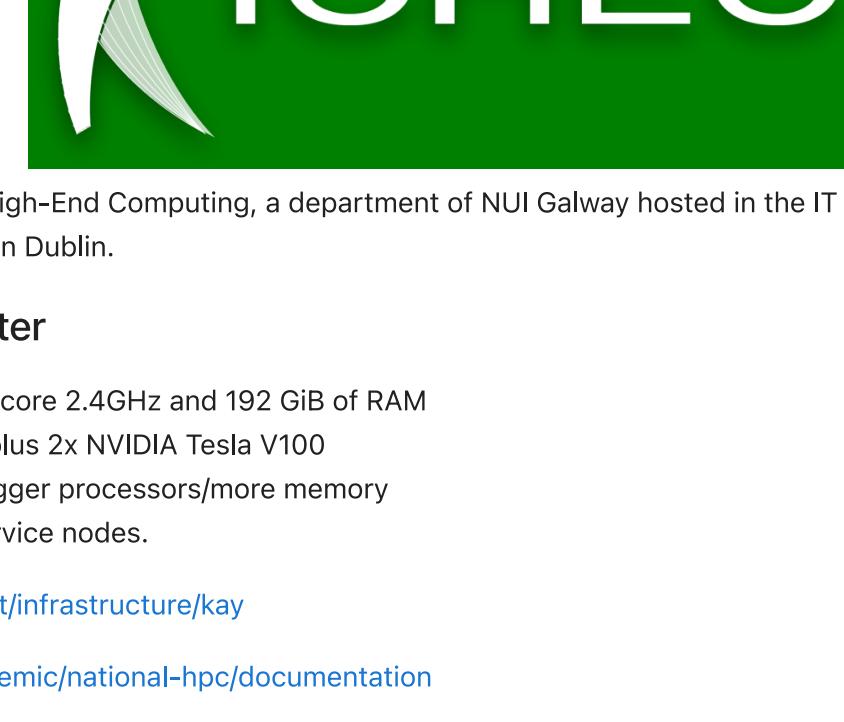
Python has a built-in profiler: <https://docs.python.org/3/library/profile.html>

Sophisticated understanding of algorithms and engineering and budget trade-offs might be needed! You could pay a developer to make your algorithm use less RAM, but it might be cheaper to just buy a load of extra RAM. You could implement a Hadoop solution, but it might be better to just learn to use basic tools properly.

We have barely scratched the surface here, but you can see more in **Tools and Techniques for Large-Scale Data Analytics** in Semester 2.

For now, we'll assume that our program is reasonably well-optimised, and we want to run it multiple times with different hyperparameters, so a possible solution is to use **taskfarming** on a large compute server with multiple CPUs (maybe even

ICHEC



ICHEC is the Irish Centre for High-End Computing, a department of NUI Galway hosted in the IT Building. The server farm is in Waterford and some staff are in Dublin.

The Kay supercomputer

- 336 nodes each of 2x 20-core 2.4GHz and 192 GiB of RAM
 - 16 GPU nodes, as above plus 2x NVIDIA Tesla V100
 - Some more nodes with bigger processors/more memory
 - Login nodes and other service nodes.
- <https://www.ichec.ie/about/infrastructure/kay>
- <https://www.ichec.ie/academic/national-hpc/documentation>

Condominium access versus Class-C access

- Each Irish institution has "condominium" access to ICHEC, including MSc students.
<https://www.ichec.ie/academic/condoniu>
- Alternatively, researchers can apply for "Class-C projects", and then add other researchers to the project to give them access.

Condominium access

The MScAI (full-time) will use this. To get access:

- Register on <https://register.ichec.ie/register>
- Fill in the Word form (in Blackboard) and send to Noreen Goggin noreen.goggin@nuigalway.ie

See [ichec.pdf](#) for full details.

Class-C access

The MScAI (part-time) will use this. To get access:

- Register on <https://register.ichec.ie/register>
- Send you ICHEC username, student number, and programme name to the Programme Director.
- Go to <https://register.ichec.ie/login> and login, and find the project `ngcom018c`, click `Apply` and `Confirm`.

See [ichec.pdf](#) for full details.

ssh

`ssh` is a command for logging-in to a remote node. When you have an ICHEC account, you can run this on your laptop:

`$ ssh my_username@kay.ichec.ie`

You'll be asked for your password and you'll end up at a Linux prompt on a `kay` login node. The login node is for managing your files and submitting batch jobs. Do not directly `run` any large jobs there!

Access is from within the NUI Galway network only!

Later on, for convenience, you can set up access from home/elsewhere also by generating an SSH keypair:

<https://www.ichec.ie/academic/national-hpc/documentation/ssh-keys>

Python on ICHEC

Lots of people are using Python for jobs on ICHEC.

<https://www.ichec.ie/academic/national-hpc-service/software/python-conda>

Virtual environments

It's recommend to use a **virtual environment** to manage your Python/Anaconda installation. A `venv` is just a collection of software installed with Anaconda.

You can create multiple virtual environments and activate the one you want. One advantage is you might have one project that uses Tensorflow 1 and another that uses Tensorflow 2, and you want to be able to run both projects at different times. You can create a `venv` for each project with the right packages installed and just switch between them.

Setting up your environment

You can do this on a login node:

```
# tell kay we will use Anaconda
module load conda/2
# create a new venv
conda create --name my_env
# activate it
conda activate my_env
# install the stuff we need
conda install tensorflow-gpu networkx scikit-learn
# for completeness: exit the virtual environment
conda deactivate
```

A `venv` is just a collection of software installed with Anaconda. It doesn't include the contents of your home directory. So, the next step is to make and/or copy your work directories to `kay`, e.g. on a login node:

`mkdir work`

You'll need to be able to navigate a Unix filesystem, e.g.:

- `mkdir` # make directory
- `mv` # move/rename
- `cp` # copy
- `cd` # change directory
- `ls` # list directory

SCP

`scp` is a command for copying files to a remote computer. When you have an ICHEC account, you can run this on your laptop (notice `-r` is for "recursive copy"):

`$ scp -r my_project_directory my_username@kay.ichec.ie:/ichec/home/users/my_username/work/`

NFS

The contents of your home directory on the login node (which we started to create above) are synced back and forth to any compute node on `kay`. Your jobs might run on via **NFS**, the network filesystem.

Interactive testing with `srun`

Then you can request a node for eg 30m interactive testing. This is just for testing/prototyping, not for large runs.

Run this from the login node:

`srun -p DevQ -N 1 -A nuig02 -t 0:30:00`

If one is available, you will be delivered to a compute node on DevQ. Then:

```
module load conda/2
conda info --envs
source activate my_env
cd work/my_project_directory
python my_prog.py
```

``

The first line is a standard way of telling the operating system that it is a shell script to be run by `/bin/sh`, a standard shell.

`sbatch` is the Unix command which submits a job to a queue. We run it like this on a login node:

```
cd work/my_project_directory
sbatch my_batch_job.sh
```

Taskfarming

The above procedure only makes sense if `my_prog.py` is a very large job that requires a whole node to itself!

An easier way to use supercomputer facilities is to run many runs in parallel, one per core. Suppose our program `my_prog.py` has a hyperparameter `n` which takes on values 0-9, and for each we want to run 100 repetitions with 100 random seeds. We write our program to accept arguments `my_prog.py <n> <seed>`.

Then we can just put these commands into a file named e.g. `tasks.sh`:

```
# each line is one command
python my_prog.py 0 0
python my_prog.py 0 1
python my_prog.py 0 2
...
python my_prog.py 9 99
```

Then, instead of `python my_prog.py` at the bottom of `my_batch_job.sh`, we'll put a `taskfarm` command:

```
taskfarm tasks.sh
```

It will take each line of `tasks.sh` as a task. It will start tasks, usually one per core, and when they finish it will start new ones, until the whole file is finished.

After submitting

- You can check on your queued jobs: `squeue -u my_username`
- You should receive an email when the job starts.
 - It might contain error messages. To debug, check your account name and check that your tasks run ok interactively.
- When your job finishes, you should receive another email.

GPU on `kay`

Ensure that your `venv` includes (e.g.) `tensorflow-gpu` (it should as we installed it earlier). Then request a GPU node for interactive testing:

`srun -p GpuQ -N 1 -A nuig02 -t 0:30:00`

If one is available, you will be delivered to a compute node with GPU. Then run these:

```
module load cuda/10.0
module load conda/2
conda info --envs # what envs exist?
```

```
source activate my_env
# test that GPU support is working - you should see lots of messages
# including "name: Tesla V100-PCIE-16GB"
```

```
python -c 'import tensorflow as tf; tf.test.gpu_device_name()'
```

Then make sure to include the line `module load cuda/10.0` in your `my_batch_job.sh` file.

On the [Data Science module page](#) (Semester 1 for full-time students, in Year 2 Semester 2 for part-time), you'll have opportunities

to use deep neural network libraries such as Tensorflow or PyTorch on GPU.

Scikit-Learn: Introduction



Scikit-Learn provides a wide variety of machine learning algorithms and utilities with a fairly uniform interface.

A useful resource is the Vanderplas *Python Data Science Handbook* <https://jakevdp.github.io/PythonDataScienceHandbook/> and we will draw on this, referring to it as *PDSH*.

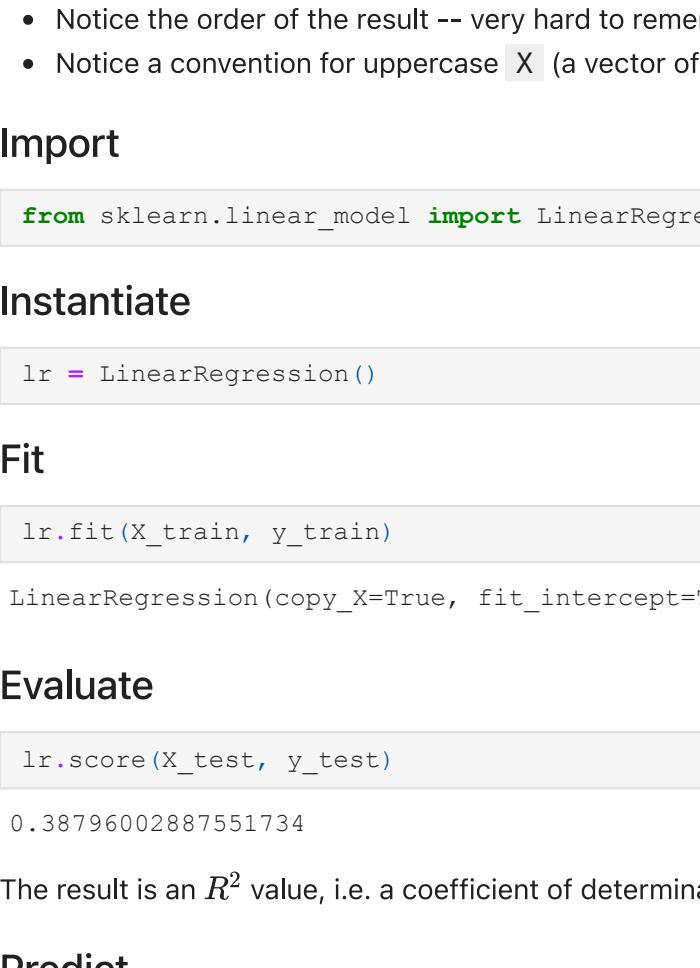
We'll start with **supervised learning**.

Vanderplas (PDSH) describes the core workflow for supervised learning in Scikit-Learn as: "import/instantiate/fit/predict"

I would add another step:

- import
- instantiate
- fit
- **evaluate**
- predict

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
d = np.loadtxt("data/mpg_extract.csv", delimiter=",",
               skiprows=1)
d[:, 0] # year, mpg
plt.scatter(d[:, 0], d[:, 1]);
```



- Independent variables: array of shape `(n_samples, n_variables)`
- (even if `n_variables == 1`)
- Our dependent variable is typically of shape `(n_samples,)`.

```
In [3]: X = d[:, 0:1]
y = d[:, 1]
X.shape, y.shape
```

```
Out[3]: ((398, 1), (398,))
```

Make a train-test split

```
In [4]: from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

- Notice the order of the result -- very hard to remember!
- Notice a convention for uppercase `X` (a vector of variables) and lowercase `y` (a single variable)

Import

```
In [5]: from sklearn.linear_model import LinearRegression
```

Instantiate

```
In [6]: lr = LinearRegression()
```

Fit

```
In [7]: lr.fit(X_train, y_train)
Out[7]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Evaluate

```
In [8]: lr.score(X_test, y_test)
```

```
Out[8]: 0.38796002887551734
```

The result is an R^2 value, i.e. a coefficient of determination. Higher is better.

Predict

For an unknown `X`:

```
In [9]: X_query = np.array([[83.1]])
fX_query = lr.predict(X_query)
fX_query
```

```
Out[9]: array([31.30689899])
```

Notice the result is of shape `(1,)`.

Understanding the model

Look at the fitted parameters a and b_i (for $i \in (0, n_vars-1)$).

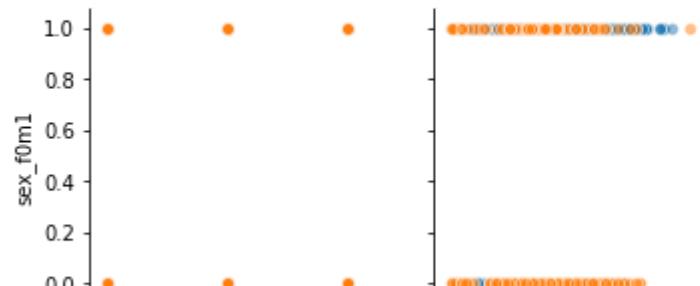
```
In [10]: a = lr.intercept_
B = lr.coef_
a, B
```

```
Out[10]: (-63.18180027796002, array([1.13841806]))
```

Visualise

We'll look at our test data, newly-fitted model, and the query point.

```
In [11]: X_grid = np.linspace(70, 83, 14)
fX_grid = a + B * X_grid
plt.scatter(X_test, y_test)
plt.plot(X_grid, fX_grid)
plt.scatter(X_query, fX_query, s=100, c="red");
```



Observe that this is *extrapolation*, not *interpolation*, i.e. we are trying to predict for a point which is "outside" the observed `X` (not just unseen).

Summary

- get data
- train-test split
- import
- instantiate
- fit on training data
- look at model parameters
- evaluate on test data
- visualise
- **query on new data**

Pandas instead of Numpy

We can pass in a Pandas `DataFrame` as `X`

```
In [12]: import pandas as pd
df = pd.read_csv("data/mpg_extract.csv",
                  names=["year", "mpg"], skiprows=1)
X = df[["year"]]
y = df[["mpg"]]
X_train, X_test, y_train, y_test = train_test_split(X, y)
lr.fit(X_train, y_train)
lr.predict([[85]]) # result is a Numpy array, not Pandas
```

```
Out[12]: array([33.97513394])
```

Exercises

1. Replace `LinearRegression` here with `DecisionTreeRegressor`. Is R^2 better?
2. Read in the dataset `data/titanic.csv`. Set `X` to be the first three columns, and `y` to be the `survived` column. The goal is to predict `survived`. Use `sklearn.linear_model.LogisticRegression`, and follow the steps above for a train-test split and fit.

Exercises

1. Evaluate on the test set using `score`. What does the score value mean in classification? Which is better, high or low? Look it up if needed.
2. Predict whether the following (fictional) passenger would survive: Jack, age 20, travelling in 3rd class. Use `predict_proba` to predict also the *probability* of surviving.

Solutions

```
In [149...]: from sklearn.tree import DecisionTreeRegressor
dt = DecisionTreeRegressor()
dt.fit(X_train, y_train)
dt.score(X_test, y_test) # R^2
```

```
Out[149...]: 0.3610267988691396
```

```
In [127...]: df = pd.read_csv("data/titanic.csv")
df.head()
```

```
Out[127...]: pclass  sex_f0m1  age  survived
0            3        1  22.0      0
1            1        0  38.0      1
2            3        0  26.0      1
3            1        0  35.0      1
4            3        1  35.0      0
```

```
In [143...]: import seaborn as sns
# the pairplot is not perfect for discrete vars
# but still useful. several extra args to improve appearance
sns.pairplot(df, vars=["pclass", "age", "sex_f0m1"],
              diag_kind="hist",
              hue="survived",
              plot_kws={"alpha": 0.5},
              diag_kws={"alpha": 0.5}
              );
```



```
In [134...]: X = df[["pclass", "age", "sex_f0m1"]]
y = df[["survived"]]
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
In [136...]: from sklearn.linear_model import LogisticRegression
logr = LogisticRegression()
```

```
In [137...]: logr.fit(X_train, y_train, solver="lbfgs")
```

```
/home/jmmcd/anaconda3/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
```

```
Out[137...]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                                intercept_scaling=1, l1_ratio=None, max_iter=100,
                                multi_class='warn', n_jobs=None, penalty='l2',
                                random_state=None, solver='warn', tol=0.0001, verbose=0,
                                warm_start=False)
```

```
In [138...]: logr.score(X_test, y_test)
```

```
Out[138...]: 0.8100558659217877
```

`help(logr.score)` -> percentage accuracy, so higher is better, so 81% is pretty good.

```
In [145...]: jack = np.array([[3, 20, 1]])
logr.predict(jack)
```

```
Out[145...]: array([0])
```

```
In [146...]: logr.predict_proba(jack)
```

```
Out[146...]: array([[0.85169886, 0.14830114]])
```

This means there is a 14.8% probability of surviving :-)

Scikit-Learn: Analysing Errors

After running the import/instantiate/fit/evaluate steps, we might see that our performance is not good enough on test data. Before deploying our model, we should diagnose the problem and see if we can fix it. In this notebook, we'll see a helpful tool for diagnosis.

The code and text are derived from <https://stackoverflow.com/questions/28256058/plotting-decision-boundary-of-logistic-regression>. I've mostly changed to use `meshgrid`.

```
In [8]: import
```

```
In [10]: sklearn.__version__
```

```
Out[10]: '0.23.2'
```

```
In [1]: import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="white")
```

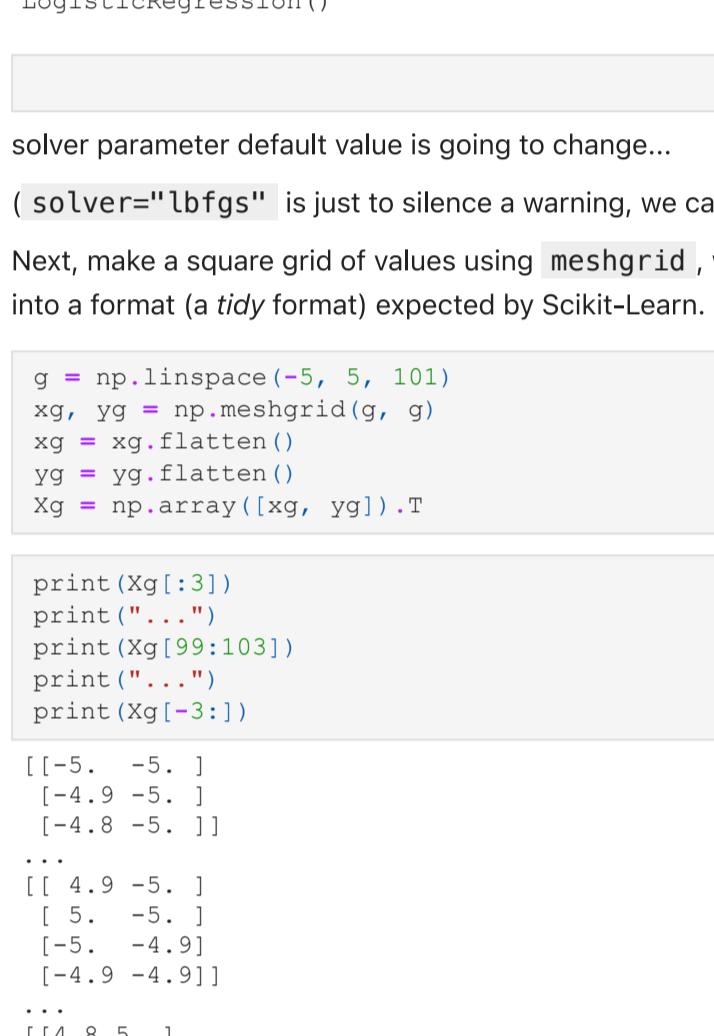
First, generate some fake data:

```
In [2]: X, y = make_classification(200, 2, 2, 0, weights=[.5, .5],
                               random_state=15)
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

/Users/jmmcd/anaconda3/lib/python3.7/site-packages/sklearn/utils/validation.py:70: FutureWarning: Pass `n_inf` or `max_nfev=2`, `n_redundant=0` as keyword args. From version 0.25 passing these as positional arguments will result in an error
FutureWarning)

Now we take a look:

```
In [3]: f, ax = plt.subplots(figsize=(6, 6))
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
           cmap="RdBu", vmin=-.2, vmax=1.2,
           edgecolor="white", linewidth=1);
```



Fit the classifier to the training set:

```
In [7]: clf = LogisticRegression()
clf.fit(X_train, y_train)
```

```
Out[7]: LogisticRegression()
```

```
In [ ]:
```

solver parameter default value is going to change...

(`solver="lbfgs"` is just to silence a warning, we can ignore it.)

Next, make a square grid of values using `meshgrid`, which we saw when studying fractals. We have to manipulate a bit to get it into a format (a *tidy* format) expected by Scikit-Learn.

```
In [67]: g = np.linspace(-5, 5, 101)
xg, yg = np.meshgrid(g, g)
xg = xg.flatten()
yg = yg.flatten()
Xg = np.array([xg, yg]).T
```

```
In [68]: print(Xg[:3])
print("...")
print(Xg[99:103])
print("...")
print(Xg[-3:])

[[[-5. -5.]
 [-4.9 -5.]
 [-4.8 -5.]]]
 ...
 [[[ 4.9 -5.]
 [ 5. -5.]
 [-5. -4.9]
 [-4.9 -4.9]]]
 ...
 [[[4.8 5.]
 [4.9 5.]
 [5. 5.]]]
```

Next we evaluate the probability of each point in the grid.

```
In [69]: probs = clf.predict_proba(Xg)
probs[:5]
```

```
Out[69]: array([[9.99816941e-01, 1.83059102e-04],
 [9.99829506e-01, 1.70494015e-04],
 [9.99841209e-01, 1.58791252e-04],
 [9.99852108e-01, 1.47891651e-04],
 [9.99862260e-01, 1.37740108e-04]])
```

But our model returns a vector of probabilities for each X : ($P(y=0)$, $P(y=1)$). This is redundant and we can only visualise one colour per pixel, so we take $P(y=1)$ only.

```
In [70]: probs = probs[:, 1]
probs[:5]
```

```
Out[70]: array([0.00018306, 0.00017049, 0.00015879, 0.00014789, 0.00013774])
```

Now we reshape back to a square grid.

```
In [71]: probs = probs.reshape(101, 101)
```

Now, plot the probability grid as a contour map and additionally show the test set samples on top of it. There is quite a bit going on here. No need to remember details. Unfortunately it has to go all in one cell, and doesn't fit on a single slide. So, if you want to read the code, please just see the `.ipynb` file.

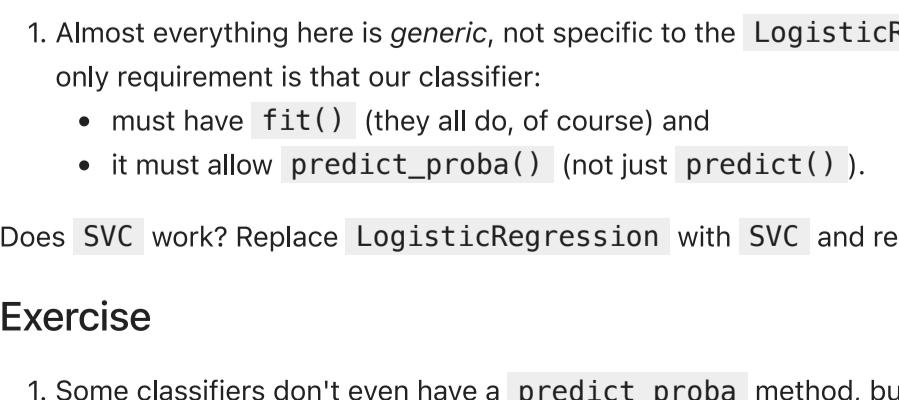
```
In [74]: f, ax = plt.subplots(figsize=(8, 6))

# contourf -> contour plot
# RdBu -> Red to Blue colour map
# vmin, vmax -> bounds on probs
contour = ax.contourf(g, g, probs, 25, cmap="RdBu", vmin=0, vmax=1)

# colorbar -> legend
ax_c = f.colorbar(contour)
ax_c.set_label("$P(y=1)$")
ax_c.set_ticks([0, .25, .5, .75, 1])

# s -> marker size
ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, s=50,
           cmap="RdBu", vmin=-.2, vmax=1.2,
           edgecolor="white", linewidth=1)

ax.set(aspect="equal",
       xlim=(-5, 5), ylim=(-5, 5),
       xlabel="$X_0$", ylabel="$X_1$");
```



This is the space of independent variables, with each point in the grid coloured by its value of $P(y=1)$. If we take the decision boundary at $P(y=1) = 1/2$ (or any other threshold) then we have a linear decision boundary as expected.

We also see the test points, and we can clearly see a few individual errors, i.e. blue points in the red area and vice versa. By creating a plot like this, we can look at individual errors and start to investigate them.

Further reading

Another useful tool is the *confusion matrix*. See <https://jakevdp.github.io/PythonDataScienceHandbook/05.02-introducing-scikit-learn.html#Classification-on-digits> for an example.

Further ideas

Depending on our problem, it might be interesting to plot other items, such as a sample of the training data, with a different marker shape from the test data.

Exercise

- Almost everything here is *generic*, not specific to the `LogisticRegression` classifier. By the principle of duck typing, the only requirement is that our classifier:
 - must have `fit()` (they all do, of course) and
 - it must allow `predict_proba()` (not just `predict()`).

Does `SVC` work? Replace `LogisticRegression` with `SVC` and re-run.

Exercise

- Some classifiers don't even have a `predict_proba` method, but in `SVC` it just refuses to work -- **unless** we call the constructor with `SVC(probability=True)`. So, do that, and re-run, and take a look at the new decision boundary. What is it like?

Solutions

- As you know if you saw **Exercise 2**, the answer is no. You'll see an error about "probability is False".

2. Now the decision boundary is *non-linear*, and seems to skirt nicely around the data.

Scikit-Learn: Unsupervised Learning

Recall the workflow:

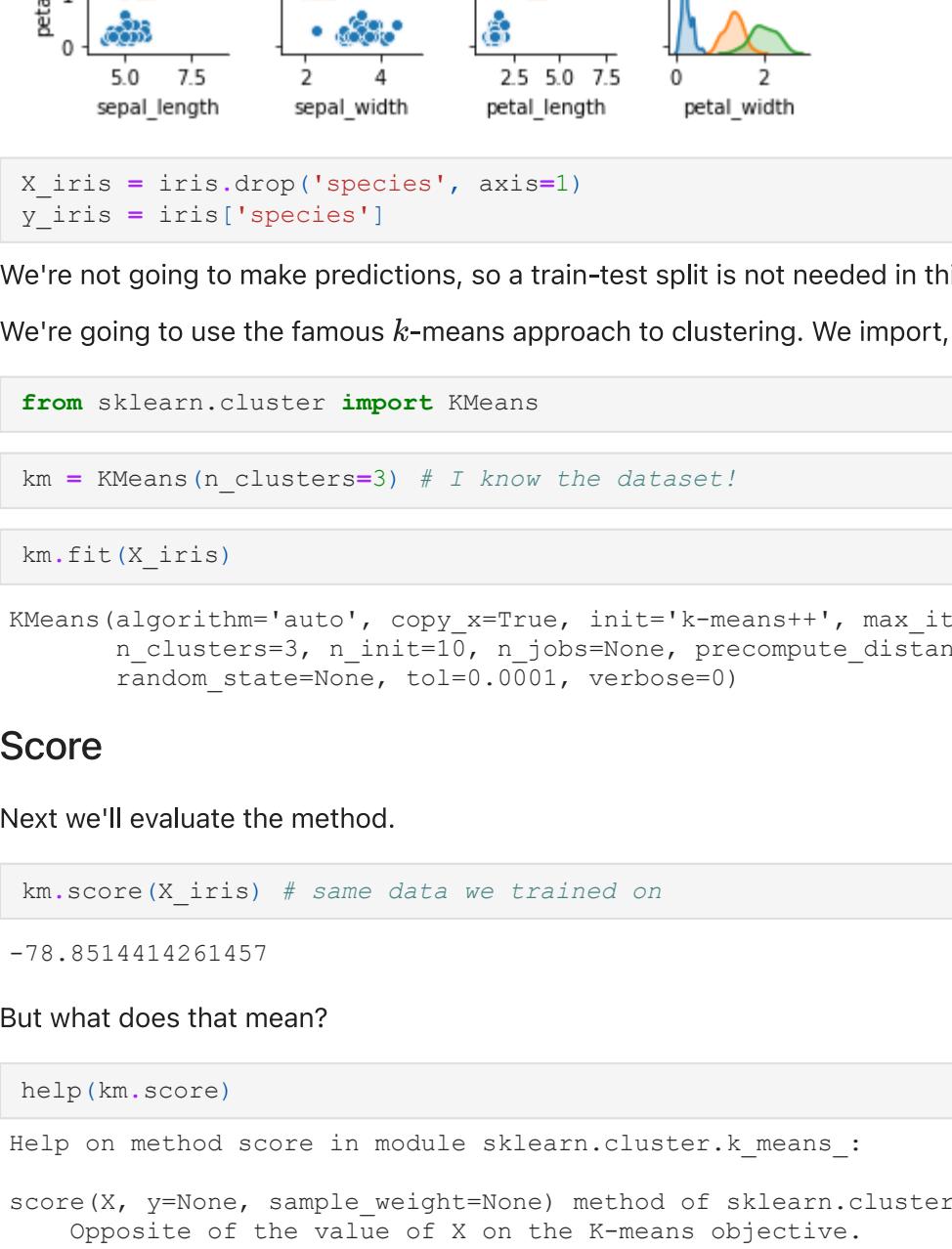
- import
- instantiate
- fit
- evaluate
- predict

For unsupervised learning, such as clustering, the process is exactly the same. Let's consider the well-known Iris dataset, which is built-in to Seaborn. We'll start by importing and taking a quick look at the dataset.

```
In [1]: import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
iris = sns.load_dataset('iris')
iris.head()
```

```
Out[1]:   sepal_length  sepal_width  petal_length  petal_width  species
0          5.1         3.5          1.4         0.2    setosa
1          4.9         3.0          1.4         0.2    setosa
2          4.7         3.2          1.3         0.2    setosa
3          4.6         3.1          1.5         0.2    setosa
4          5.0         3.6          1.4         0.2    setosa
```

```
In [2]: sns.pairplot(iris, hue='species', height=1.5);
```



```
In [3]: X_iris = iris.drop('species', axis=1)
y_iris = iris['species']
```

We're not going to make predictions, so a train-test split is not needed in this example.

We're going to use the famous k -means approach to clustering. We import, instantiate and fit:

```
In [4]: from sklearn.cluster import KMeans
```

```
In [5]: km = KMeans(n_clusters=3) # I know the dataset!
```

```
In [6]: km.fit(X_iris)
```

```
Out[6]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
n_clusters=3, n_init=10, n_jobs=None, precompute_distances='auto',
random_state=None, tol=0.0001, verbose=0)
```

Score

Next we'll evaluate the method.

```
In [7]: km.score(X_iris) # same data we trained on
```

```
Out[7]: -78.8514414261457
```

But what does that mean?

```
In [8]: help(km.score)
```

```
Help on method score in module sklearn.cluster.k_means_:
```

```
score(X, y=None, sample_weight=None) method of sklearn.cluster.k_means_.KMeans instance
Opposite of the value of X on the K-means objective.
```

```
Parameters
-----
X : {array-like, sparse matrix}, shape = [n_samples, n_features]
New data.
```

```
y : Ignored
    not used, present here for API consistency by convention.
```

```
sample_weight : array-like, shape (n_samples,), optional
    The weights for each observation in X. If None, all observations
    are assigned equal weight (default: None)
```

```
Returns
-----
score : float
    Opposite of the value of X on the K-means objective.
```

"the K-means objective" is a measure of how good our clustering is: we try to minimise the within-cluster sum of squared distances. Our `score` is the negative, so larger values (closer to zero) are better.

Using the model

In unsupervised learning, what is the output? We can look at the labels for the training data. Note that these are just labels and the ordering (0, 1, 2) is arbitrary. Although there are three species in the data, we have not used the species labels in training. Our hope is only that we'll find approximately the same three clusters in our unsupervised learning.

```
In [9]: km.labels_
```

```
Out[9]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 
```




Scikit-Learn: Representation Learning

Different names for the same idea:

- Representation learning
- Learning embeddings
- Manifold learning
- Dimensionality reduction

There are two main goals:

- Reduce dimensionality to save memory/CPU in later supervised/unsupervised tasks, or for visualisation
- Achieve a "better" representation for data which allows downstream algorithms to succeed.

It can be linear:

- Principal components analysis (PCA)
- Random projection

It can be non-linear:

- t-SNE
- Multi-dimensional scaling
- Locally linear embedding

In this notebook, we'll see some examples of representation learning. We'll also see how the Scikit-Learn API makes it easy to try out multiple techniques at once.

The usual imports for plotting.

```
In [2]: import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

We'll import a few representation learning models, some for the exercises.

```
In [3]: from sklearn.manifold import MDS, TSNE, LocallyLinearEmbedding
from sklearn.decomposition import PCA
```

We're going to work with a small dataset of hand-written digits. (Not MNIST.)

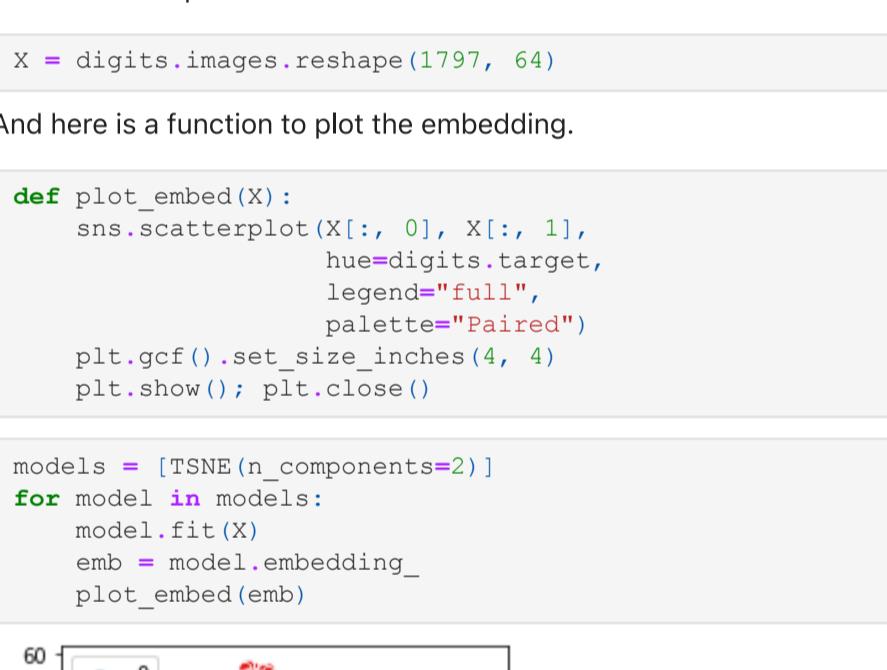
```
In [4]: from sklearn.datasets import load_digits
digits = load_digits()
digits.images.shape
```

```
Out[4]: (1797, 8, 8)
```

Here is a nice function (from PDSH) for plotting some of the images with labels.

```
In [6]: # from PDSH -- for illustration only - you don't need to understand details
fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                       subplot_kw={'xticks':[], 'yticks':[]},
                       gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
            transform=ax.transAxes, color='green')
```



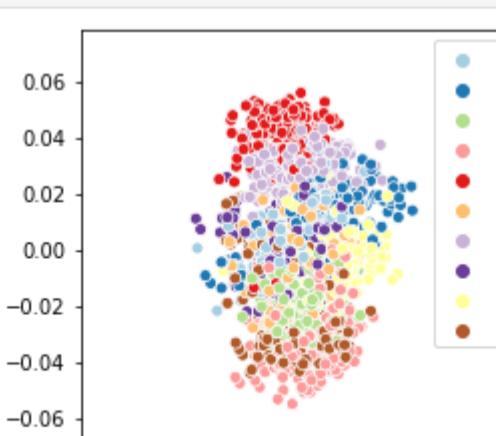
Scikit-Learn expects each X as a flat vector.

```
In [7]: X = digits.images.reshape(1797, 64)
```

And here is a function to plot the embedding.

```
In [29]: def plot_embed(X):
    sns.scatterplot(X[:, 0], X[:, 1],
                    hue=digits.target,
                    legend="full",
                    palette="Paired")
    plt.gcf().set_size_inches(4, 4)
    plt.show(); plt.close()
```

```
In [31]: models = [TSNE(n_components=2)]
for model in models:
    model.fit(X)
    emb = model.embedding_
    plot_embed(emb)
```



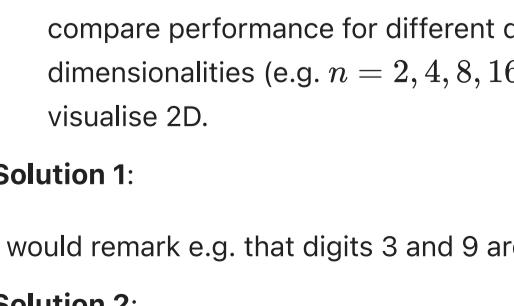
Observe the range of x and y values. We don't try to interpret these axes. In fact, one could rotate the whole thing by 90 degrees, or any other amount, and it would be the "same" result -- for visualisation, or for downstream algorithms.

PCA

`sklearn.decomposition.PCA` is the class to use for Principal Components Analysis. In principle it should have an API similar to t-SNE. But there are a couple of differences:

```
In [25]: models = [PCA(n_components=2)]

for model in models:
    model.fit(X.T) # transpose here and below
    emb = model.components_.T # not .embedding_
    plot_embed(emb)
```



PCA seems not to produce a nice separated embedding, like t-SNE. But it has the important advantage that after fitting on X , we can later map new data into the embedding space. In Scikit-Learn's terms, `PCA` is *inductive* because it has a `transform` method. `TSNE` is *transductive* because it doesn't. As a result, `PCA` can be used to transform data for downstream supervised learning tasks but `TSNE` can't.

Exercise:

1. More than any other ML technique, it's essential to look at the result of representation learning, and judge it subjectively. Look at the embedding produced by t-SNE. Are digits in "good" places?
2. Add some other models so that they run in our loop. Consider `MDS`, `LocallyLinearEmbedding`, or whatever you like from the same module. Use a 2D embedding in each case. Do they need different hyperparameters? Note: `MDS.fit()` takes ~60s on a 5-year-old Thinkpad.

Exercise:

1. Each model has a different definition for `score()`, so it is not useful to compare across models. But it is interesting to compare performance for different dimensions. Make a table showing the score for each model, and for several different dimensionalities (e.g. $n = 2, 4, 8, 16$, bearing in mind the digits themselves are 64D) and bearing in mind that we can only visualise 2D.

Solution 1:

I would remark e.g. that digits 3 and 9 are close by each other, and so are 4 and 7. These reflect similarities in digit shape.

Solution 2:

```
models = [TSNE(n_components=2), LocallyLinearEmbedding(n_components=2), MDS(n_components=2)]
```

should do the trick. In my run, `LocallyLinearEmbedding` does not do well, but it can be useful for some datasets.

Solution 3:

Just use eg `n_components = [2, 4, 8, 16]` and iterate `for n_components in n_components:`.

Scikit-Learn: Summary

We have now seen supervised, unsupervised, and representation learning. All use the Scikit-Learn *Estimator* API. In this notebook, we'll look at a few more details of that API, and then briefly summarise what is in Scikit-Learn.

The Estimator API

"the main API implemented by scikit-learn is that of the estimator. An estimator is any object that learns from data; it may be a classification, regression or clustering algorithm or a transformer that extracts/filters useful features from raw data."

An estimator object has a `fit` method which may accept `X` or `X` and `y`:

```
estimator.fit(X)
# OR...
estimator.fit(X, y)
```

A predictor object is an estimator which also has a `predict` method:

```
estimator.predict(X)
```

Some predictors whose `predict` output is discrete (e.g. clustering or classification) will implement either `predict_proba` or `decision_function`, which return real values:

```
estimator.predict_proba(X)
```

The `predict` method is then usually implemented as a threshold over the result, e.g.:

```
def predict(self, X):
    return self.predict_proba(X) > self.threshold
```

A *transformer* is an estimator with either `transform` or `fit_transform`. Often transformers are representation learning approaches. `fit_transform` is just a shortcut to calling `fit` and then `transform` on the same data.

A *model* is an object with a `score` method which evaluates how good it is, e.g. R^2 or classification accuracy. Higher is always better.

```
score(X)
```

or

```
score(X, y)
```

For a little more detail on this, see https://scikit-learn.org/stable/tutorial/statistical_inference/settings.html#estimators-objects

A typical example is `LogisticRegression`:

```
lr = LogisticRegression()
lr.fit(X, y)
lr.score(X_test, y_test)
lr.predict_proba(X_query)
lr.predict(X_query)
```

Semantics

Calling `predict` before `fit` is disallowed.

After fitting, the estimator object will usually have some new attributes named with a trailing underscore, e.g. `lr.coef_` and `lr.intercept_` for linear regression, or `support_vectors_` and some others for an SVM.

Also, a call to `fit` over-writes the result of any previous call.

```
lr.fit(X1, y1)
lr.fit(X2, y2)
```

has the same effect as just: `lr.fit(X2, y2)`.

(A few estimators allow `warm_start=True` in the constructor, or `partial_fit(X, y)`, so that we can pick up training where we left off. But we won't cover these.)

Scikit-Learn Summary Table

Problem	Example	Technique	Create	Fit	Evaluate	Use
Unsupervised						
Clustering	Customer segmentation	<i>k</i> -means	<code>km = KMeans(nclusters=2)</code>	<code>km.fit(X)</code>	<code>km.score(X)</code>	<code>km.labels_</code>
Density estimation	Plotting a distribution smoothly	Kernel density estimation	<code>kde = KernelDensity()</code>	<code>kde.fit(X)</code>	(none)	<code>kde.score_samples(new_X)</code>
Representation learning	Visualising data	Multi-dimensional scaling	<code>mds = MDS()</code>	<code>mds.fit(X)</code>	(none)	<code>mds.embedding_</code>
Supervised						
Regression	Predict car values	Linear regression	<code>lr = LinearRegression()</code>	<code>lr.fit(train_X, train_y)</code>	<code>lr.score(test_X, test_y)</code>	<code>lr.predict(new_X)</code>
Classification	Predict customer churn	Support vector machines	<code>svm = SVC()</code>	<code>svm.fit(train_X, train_y)</code>	<code>svm.score(test_X, test_y)</code>	<code>svm.predict(new_X)</code>

And here's a decision tree from a Scikit-Learn collaborator. As a decision aid (to help you choose an algorithm) it's of dubious value, but as a summary it's nice!



<https://peekaboo-vision.blogspot.com/2013/01/machine-learning-cheat-sheet-for-scikit.html>

Finally, this page lists all the main packages and so gives an overview of what is available:

<https://scikit-learn.org/stable/modules/classes.html>

Scikit-Learn: Hyperparameter Tuning and Cross-Validation

When approaching an ML problem we often train multiple models. There are at least three possibilities:

- Different models *per se*, e.g. Logistic Regression versus SVM
- Tuning hyperparameter values (aka **model selection**)
- Different features (**feature selection**)

This notebook is about hyperparameter tuning only. We'll cover feature selection in the next.

Hyperparameter Tuning with a Train-Test Split

To get started, we'll consider a simple scenario: a single train-test split. How can we carry out hyperparameter tuning using Scikit-Learn?

Let's try the `RandomForestRegressor` which wins a lot of Kaggle-type competitions.

```
In [2]: import itertools
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
import pandas as pd
```

We'll use the Boston housing dataset like before.

```
In [4]: Xy = pd.read_csv("data/boston.csv", index_col=0)
Xy.head()
```

```
Out[4]:
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

```
In [3]: X = Xy.drop("MEDV", axis=1)
X.head()
```

```
Out[3]:
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

```
In [4]: y = Xy["MEDV"]
y.head()
```

```
Out[4]: 0    24.0
1    21.6
2    34.7
3    33.4
4    36.2
Name: MEDV, dtype: float64
```

As we said, we are going to use a single train-test split, as is already familiar.

```
In [5]: X_train, X_test, y_train, y_test = train_test_split(
    X, y)
```

Now we are ready to run an experiment. Let's use `itertools.product` as before.

```
Out[6]: n_estimatorsss = [1, 5, 10, 50, 100, 500, 1000]
max_depths = [2, 4, 6, 8, 10, None] # NB None
for n_estimators, max_depth in (
    itertools.product(n_estimatorsss, max_depths)):
    rf = RandomForestRegressor(
        n_estimators=n_estimators,
        max_depth=max_depth)
    rf.fit(X_train, y_train)
    fmt = "%d\t%s\t%.2f"
    print(fmt % (n_estimators, str(max_depth),
                 rf.score(X_test, y_test)))
```

```
1      2      0.67
1      4      0.84
1      6      0.84
1      8      0.79
1      10     0.80
1      None    0.79
5      2      0.75
5      4      0.86
5      6      0.89
5      8      0.89
5      10     0.89
5      None    0.89
10     2      0.73
10     4      0.87
10     6      0.90
10     8      0.90
10     10     0.89
10     None    0.90
50     2      0.79
50     4      0.88
50     6      0.90
50     8      0.92
50     10     0.92
50     None    0.91
100    2      0.77
100    4      0.88
100    6      0.91
100    8      0.91
100    10     0.92
100   None    0.92
500    2      0.78
500    4      0.89
500    6      0.91
500    8      0.91
500    10     0.92
500   None    0.92
1000   2      0.78
1000   4      0.88
1000   6      0.91
1000   8      0.91
1000  10     0.92
1000 None    0.92
```

Remember that by default the `score` for regression is the coefficient of determination R^2 , where higher is better. The best I observe is $R^2 = 0.91$ with e.g. (50, 8).

Cross-Validation

Disadvantages of a single train-test split:

- Vulnerable to a single random decision (e.g. many "easy" examples in the test set)
- Some of the data doesn't contribute to training

Cross-validation solves these problems by splitting the data into k folds, and then training k times, each time on $1 - 1/k$ of the data, and validating on the remaining $1/k$:



https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation

Scikit-Learn provides easy interfaces for cross-validation.

```
In [7]: from sklearn.model_selection import cross_val_score
rf = RandomForestRegressor(n_estimators=50,
                           max_depth=8)
cross_val_score(rf, X_train, y_train, cv=5)
```

```
Out[7]: array([0.82305379, 0.74622642, 0.83550338, 0.8830016 , 0.8856796 ])
```

Here, we see a *big* difference in performance due to different folds. This is a warning not to blindly trust the result of any single test set.

Next, we can use CV to help tune hyperparameters.

```
In [8]: n_estimatorsss = [5, 50] # to speed things up
max_depths = [2, 8, None] # we'll try fewer values
for n_estimators, max_depth in (
    itertools.product(n_estimatorsss, max_depths)):
    rf = RandomForestRegressor(
        n_estimators=n_estimators,
        max_depth=max_depth)
    # NB mean
    score = cross_val_score(rf, X_train, y_train,
                            cv=5).mean()
    fmt = "%d\t%s\t%.2f"
    print(fmt % (n_estimators, str(max_depth), score))
```

```
5      2      0.71
5      8      0.80
5      None    0.76
50     2      0.74
50     8      0.83
50     None    0.83
```

Grid Search

What we have done here, and earlier, is a *grid search*: we tried out all combinations of hyperparameter values. We did it manually, but let's never do that again: Scikit-Learn provides it for us. We provide a `dict` giving the values to be tried.

```
In [9]: from sklearn.model_selection import GridSearchCV
param_grid = {'n_estimators': [5, 50],
              'max_depth': [2, 8, None]}
grid = GridSearchCV(RandomForestRegressor(),
                    param_grid, cv=5)
```

```
In [10]: grid.fit(X_train, y_train)
grid.best_params_
```

```
/Users/jmmcd/anaconda3/lib/python3.7/site-packages/sklearn/model_selection/_search.py:813: DeprecationWarning:
The default of the 'iid' parameter will change from True to False in version 0.22 and will be removed in 0.24. This will change numeric results when test-set sizes are unequal.
  DeprecationWarning)
```

```
Out[10]: {'max_depth': None, 'n_estimators': 5}
```

Multiple score functions

There is also the `cross_validate` function which allows us to measure multiple `score` functions and also timing information.

```
In [19]: from sklearn.model_selection import cross_validate
scoring = ['r2', 'neg_mean_squared_error']
scores = cross_validate(rf, X_train, y_train,
                        scoring=scoring, cv=5)
for key in scores:
    print("%s %.2f" % (key, scores[key].mean()))
```

```
fit_time 0.05
score_time 0.01
test_r2 0.85
test_neg_mean_squared_error -12.43
```

Leave-one-out Cross-Validation

There is even *leave-one-out* cross-validation, where the number of folds equals the number of training samples. However, it doesn't work when `score` is R^2 !

```
from sklearn.model_selection import LeaveOneOut
rf = RandomForestRegressor(n_estimators=50, max_depth=8)
cross_val_score(rf, X, y, cv=LeaveOneOut())
```

We'll see a warning:

```
UndefinedMetricWarning: R^2 score is not well-defined with less than two samples
```

This warning is correct. When running `leave-one-out`, we only get a single \hat{y} after each `fit`. Scikit-Learn tries to calculate R^2 on it, and it isn't well-defined. It would be better if Scikit-Learn would just save all the \hat{y} values and then calculate a single R^2 .

Probably they'll fix this soon. But to demonstrate, we could just use MSE instead.

```
In [14]: from sklearn.model_selection import LeaveOneOut
rf = RandomForestRegressor(n_estimators=50, max_depth=8)
scores = cross_val_score(rf, X, y, cv=LeaveOneOut(),
                        scoring="neg_mean_squared_error")
print(scores.mean())
```

```
-10.76049920030379
```

There are many more possibilities for model selection, including *stratified* CV:

- https://scikit-learn.org/stable/model_selection.html

But usually just choosing (say) a simple 5-fold CV in a grid search is good enough.

There is also the `cross_validate` function which allows us to measure multiple `score` functions and also timing information.

```
In [19]: from sklearn.model_selection import cross_validate
scoring = ['r2', 'neg_mean_squared_error']
scores = cross_validate(rf, X_train, y_train,
                        scoring=scoring, cv=5)
for key in scores:
    print("%s %.2f" % (key, scores[key].mean()))
```

```
fit_time 0.05
score_time 0.01
test_r2 0.85
test_neg_mean_squared_error -12.43
```

Leave-one-out Cross-Validation

There is even *leave-one-out* cross-validation, where the number of folds equals the number of training samples. However, it doesn't work when `score` is R^2 !

```
from sklearn.model_selection import LeaveOneOut
rf = RandomForestRegressor(n_estimators=50, max_depth=8)
cross_val_score(rf, X, y, cv=LeaveOneOut())
```

We'll see a warning:

```
UndefinedMetricWarning: R^2 score is not well-defined with less than two samples
```

This warning is correct. When running `leave-one-out`, we only get a single \hat{y} after each `fit`. Scikit-Learn tries to calculate R^2 on it, and it isn't well-defined. It would be better if Scikit-Learn would just save all the \hat{y} values and then calculate a single R^2 .

Probably they'll fix this soon. But to demonstrate, we could just use MSE instead.

```
In [14]: from sklearn.model_selection import LeaveOneOut
rf = RandomForestRegressor(n_estimators=50, max_depth=8)
scores = cross_val_score(rf, X, y, cv=LeaveOneOut(),
                        scoring="neg_mean_squared_error")
print(scores.mean())
```

```
-10.76049920030379
```

There are many more possibilities for model selection, including *stratified* CV:

- https://scikit-learn.org/stable/model_selection.html

But usually just choosing (say) a simple 5-fold CV in a grid search is good enough.

Scikit-Learn: Pipelines

Pipelines

Often, we'll have several preprocessing steps and then our main model. For example, we might have a data with missing values which would also benefit from having the square of each feature -- ideas we discussed in the previous notebook/video.

```
In [2]: import numpy as np
X = np.array([[np.nan, 0, 3],
              [3, 7, 9],
              [3, 5, 2],
              [4, np.nan, 6],
              [8, 8, 1]])
y = np.array([14, 16, -1, 8, -5])
```

Pipelines

- A *pipeline* is sequence of Scikit-Learn estimators
- Each transforms the `X`
- All but the last must have `transform()`: its output becomes the input of the next `fit()` and `transform()`.



Motivation

- **Convenience and encapsulation:** Call `fit` and `predict` just once
- **Joint parameter selection:** grid search over parameters of pipeline components together
- **Safety:** avoid errors leaking test data into training.

```
In [1]: from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

model = make_pipeline(SimpleImputer(strategy='mean'),
                      PolynomialFeatures(degree=2),
                      LinearRegression())
```

```
In [4]: model.fit(X, y)
```

```
Out[4]: Pipeline(memory=None,
                  steps=[('simpleimputer',
                          SimpleImputer(add_indicator=False, copy=True, fill_value=None,
                                         missing_values=np.nan, strategy='mean',
                                         verbose=0)),
                         ('polynomialfeatures',
                          PolynomialFeatures(degree=2, include_bias=True,
                                             interaction_only=False, order='C')),
                         ('linearregression',
                          LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                                           normalize=False))],
                  verbose=False)
```

This is equivalent to writing:

```
X = SimpleImputer(strategy="mean").fit_transform(X)
X = PolynomialFeatures(degree=2).fit_transform(X)
model = LinearRegression()
model.fit(X, y)
```

ColumnTransformer

Often we'll want to transform some columns with e.g. one-hot encoding, and others with e.g. missing value imputation, and leave other columns alone.

See `ColumnTransformer` for this.

Reference

<https://scikit-learn.org/stable/modules/compose.html>

Exercise 1

Confirm that our trained `model` can handle `np.nan` transparently and makes sensible predictions, by passing in a query point in the *original* format.

Exercise 2

We mentioned that "convenience and encapsulation" are part of the motivation for using pipelines. One situation where this is especially true is when we are *saving* models to disk. Scikit-Learn doesn't provide its own method for this, but instead uses the `pickle` module built-in to the standard library. For any object `c`, we can write:

```
import pickle
pickle.dump(c, open("data/my_object.pkl", "wb"))
# later...
c2 = pickle.load(open("data/my_object.pkl", "rb"))
```

1. We have to open the file ourselves -- we pass a `file` object, not a filename, to `dump` and to `load`.
2. We have to read/write using *binary mode* (`"wb"` and `"rb"`) because the pickle format is binary, not plain-text.

So, the exercise is to save our trained Scikit-Learn pipeline model to disk, and then read it in again. Confirm that the model we read in from disk gives the same results as the model we wrote to disk.

See <https://scikit-learn.org/stable/tutorial/basic/tutorial.html#model-persistence>.

Exercise 3

Check how large is the model saved on disk, e.g. using `ls` on Unix or `dir` on Windows. For the curious: how does the size compare to a bare LR model?

Solution 1

```
In [11]: model.predict([[np.nan, 0.5, 3.5]])
```

```
Out[11]: array([13.91644421])
```

Solution 2

```
In [10]: import pickle
model = make_pipeline(SimpleImputer(strategy='mean'),
                      PolynomialFeatures(degree=2),
                      LinearRegression())
model.fit(X, y)
pickle.dump(model, open("data/LR_pipeline.pkl", "wb"))
model2 = pickle.load(open("data/LR_pipeline.pkl", "rb"))
model2.predict([[np.nan, 0.5, 3.5]])
```

```
Out[10]: array([13.91644421])
```

Solution 3

For me, the file is 1252 bytes:

```
$ ls -l LR_pipeline.pkl
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

252 28 Oct 19:18 LR_pipeline.pkl

OOP in Python

This notebook is about OOP in Python. Students who have not studied OOP concepts such as objects, class, constructors, methods, inheritance, and polymorphism in another language should refer to Downey, Chapters 15--18, and feel free to contact me for further resources and help.

Contents

- Defining classes
- Special double-underscore methods
- Inheritance and duck-typing

In Python, we define a class using `class`. The following, believe it or not, is a working class which can be instantiated.

```
In [6]: class C:  
    pass
```

Now we can make an object `c` of type `C`:

```
In [3]: c = C() # call the constructor
```

```
In [4]: print(c)
```

```
<__main__.C object at 0x7f7de4273160>
```

Now let's make a constructor. The `self` is the first argument of all methods and it refers to the object itself -- the equivalent of `this` in some languages.

```
In [8]: class C:  
    def __init__(self, data=17):  
        self.data = data  
c = C()  
print(c.data)
```

```
17
```

In Python, we don't have any restrictions on accessing fields:

```
In [9]: c.data += 1  
print(c.data)
```

```
18
```

A class has its own namespace. So it's fine to have a variable called `data` and a variable called `c.data` at the same time with different values.

Double-underscore methods

The `__init__` constructor is just one of several special double-underscore methods in Python. When we use some common Python things like `len` and `<`, it results in a call to a particular "dunder" method, for example:

call	translation
C(data)	C.__init__(data)
len(c)	c.__len__()
str(c)	c.__str__()
c < d	c.__lt__(d)

So, let's implement some of these.

```
In [2]: class C:  
    def __init__(self, data=17):  
        self.data = data  
    def __str__(self):  
        return f"C({self.data})"  
    def __lt__(self, other):  
        return self.data < other.data  
c1 = C()  
c2 = C(18)  
print(c1 < c2)
```

```
True
```

A free lunch

Sometimes implementing one special method is enough to get others "for free". We implemented `__lt__` for "less than", allowing `c1 < c2` to work. We didn't implement `__gt__`. Nevertheless, it works:

```
In [12]: c1 > c2
```

```
Out[12]: True
```

That's because Python knows that these operators are symmetric. If we wanted to define `>` to be non-symmetric to `<`, we could do so, just by implementing both `__lt__` and `__gt__` separately.

Inheritance

We can make a new class which has some behaviour (methods and fields) of its own, and inherit other behaviour from a superclass. The syntax is: put the parent (superclass) in parenthesis after `class C`.

```
In [17]: class D(C):  
    def hello(self):  
        print(f"Hello, my value is {self.data}")
```

This new class has composite behaviour:

```
In [23]: d = D(12)  
print(d) # inherited from C  
d.hello() # in D itself
```

```
C(12)  
Hello, my value is 12
```

```
super
```

Sometimes we want to inherit a method, but also add to it. This is especially common in `__init__`.

```
In [30]: class D(C):  
    def __init__(self, data=17):  
        super().__init__(data)  
        self.config = "specific to D"  
    def hello(self):  
        print(f"Hello, my value is {self.data}")  
d = D(15)  
print(d.config)
```

```
specific to D
```

Duck typing

Don't forget, we have duck typing! In fact, duck typing is the Python approach to *polymorphism*. For example:

```
In [24]: d < c
```

```
Out[24]: True
```

We can understand this by the substitution model:

```
d < c  
d.__lt__(c)  
d.data < c.data  
12 < 17  
True
```

Multiple inheritance

It is fairly common to use inheritance, and even chains of inheritance: class D inherits from C which inherits from B, etc. Any class which doesn't explicitly inherit from another class implicitly inherits from `object` -- a kind of null class which is "the root of the inheritance hierarchy".

However, we can also have *multiple inheritance*, where class D inherits from both B and C (and neither B nor C inherits from the other).



```
In [26]: class B:  
    def bzzz(self):  
        print("bzzz")  
class D(B, C):  
    def hello(self):  
        print(f"Hello, my value is {self.data}")
```

```
In [27]: d = D()  
d.bzzz()  
d.hello()
```

```
bzzz  
Hello, my value is 17
```

(A minor point: what happens if both B and C have a function with the same name? The Python *Method Resolution Order* comes into play to decide. We won't discuss this.)

Exercise

Try running this code, to observe what happens. Then edit the definition of `C`, implementing `__eq__` and `__le__`, to fix it.

```
C(17) == C(17)
```

```
C(17) <= C(18)
```

Solution

```
C(17) == C(17)
```

is `False` because if an object doesn't have `__eq__`, Python will fall back to `id`:

```
id(C(17)) == id(C(17)) # False
```

```
In [33]: class C:  
    def __init__(self, data=17):  
        self.data = data  
    def __str__(self):  
        return f"C({self.data})"  
    def __lt__(self, other):  
        return self.data < other.data  
    def __eq__(self, other):  
        return self.data == other.data  
    def __le__(self, other):  
        return self.data <= other.data
```




Scikit-Learn: Mimicking the Estimator API

As we have seen, Scikit-Learn has some infrastructure which helps to make ML projects run smoothly in practice, e.g.:

- Appropriate `score` methods for various models;
- Cross-validation e.g. `cross_validate_score` ;
- Pipelines.

If we are writing a new ML algorithm, it is natural to try to conform to the Scikit-Learn API so that our code will work well with this infrastructure. For example, if we conform to the Estimator API, then we can put our new algorithm in a list with other models and compare them.

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
class FabNewClassifier:
    pass # whatever
clfs = [LogisticRegression, SVC, FabNewClassifier]
for clf in clfs:
    clf = clf()
    clf.fit(X, y)
    print(clf.score(X, y))
```

To make this concrete, we're going to:

1. Invent a new classifier and implement it in Python with Numpy.
2. Re-factor it as a class with `fit`, `predict`, and `score` methods, which allows the above polymorphism to work.
3. *Inherit* from Scikit-Learn base classes to get uniformity and to get functionality "for free".

```
In [67]: import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy
%matplotlib inline
```

```
In [68]: iris = sns.load_dataset("iris")
X = iris.drop("species", axis=1).values
y = iris["species"].values
```

The classifier we'll implement is 1-nearest neighbours.

```
In [69]: def one_nn(X, y, Q):
    D = scipy.spatial.distance.cdist(X, [Q])
    nearest = np.argmin(D)
    print("Query", Q)
    print("nearest", nearest)
    print("X[nearest]", X[nearest])
    print("D[nearest]", D[nearest])
    print("y[nearest]", y[nearest])
    return y[nearest]
```

We can write it as a single function because k -NN has no separate training phase. We use `scipy.spatial.distance.cdist` to calculate all distances from the query `Q` to each point in the training data `X`. Then we return the `y` label of whichever point in `X` is nearest to `Q`. We print out everything that's happening just to help explain, but of course in real code it should just return a value, not print anything.

```
In [70]: Q = [5.75, 2.0, 4.0, 1.5]
one_nn(X, y, Q)
```

```
Query [5.75, 2.0, 4.0, 1.5]
nearest 53
X[nearest] [5.5 2.3 4. 1.3]
D[nearest] [0.43874822]
y[nearest] versicolor
```

```
Out[70]: 'versicolor'
```

Refactoring to mimic the Estimator API

We already know enough to refactor our 1-NN as an Estimator class. We have to provide:

- `fit(X, y)`
- `predict(X)`
- `score(X, y)`

```
In [40]: class OneNN:
    def fit(self, X, y):
        self.X = X
        self.y = y
    def predict(self, X):
        D = scipy.spatial.distance.cdist(self.X, X)
        nearest = np.argmin(D, axis=0)
        return self.y[nearest]
    def score(self, X, y):
        fx = self.predict(X)
        return np.mean(fx == y) # accuracy
```

`OneNN` is now an object. `fit` stores the `X` and `y` but doesn't actually do anything for 1-NN.

The biggest change here is that in Scikit-Learn, `predict` should accept a 2D array of query points, not a single point. So we've added `axis=0` to take account of that.

```
In [43]: onenn = OneNN()
onenn.fit(X, y)
Q3 = np.array([[5.75, 2, 4, 1.5],
               [5.0, 1.5, 2.4, 1.6],
               [4.6, 2.8, 2.2, 1.7]])
onenn.predict(Q3)
```

```
Out[43]: array(['versicolor', 'versicolor', 'versicolor'], dtype=object)
```

And our `score` method calculates an accuracy value. E.g. in 1-NN we will always get accuracy of 1 on training data. (Why?)

```
In [ ]: print(onenn.score(X[:3], y[:3]))
```

The Estimator API and inheritance

However, our `OneNN` is still not quite compatible with Scikit-Learn. There are some extra details which help to keep things uniform and make our lives easier. For a start, estimators should inherit from `sklearn.base.BaseEstimator`.

```
In [59]: from sklearn.base import BaseEstimator, ClassifierMixin
class OneNN(BaseEstimator, ClassifierMixin):
    def fit(self, X, y):
        self.X = X
        self.y = y
        return self
    def predict(self, X):
        D = scipy.spatial.distance.cdist(self.X, X)
        nearest = np.argmin(D, axis=0)
        return self.y[nearest]
```

A *Mixin* is a class M which is designed for the *multiple inheritance* scenario where a class C is designed to inherit from M and from some other class D as well. In other words, C is composed of M "mixed-in" with D.

In Scikit-Learn, there is (e.g.) a `ClassifierMixin` which has the `score` behaviour. This makes sense, because all classifiers should share the same `score` behaviour: our `OneNN` class should not implement a custom version.

```
In [74]: Q3 = np.array([[5.75, 2, 4, 1.5],
                  [5.0, 1.5, 2.4, 1.6],
                  [4.6, 2.8, 2.2, 1.7]])

onenn = OneNN().fit(X, y)
onenn.predict(Q3)
```

```
Out[74]: array(['versicolor', 'virginica', 'virginica'], dtype=object)
```

```
In [65]: onenn.score(X[:3], y[:3])
```

```
Out[65]: 1.0
```

More details of the API

- Here is a flavour of the "rules" of the API which are designed to keep things uniform and make both users and developers' lives easier.
- The arguments of `__init__` should be keyword arguments with defaults. So, calling `C()` (no arguments) will work.
 - In order to fit in pipelines, even unsupervised estimators need to accept `y=None`
 - The `fit` method should return `self`. This allows a nice "chained" usage `OneNN().fit(X, y).predict(Q3)`
 - "In iterative algorithms, the number of iterations should be specified by an integer called `n_iter`."
 - "You can check whether your estimator adheres to the scikit-learn interface and standards by running `utils.estimator_checks.check_estimator` on the class"
 - There is a template for new "contrib" projects: <https://github.com/scikit-learn-contrib/project-template/>
 - <https://scikit-learn.org/dev/developers/develop.html>
 - <https://scikit-learn.org/stable/modules/classes.html#module-sklearn.base>

Finite State Machines

For the last 30 years or so, machine learning methods have gradually taken over as the best methods for solving AI problems. In the 30 years before that, most approaches were *symbolic* and logical rather than numerical, probabilistic and learning-oriented. We sometimes call those approaches (half-joking) "good old-fashioned AI".

Task	GOFAI	ML-style AI
Games	Minimax (alpha-beta)	Reinforcement learning
Facts	Logical inference	Probabilistic inference
Natural language	Parsing with grammars	Recurrent neural networks
Agent behaviour	Finite state machines	Reinforcement learning

A finite state machine is a neat representation for behaviour. It's still used quite a lot in applications:

- Parsing computer language
- Representing agent behaviour in video games
- Elevators? (A typical textbook example!).

One reason why computer scientists like FSMs is that a computer IS an FSM -- an amazingly complex one. We'll study simple ones.

We'll motivate the idea from an unusual angle and then see an example in game AI.

Dispatch table

Suppose we had some code like this (an artificial example):

```
In [1]: def g0(): print("g0")
def g1(): print("g1")
def g2(): print("g2")

def f(a, b, c):
    if a and c:
        if b:
            g0()
        else:
            g2()
    elif c and not a:
        g1()
    elif c or a:
        g2()
```

It's hard to predict what will happen for any input!

```
In [2]: f(0, 1, 1)
```

g1

There could even be inputs which don't run any `g`:

```
In [3]: f(0, 0, 0)
```

A *dispatch table* is a different representation for `f`:

```
In [5]: dt = {
    (1, 0, 1): g0, # a and c
    (1, 1, 1): g0, # a and c
    (0, 0, 1): g1, # c and not a
    (0, 1, 1): g1, # c and not a
    (1, 0, 0): g2, # c or a
    (1, 1, 0): g2, # c or a
}

def f(a, b, c): dt[(a, b, c)]()
```

```
In [6]: f(0, 1, 1)
```

g1

With this new representation, we can more easily see that some combinations `(a, b, c)` are not accounted for at all, because they're not in the dictionary `dt`. They'll also throw a `KeyError`.

```
In [10]: f(0, 0, 0)
```

```
-----  
KeyError: Traceback (most recent call last)  
<ipython-input-10-d94e03aff689> in <module>  
----> 1 f(0, 0, 0)
```

```
<ipython-input-8-6d9dd5649f5a> in f(a, b, c)
     8     (1, 1, 0): g2, # c or a
     9
--> 10     dt[(a, b, c)]()
```

```
KeyError: (0, 0, 0)
```

We might decide to have a default to deal with that.

```
In [11]: def f(a, b, c):
    try:
        dt[(a, b, c)]()
    except KeyError:
        pass # the original f does nothing in this case
```

Our claim is that this dispatch table representation may be useful to help us write maintainable and understandable code.

Finite State Machines

A FSM is a generalisation of a dispatch table in two ways.

1. There are several "states". Every state has its own dispatch table.
2. Every time the function runs with some input, it may not only do something by calling a function, it may also *change state*.

It's often useful to think of the states as representing *internal* data, and the inputs as input from the environment.

Example: an FSM for a ghost in "Pacman"

From <http://mentalfloss.com/article/49068/how-win-pac-man>

Here's a video to remind us of the gameplay.

Inspired by Yannakakis and Togelius, *Artificial Intelligence and Games*

This FSM doesn't have any *actions*. It's just a mapping of the form:

$(\text{state}, \text{input}) \rightarrow \text{state}$

The states are in circles: **Seek Pacman**, **Evaude Pacman**, **Go to regenerate**. The possible inputs are Eaten by Pacman, Pacman has power pill, Power pill time up, Finished regenerating.

E.g. when in state **Evaude Pacman**, if we receive the *input* Power pill time up, we'll transition to **Seek Pacman**.

In our example, the states are "high-level" states so there would have to be some extra logic in the game.

We also make an important assumption: no unexpected input will ever appear. E.g. we'll never get input Eaten by Pacman while in the **Go to regenerate** state. If we prefer not to make that assumption, we have to add appropriate arrows, one for every possible input on every state.

Implementation

In the dispatch table, we just had a function `def f(a, b, c): dt[(a, b, c)]()`. If we wanted to change the function, we would just change `dt`.

Similarly, all FSMs share the same code implementation, and differ in their *data*.

Here's the data. It uses the same idea as in the dispatch table: a dictionary mapping "inputs" to outputs, but here the "inputs" are composed of the current state and the input from the environment.

```
In [10]: SISAs = {
    # (state, input): state
    ("Seek Pacman", "Pacman has power pill"):
        "Evaude Pacman",
    ("Seek Pacman", "Eaten by Pacman"):
        "Go to regenerate",
    ("Evaude Pacman", "Power pill time up"):
        "Seek Pacman",
    ("Evaude Pacman", "Eaten by Pacman"):
        "Go to regenerate",
    ("Go to regenerate", "Finished regenerating"):
        "Seek Pacman"
}
```

Every FSM has a start state. Some also have one or more end states: whenever we reach an end state, we stop executing. If there's no end state, as here, we keep running until the user chooses to stop.

```
In [11]: start_state = "Seek Pacman"
end_states = set() # empty set, no end states
```

Here's one way to write the code:

```
In [1]: def fsm(SISAs, start_state, end_states, input_symbols):
```

```
    state = start_state
    # (current_state, input_symbol) -> next_state
```

```
    for input_symbol in input_symbols:
```

```
        yield state
```

```
        if state in end_states:
```

```
            break
```

```
        try:
```

```
            state = SISAs[state, input_symbol]
```

```
        except KeyError:
```

```
            pass # stay in same state
```

Now let's suppose these are the sequence of game events:

```
In [16]: inputs = [
    "Pacman has power pill",
    "Power pill time up",
    "Pacman has power pill",
    "Eaten by Pacman",
    "Finished regenerating"
]
```

And here is how we can use it:

```
In [17]: for state in fsm(SISAs, start_state, end_states, inputs):
```

```
    print(state)
```

Seek Pacman

Evaude Pacman

Seek Pacman

Evaude Pacman

Go to regenerate

In a real application, our `for`-loop could be more complex, with some other function calls and logic.

Exercises

In the lab this week we'll see some exercises using FSMs to model a call centre automated voice response system.

Graphs

As we know, a *graph* is the word often used in maths and computer science for a *network* -- a collection of objects with connections among them.

Graph theory is a beautiful and fascinating branch of mathematics with many applications in computer science, artificial intelligence, and applications.

Don't confuse a graph (network) with a graph (plot).

Some of the disparate terminology we use:

object	connection
node	edge
vertex	arc
point	line

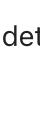


Wiki

The *founding problem of graph theory*: is it possible to traverse all the bridges of Konigsberg and end back at the starting-point without re-crossing any? Here the *nodes* are pieces of land and *edges* are bridges.



A practical problem: what is the shortest path from one location to another, via a network of roads?



Credit: <http://bitcoinwiki.co/wp-content/uploads/censorship-free-social-network-akasha-aims-to-tackle-internet-censorship-with-blockchain-technology.jpg>

A classic social networks problem: who are the most influential people? How many steps or "degrees of separation" from one to the next?



Inspired by Yannakakis and Togelius, *Artificial Intelligence and Games*

FSMs are graphs, where states are nodes and state transitions are *directed edges* and input symbols are edge labels.

With directed edges, we draw them as an arrow (as in the FSM) because order matters: $i \rightarrow j$ is not the same as $j \rightarrow i$.

The web

- Web pages (URLs) are nodes; hyperlinks are directed edges.



Wiki

The *PageRank* problem: which pages are the most trusted? A page is trusted if other highly trusted pages link to it. But isn't this circular?

Edge properties

In all the examples we've seen, nodes have names or just integer labels.

Usually edges are just plain connections with no properties, names or labels (sometimes direction). Sometimes edges do have properties such as a numerical *weight* per edge.

E.g. in a **water grid**, each edge might represent a pipe, and each pipe might have a different capacity represented as a number in litres per second.

Collaboration graphs

E.g. in scientific research: authors are nodes; there is an edge where two authors have co-authored a paper.

Here edges are *undirected* because saying "a co-authored with b" is the same as saying "b co-authored with a".

A classic problem of social networks: can we automatically detect communities?

Electrical grids

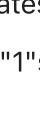
- The electricity network: power stations, transformers, and consumers are nodes; electricity wires are edges.



A classic logistics problem: on a small island, we have a power station and a set of consumers. Every meter of electricity wires costs money. What wires should we build to save money but ensure everyone is connected?

Family trees

A *tree* is a graph with no cycles. Your family tree is a directed tree *rooted* at you, with an edge from each person to each of their parents.



Time.com

By the way, it's not *really* a tree, if we go back far enough.

Abstract syntax tree

An AST is an internal representation for computer code. The Python interpreter, the C++ compiler, and all other interpreters and compilers translate our source code into an AST before executing it.

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```



Tangent: Python has some nice built-in tools for *introspection* including looking at the AST of a piece of Python code. It's not quite as clean as the example above because it includes extra technical details.

```
import ast
s = open("code/fib2.py").read()
n = ast.parse(s)
print(ast.dump(n))

@Module(body=[FunctionDef(name='fib', args=arguments(args=[arg(arg='n', annotation=None)], varargs=None, kwonlyargs=[], kw_defaults=[], defaults=[]), body=[If(test=Compare(left=Name(id='n', ctx=Load()), ops=[Lt()]), comparators=[Num(n=2)]), Return(value=BinOp(left=Name(id='n', ctx=Load()), op=Add(), right=Num(n=1))], orelse=[Return(value=BinOp(left=Name(id='n', ctx=Load()), op=Sub(), right=Num(n=1))], keywords=[], op=Add(), right=Call(func=Name(id='fib', ctx=Load()), args=[BinOp(left=Name(id='n', ctx=Load()), op=Sub(), right=Num(n=2))], keywords=[]))], decorator_list=[], returns=None)"
```

(The details of this are not examinable.)

Project planning

In project management, we often decompose a project into a large list of tasks to be accomplished by many different people using various resources. Some of these tasks have *dependencies*: task j cannot be started until we have the output of task i . This induces a directed graph.

- What would a *cycle* in such a graph mean?

Assuming no cycles, we can perform a *topological sort*, which just means ordering the tasks so that for any dependency (i, j) , task j is sorted after task i . Creating this ordering allows us to create a timeline and identify tasks which can be done in parallel.

Exercise. Suppose we have the following tasks. Identify the dependencies, draw the dependency graph to confirm there are no cycles, then write down a topological sort. Which tasks can be done in parallel?

tasks	more tasks	more tasks
get coffee from fridge	fill mocha with water	put coffee in mocha
close mocha	heat mocha	get cup
get spoon	get sugar from cupboard	put sugar in cup
pour coffee into cup	stir	put coffee away

put sugar away

Software installation

A related example is the dependency graph of software libraries. As we know:

- Seaborn can only work if Matplotlib is installed;

- Scipy requires Numpy;

- Pandas requires Numpy;

- and so on.

A classic problem in software management: recursively finding all the necessary dependencies for a library.

Next time you install a new library using Anaconda, take a look at all the other libraries that it depends on.

Data-flow graphs

Nodes are computations, and edges represent the results of computations being used as inputs in later computations.

From Hazelcast.com

Node and graph terminology and properties

- Order: the number of nodes in the graph
- Size: the number of edges in the graph.

Cycles

- Cycle: a sequence of 2+ nodes with edges allowing travel from start, along sequence, leading back to start
- Directed cycle: in a directed graph, a cycle exists only if this travel is consistent with edge direction
- A graph with no cycles is called *acyclic*.

- The *degree* of node n is the number of neighbours

- In a *directed graph*, we have a separate *out-degree* and *in-degree*

- In a graph with *weighted edges*, the degree is the sum of relevant edge weights.

Graph representations

There are several possible representations for a graph:

- Adjacency matrix
- List of edges
- Adjacency-list
- dict-of-dicts-of-dicts, as used by NetworkX, is a variant of adjacency-list.

Adjacency matrix

In an adjacency matrix, we represent a graph of n nodes as an $n \times n$ matrix. In a binary adjacency matrix, a 0 in location (i, j) indicates no edge present between nodes i and j ; a 1 indicates the edge is present.

If our edges have weights, we just put those instead of the "1"s.

Conclusion

Graph theory is the study of things that can be represented as graphs. And a lot of things can be represented as graphs.

Classroom MScAI students can study more in the module Web and Network Science, CT5113, Dr Conor Hayes, Semester 2.

Exercises

- The "adjacency lists" representation is really more like a dictionary, where every node maps to a list of its neighbours, i.e. the nodes which are *adjacent* to it. Here is a graph in adjacency-lists format:

```
G = {
    0: [1, 2, 3],
    1: [0, 3],
    2: [0],
    3: [0, 1]
```

Draw this on paper. Write down its adjacency matrix on paper.

- Calculate the sum of row 0 in the adjacency matrix. What does it mean?

- What do you observe about the diagonal?

- Suppose you saw a 1 on the diagonal. How could this be represented on paper? Think of a real-world situation where this would be useful.

Exercise. Suppose we have the following tasks. Identify the dependencies, draw the dependency graph to confirm there are no cycles, then write down a topological sort. Which tasks can be done in parallel?

tasks	more tasks	more tasks
get coffee from fridge	fill mocha with water	put coffee in mocha
close mocha	heat mocha	get cup
get spoon	get sugar from cupboard	put sugar in cup
pour coffee into cup	stir	put coffee away

put sugar away

Software installation

A related example is the dependency graph of software libraries. As we know:

- Seaborn can only work if Matplotlib is installed;

- Scipy requires Numpy;

- Pandas requires Numpy;

- and so on.

A classic problem in software management: recursively finding all the necessary dependencies for a library.

Next time you install a new library using Anaconda, take a look at all the other libraries that it depends on.

Data-flow graphs

Nodes are computations, and edges represent the results of computations being used as inputs in later computations.

From Hazelcast.com

Node and graph terminology and properties

- Order: the number of nodes in the graph

- Size: the number of edges in the graph.

Cycles

- Cycle: a sequence of 2+ nodes with edges allowing travel from start, along sequence, leading back to start

- Directed cycle: in a directed graph, a cycle exists only if this travel is consistent with edge direction

- A graph with no cycles is called *acyclic*.

- The *degree* of node n is the number of neighbours

- In a *directed graph*, we have a separate *out-degree* and *in-degree*

- In a graph with *weighted edges*, the degree is the sum of relevant edge weights.

Graph representations

There are several possible representations for a graph:

- Adjacency matrix
- List of edges
- Adjacency-list
- dict-of-dicts-of-dicts, as used by NetworkX, is a variant of adjacency-list.

Adjacency matrix

In an adjacency matrix, we represent a graph of n nodes as an $n \times n$ matrix. In a binary adjacency matrix, a 0 in location (i, j) indicates no edge present between nodes i and j ; a 1 indicates the edge is present.

If our edges have weights, we just put those instead of the "1"s.

Conclusion

Graph theory is the study of things that can be represented as graphs. And a lot of things can be represented as graphs.

Classroom MScAI students can study more in the module Web and Network Science, CT511

NetworkX

NetworkX <https://networkx.github.io/> is a nice library of graph algorithms in Python. It can be installed via Anaconda.

We previously saw several possible representations for graphs which can be used in computer programs:

- Adjacency matrix
- List of edges
- Adjacency lists

The adjacency list format is like this. For each node, we store its list of neighbours:

```
G = {
    0: [1, 2, 3],
    1: [0, 3],
    2: [0],
    3: [0, 1],
    4: []
}
```

Next, we'll see that the NetworkX representation is just an extension of this, allowing for each edge to have extra properties such as weights.

```
In [2]: G = {
    0: {1: {"w": 0.5}, 2: {"w": 0.1}, 3: {"w": 0.1}},
    1: {0: {"w": 0.5}, 3: {"w": 0.3}},
    2: {0: {"w": 0.1}},
    3: {0: {"w": 0.1}, 1: {"w": 0.3}},
    4: {}
}
```

One way to make a graph in NetworkX is just to build it up by adding nodes and then edges, as follows. (Later we'll see how to read graphs in.)

```
In [3]: import networkx as nx # conventional import
G = nx.Graph()
G.add_nodes_from(range(5))
```

```
In [4]: G.nodes()
```

```
Out[4]: NodeView((0, 1, 2, 3, 4))
```

```
In [5]: G.edges()
```

```
Out[5]: EdgeView([])
```

```
In [6]: G.add_edge(0, 1, w=0.5)
G.add_edge(0, 2, w=0.1)
G.add_edge(0, 3, w=0.1)
G.add_edge(1, 3, w=0.3)
```

```
In [7]: G.edges()
```

```
Out[7]: EdgeView([(0, 1), (0, 2), (0, 3), (1, 3)])
```

```
In [8]: G.edges(data=True)
```

```
Out[8]: EdgeDataView([(0, 1, {'w': 0.5}), (0, 2, {'w': 0.1}), (0, 3, {'w': 0.1}), (1, 3, {'w': 0.3})])
```

And now, notice that `G` itself functions as a `dict`, mapping from a node to its adjacency list **with edge properties**:

```
In [9]: for node in G.nodes():
    print(node, ":", G[node])
0 : {1: {'w': 0.5}, 2: {'w': 0.1}, 3: {'w': 0.1}}
1 : {0: {'w': 0.5}, 3: {'w': 0.3}}
2 : {0: {'w': 0.1}}
3 : {0: {'w': 0.1}, 1: {'w': 0.3}}
4 : {}
```

The nodes don't have to be integers! They can be strings, floats, or arbitrary (hashable) Python objects.

(Remember mutable objects such as lists are not hashable and cannot be stored in dictionaries.)

- `G = nx.Graph() # default, undirected`
- `G = nx.DiGraph() # directed graph`

Algorithms

- Creating standard graphs and random graphs
- Breadth-first and depth-first traversal
- Connectivity
- Communities, cliques, clusters, etc.
- Isomorphism (are two graphs the same if we ignore labels)

More algorithms

- Centrality e.g. PageRank and related algorithms
- Cycles
- Shortest paths
- Max-flow/min-cut
- Diameter
- Linear algebra methods on adjacency matrices

Exercise

We'll see how to read in a graph to NetworkX from a plain-text format, and run some algorithms on it. The scenario is: on a small island, we are just about to finish building a power plant at town 0. All the towns are already connected by a road network. The Minister has decided to build the electricity network along the roads. We want to achieve connectivity to all towns at minimum cost. Our data is stored in `data/power_plant_edgelist.csv`.

1. As we can see in the first few lines below, it is just an edge-list with weights. Why are we not worried about the possibility of isolated nodes, not captured by the edge list?

```
In [10]: import networkx as nx
fname = "data/power_plant_edgelist.csv"
f = open(fname)
for i in range(10):
    print(f.readline(), end="")
f.close()
```

This file contains the road structure

on a small island. Towns are

at nodes 0-13. Each road segment is

notated as:

town town distance

0 1 0.200000

0 2 0.223607

0 3 0.500000

0 4 0.282843

0 5 0.500000

Exercises

1. Read in this file using `nx.read_edgelist`. It should be an undirected, weighted graph.

2. Confirm that the road network is fully connected. Use `nx.is_connected()`.

3. Use Kruskal's Minimum Spanning Tree to find the lowest-cost electricity network.

4. The Minister is planning to take a drive in her Mercedes when she comes to cut the ribbon. She will travel from the power plant to the most distant village, using the fastest route. What is that distance?

Solutions

Use `nx.read_edgelist?` to get help. `create_using` tells `nx` that it should create a `DiGraph` (directed graph), and `nodetype` tells `nx` to convert each node from `str` to `int`.

The tricky part is the syntax for the edge data, in this case the weights. It is not a tuple ("weight", float) -- it is a sequence of tuples, one tuple for each piece of edge data. There's only one piece of edge data, so it is a sequence of one tuple.

```
In [11]: G = nx.read_edgelist(fname, create_using=nx.Graph, nodetype=int, data=[("weight", float)])
```

Some graph properties:

```
In [12]: G.order()
```

```
Out[12]: 14
```

```
In [13]: G.size()
```

```
Out[13]: 56
```

Confirm that the graph is connected, ie no isolated villages:

```
In [14]: nx.is_connected(G)
```

```
Out[14]: True
```

Kruskal's Minimum Spanning Tree algorithm finds an MST, that is a tree with the same nodes as the original, and a subset of the edges, such that in the tree all nodes are connected, and the sum of edge-weights in the tree is as small as possible. This solves the problem of building the electricity network because the cost of constructing an edge is proportional to the edge distance.

```
In [21]: mst = nx.minimum_spanning_tree(G)
```

```
In [24]: # how many edges? a tree always has n-1
mst.size()
```

```
Out[24]: 13
```

```
In [25]: # What is the total cost of constructing the network?
sum(mst[e0][e1]['weight'] for e0, e1 in mst.edges())
```

```
Out[25]: 4.423083999999999
```

Dijkstra's algorithm is a famous algorithm for finding the shortest path from a node to all other nodes. Here "shortest path" takes account of edge weights. In our case, the edge weight is a measure of the distance. We have to say which node we want to start from (0) and which edge property to use ("weight").

```
In [26]: node_dists, paths = nx.single_source_dijkstra(G, 0,
                                                    weight="weight")
```

Now we just find the most distant village:

```
In [34]: farthest = max(node_dists, key=lambda n: node_dists[n])
```

```
In [35]: farthest
```

```
Out[35]: 12
```

```
In [36]: node_dists[farthest]
```

```
Out[36]: 1.0830950000000001
```

By the way, the Stanford Large Network Dataset collection <https://snap.stanford.edu/data/> has lots of interesting graphs for

Version control, git, and GitHub

Has this ever happened to you?

```
$ ls  
assignment.py  
assignment2.py  
assignment_from_Tom.py  
assignment2_from_Tom.py  
assignment2_from_Tom2.py  
assignment_final.py  
assignment_FINAL_with_Toms_changes_merged_except_spelling.py  
assignment_final_5pm_missing_some_changes.py
```

A *version control* system is a system for controlling multiple versions of a file. There can be multiple collaborators working remotely, and multiple versions, with automatic merging of changes.

Version control also serves as a *history*, a *backup*, and (if you want) a *public repository*.

Diff and patch

Version control usually works on plain text files. E.g. a `.txt` file, or a `.py` or `.java` file is plain text. Word `.docx` files are not, but people working on documents (as opposed to program code) might write them in Markdown or LaTeX, which are plain-text formats that convert into `pdf`s or `html`.

A fundamental concept is the `diff`. A diff is a line-by-line list of differences between two plain text files.

`file_a.py`

```
x = 3  
a = "elephant"  
print("hello")
```

`file_b.py`

```
x = 3  
a = "lion"  
print("hello")
```

The `diff` between the two files would look like this:

```
2c2  
< a = "elephant"  
---  
> a = "lion"
```

There is a program `diff` which produces the `diff` as seen above. If you're on Linux or OSX, you should be able to run the above command. Inside a Jupyter Notebook, shell commands can then be run using a `!` prefix as above. On Windows you might have to use gitbash which is part of the main git download from https://git-scm.com/download/win.`

There is also a program `patch` which can take `file_a.py` and *apply* the diff, to produce `file_b.py`.

This is useful, because it allows two people to start on a common file, work *independently*, and then merge their changes whenever they want. A version control system uses `diff` and `patch` internally (that's why we didn't show how to do it manually, above).

In fact, `diff` and `patch` together give rise to an *algebra* of versions -- the theory that underlies version control.



Common version control systems

There are several version control systems in common use, including

- Subversion
- Git
- Mercurial
- Bitkeeper (\$)
- Perforce (\$)

There are several places online where you can use free online version control services. E.g. for Subversion there is <https://riouxsvn.com/>. For git there is <http://www.github.com/>. We will concentrate on git and Github. Remember, git is the name of the version control system, and Github is a company which provides a nice website with free (and pay-for) git hosting. A lot of people working in software and analytics use their Github account as a CV.

Sadly, git and Github are really complex. Happily, we can avoid most of the complexity. We just need to know a few simple things:

- How to create a new repository on Github
- How to clone the repository from Github to our disk
- How to add a new file to our local copy
- How to push from our local copy to Github
- A little about branches, merges, and merge conflicts.

Next, we'll carry out these basic tasks in a live Github repo. Don't worry, you can delete the repo after.

Before proceeding, make an account on Github, and log in to it.

Creating a new repository



- Click "New", then enter a name (no spaces or weird punctuation), e.g. "test", and a description, e.g. "My first test repository"
- Then choose "public" and tick "Initialize this repository with a README".

Now the repository will be created, and a `README.md` file will be created inside it. You can look at the list of files in the repository and get the clone URL of the repository (needed for the next step). You can also download the entire thing as a zip, but we won't normally proceed that way.



Cloning

To get the new repository onto your local disk, you `clone`. Type the following at your command prompt. Obviously, put in the appropriate clone URL for your new repository in place of the one I have mentioned.

```
git clone https://github.com/jmmcd/ML-snippets.git
```

You'll get a new directory, in your current directory, containing one file: `README.md`. Open it up in a text editor and have a look.

Adding a new file

Now, let's write a new Python program, say `test.py`, and save it in the same directory as `README.md`. We have to tell Git that it exists, and `commit` it.

```
git add test.py  
git commit -m "Wrote a simple test program."
```

Please tell me who you are

If we see the message `Please tell me who you are`, it's because git needs to associate every commit with the person who made it. It helpfully tells us what to do:

```
git config --global user.email "you@example.com"  
git config --global user.name "Your Name"
```

What does commit mean?

When you commit, you're saying the current version of the code is in a consistent state (i.e. no half-finished changes). It's not necessarily *complete* or perfect. Usually, you commit with messages like these:

- `git commit -m "Fixed a bug in calculation of y."`
- `git commit -m "Added a new function to print stats."`
- `git commit -m "Expanded documentation."`

In order to commit, you always `add` first, to tell `git` which files you want to commit.

Pushing

So far, we've added a new file and committed it, but that only affects our local (on-disk) repository. Next we have to `push` to Github.

```
git push  
(You will be asked for your Github password.)
```

After this finishes, you can reload the web page to see your `test.py` has appeared on Github.

Changing and committing

Previously, we added a new file and then committed. Even if we `edit` an existing file, we still have to run `add` (it really means "add the file to an upcoming commit", rather than "add to the repository") and then `commit`.

So, try adding some text to the `README.md`, then `add`, `commit` with an appropriate message, and `push`.

Pulling

Suppose your colleague is working in the same repository. To check whether they've committed and pushed any changes, you can run:

```
In [ ]: git pull
```

It gets any changes from github and applies them to your local repository. If necessary it uses `merge` so that your colleague's changes and your own are merged together.

Typical (simple) workflow on a single repo

```
git pull # get any changes by others  
# edit test.py in text editor  
git add test.py # tell git that test.py will be committed  
git commit -m "Change to tigers" # commit  
git push # push changes to GitHub
```

Walk-through 1

1. On Github, make a new repository by clicking "New".
2. Initialise it with a `README`.
3. Clone to our hard disk.
4. Make a new file `test.py`.
5. Add, commit, push.
6. See the changes on Github.
7. Make a change directly on Github.
8. Pull that to hard disk.

Merge conflicts

If we are working with colleagues in a single repository, we may see *merge conflicts*. A merge conflict arises when two people push incompatible changes (e.g. I changed lion to elephant, and `commit` ted, and at the same time you changed lion to antelope, and `commit` ed, and then we both `push` ed).

In a merge conflict, we'll see a message like this from `git`:

```
git pull  
[...]  
CONFLICT (content): Merge conflict in test.py
```

`git` will also put some special markers in the conflicted file. They show the chunks of text that is coming from the remote repo (Github), and the text in our working copy (on disk):

```
<<<<  
a = "tiger"  
=====  
a = "elephant"  
>>>>
```

To solve this, we have to edit the file to decide which version is better, remove all the special markers `<<<` `==` `>>>`, and then save it, `add` and `commit` with an appropriate message.

Walk-through 1 (part 2)

1. Make a change directly on Github.
2. Make a conflicting change on hard disk.
3. Try to pull and observe problem.
4. Try to commit and pull, still observe problem.
5. Resolve merge conflict by editing the markers.
6. Add, commit, push and see result in Github.

Branching



Suppose your colleague is working in the same repository. To check whether they've committed and pushed any changes, you can run:

```
In [ ]: git pull
```

It gets any changes from github and applies them to your local repository. If necessary it uses `merge` so that your colleague's changes and your own are merged together.

Typical (simple) workflow on a single repo

```
git pull # get any changes by others  
# edit test.py in text editor  
git add test.py # tell git that test.py will be committed  
git commit -m "Created fab new function in test.py"  
# maybe keep working for a few hours, days or weeks and then  
git checkout master # switch to master branch  
git merge big_feature # merge the branch onto master  
git branch -d big_feature # delete branch  
git push # push to GitHub
```

Walk-through 1

1. On Github, make a new repository by clicking "New".
2. Initialise it with a `README`.
3. Clone to our hard disk.
4. Make a new file `test.py`.
5. Add, commit, push.
6. See the changes on Github.
7. Make a change directly on Github.
8. Pull that to hard disk.

Merge conflicts

If we are working with colleagues in a single repository, we may see *merge conflicts*. A merge conflict arises when two people push incompatible changes (e.g. I changed lion to elephant, and `commit` ted, and at the same time you changed lion to antelope, and `commit` ed, and then we both `push` ed).

In a merge conflict, we'll see a message like this from `git`:

```
git pull  
[...]  
CONFLICT (content): Merge conflict in test.py
```

`git` will also put some special markers in the conflicted file. They show the chunks of text that is coming from the remote repo (Github), and the text in our working copy (on disk):

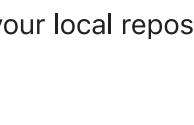
```
<<<<  
a = "tiger"  
=====  
a = "elephant"  
>>>>
```

To solve this, we have to edit the file to decide which version is better, remove all the special markers `<<<` `==` `>>>`, and then save it, `add` and `commit` with an appropriate message.

Walk-through 1 (part 2)

1. Make a change directly on Github.
2. Make a conflicting change on hard disk.
3. Try to pull and observe problem.
4. Try to commit and pull, still observe problem.
5. Resolve merge conflict by editing the markers.
6. Add, commit, push and see result in Github.

Branching



Suppose your colleague is working in the same repository. To check whether they've committed and pushed any changes, you can run:

```
In [ ]: git pull
```

It gets any changes from github and applies them to your local repository. If necessary it uses `merge` so that your colleague's changes and your own are merged together.

Typical (simple) workflow on a single repo

```
git pull # get any changes by others  
# edit test.py in text editor  
git add test.py # tell git that test.py will be committed  
git commit -m "Created fab new function in test.py"  
# maybe keep working for a few hours, days or weeks and then  
git checkout master # switch to master branch  
git merge big_feature # merge the branch onto master  
git branch -d big_feature # delete branch  
git push # push to GitHub
```

Walk-through 1

1. On Github, make a new repository by clicking "New".
2. Initialise it with a `README`.
3. Clone to our hard disk.
4. Make a new file `test.py`.
5. Add, commit, push.
6. See the changes on Github.
7. Make a change directly on Github.
8. Pull that to hard disk.

Merge conflicts

If we are working with colleagues in a single repository, we may see *merge conflicts*. A merge conflict arises when two people push incompatible changes (e.g. I changed lion to elephant, and `commit` ted, and at the same time you changed lion to antelope, and `commit` ed, and then we both `push` ed).

In a merge conflict, we'll see a message like this from `git`:

```
git pull  
[...]  
CONFLICT (content): Merge conflict in test.py
```

`git` will also put some special markers in the conflicted file. They show the chunks of text that is coming from the remote repo (Github), and the text in our working copy (on disk):

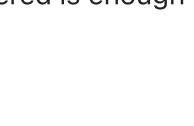
```
<<<<  
a = "tiger"  
=====  
a = "elephant"  
>>>>
```

To solve this, we have to edit the file to decide which version is better, remove all the special markers `<<<` `==` `>>>`, and then save it, `add` and `commit` with an appropriate message.

Walk-through 1 (part 2)

1. Make a change directly on Github.
2. Make a conflicting change on hard disk.
3. Try to pull and observe problem.
4. Try to commit and pull, still observe problem.
5. Resolve merge conflict by editing the markers.
6. Add, commit, push and see result in Github.

Branching



Suppose your colleague is working in the same repository. To check whether they've committed and pushed any changes, you can run:

```
In [ ]: git pull
```

It gets any changes from github and applies them to your local repository. If necessary it uses `merge` so that your colleague's changes and your own are merged together.

Typical (simple) workflow on a single repo

```
git pull # get any changes by others  
# edit test.py in text editor  
git add test.py # tell git that test.py will be committed  
git commit -m "Created fab new function in test.py"  
# maybe keep working for a few hours, days or weeks and then  
git checkout master # switch to master branch  
git merge big_feature # merge the branch onto master  
git branch -d big_feature # delete branch  
git push # push to GitHub
```

Walk-through 1

1. On Github, make a new repository by clicking "New".
2. Initialise it with a `README`.
3. Clone to our hard disk.
4. Make a new file `test.py`.</

7. Push to our Github.
8. Make a pull request. # DON'T DO THIS

Regular Expressions

In Computer Science, (formal) **language** is defined as a set of strings.

`L = {"b", "ab", "aab", "aaab"} is a language defined by enumeration.`

`L = {strings of length 1...4, starting with zero or more 'a's and ending in exactly 1 'b'} is the same language defined by a rule.`

`L = {valid email addresses according to RFC2822} is a language. https://tools.ietf.org/html/rfc2822`

`L = {valid Python files} is a language too, defined partly by Python docs and also, partly, perhaps implicitly by the behaviour of the Python interpreter.`

There are two main things we might want to do with languages.

- 1. *Recognize* strings, i.e. check whether a given string is in the language;

a. Just check, i.e. return True/False;

b. Check, and (assuming True), extract some of the structure or contents of the string;

- 2. *Generate* strings, i.e. generate 1 or more random strings from the language.

There are several different techniques suitable for *recognising* different types of languages:

- manual testing against explicit rules;
- regular expressions;
- grammars;
- finite state machines.

Recognising and extracting structure: some problems

- Validate user IDs, credit card numbers, post-codes, etc.
- Extract all email addresses from a text document
- Extract all the html tags from a html document
- Extract all the doctrings from Python code
- Check whether a URL is blacklisted
- Syntax-highlighting code
- Detecting repeated words in text, e.g. common typo "the the", "an an"
- Advanced find and replace mode in text editors/IDEs.

Regular expressions

In the 1960s and 1970s programmers realised that they were solving problems like this over and over, so they started to use regular expressions - a powerful, general method for matching strings against patterns, i.e. recognising (a certain type of) formal language.

Replace: manually-written, error-prone, wheel-reinventing code

→ a single formalism, a purpose-built mini-language, a single library implementation

"Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems." -- Jamie Zawinski

```
In [1]: # the module is just called re
import re
```

A quick example: detect all opening HTML tags

```
In [7]: p = "<[^/].*?>"
```

```
s = "<a href=test.com><font size=1>Some text</font></a>"
```

```
re.findall(p, s)
```

```
Out[7]: ['<a href=test.com>', '<font size=1>']
```

What we see: the `pattern` `p` is a string in a "domain-specific language" (a small language with its own, specialised syntax).

`s` is the target string to be matched.

Ways of using REs

- `re.match(p, s)`: check whether pattern `p` matches part of string `s` from the start
- `re.search(p, s)`: check whether `p` matches any part of `s`
- `re.findall(p, s)`: find all matches of `p` in `s`
- `re.split(p, s)`: like `str.split` but splits wherever `p` matches part of `s`
- `re.sub(p, r, s)`: for every match `p` in `s`, replace by `r` (`r` could be a string or function).

Raw strings

We previously saw f-strings `f""` which are useful for formatted output:

```
f"{} = {}"
```

A raw string uses a similar syntax `r""`. Inside a raw string, backslash-escapes such as `\n` and `\t` are not processed, they just remain as-is.

```
In [10]: print("not raw", "a\b\tc\n\de")
```

```
print("raw", "a\b\tc\n\de")
```

```
not raw a      b      c
```

```
\d\ea
```

```
raw a\b\tc\n\de
```

This is handy when writing RE patterns, because they often contain backslashes anyway. In a normal string, to write a backslash, we'd need to write a double-backslash. It would get confusing.

RE syntax

- Most characters just match themselves
- Wildcard `.` matches any one character
- Repetition markers `?`, `*`, `+`, `{n}`
- Match one of several alternatives marked with `|`
- Abbreviations like `\d`, `\s`, `\w` for character sets
- Custom sets like `[0-9a-f]` for hexadecimal
- Non-greedy marker `?`
- Backslash-escapes for special chars, e.g. `\.`
- Match beginning `^` or end `$`

(In the video presentation of this notebook, we will now go to <https://regex101.com/>, an amazing resource for learning, writing, and debugging REs.)

Literals, wildcards, and repetition

```
In [49]: s = "ax abx abbx abbbx"
```

```
In [56]: p = "ab"
```

```
re.findall(p, s)
```

```
Out[56]: ['ab', 'ab', 'ab', 'ab']
```

```
In [57]: p = "ab?x"
```

```
re.findall(p, s)
```

```
Out[57]: ['ax', 'abx']
```

```
In [49]: s = "ax abx abbx abbbx abbbbx"
```

```
In [59]: p = "ab*x"
```

```
re.findall(p, s)
```

```
Out[59]: ['ax', 'abx', 'abbx', 'abbbx', 'abbbbx']
```

```
In [58]: p = "ab*x"
```

```
re.findall(p, s)
```

```
Out[58]: ['abx', 'abbx', 'abbx', 'abbbx']
```

```
In [49]: s = "ax abx abbx abbbx abbbbx"
```

```
In [60]: p = "a.x"
```

```
re.findall(p, s)
```

```
Out[60]: ['abx']
```

```
In [61]: p = "a.*x"
```

```
re.findall(p, s)
```

```
Out[61]: ['ax abx abbx abbbx abbbbx']
```

```
In [49]: s = "ax abx abbx abbbx abbbbx"
```

```
In [63]: p = "ab{4}x"
```

```
re.findall(p, s)
```

```
Out[63]: ['abbbbx']
```

Custom classes

It is common to want to match any character from a particular set. We do that using `[]`.

```
In [4]: p = r"[-=/*+]"
```

```
s = "x += 5 + 3 / 17"
```

```
re.findall(p, s)
```

```
Out[4]: ['+=', '+', '/']
```

The syntax `a-z` or similar inside the `[]` means to take `a`, `z`, and all characters between. It works for `A-Z` and `0-9` also.

(In the previous example, we saw `-` inside `[]` but not inside a range like `a-z`.)

```
In [66]: p = r"[a-zA-Z]+"
```

```
s = "9080true6030false"
```

```
re.findall(p, s)
```

```
Out[66]: ['true', 'false']
```

The character `^` as the first character inside `[]` means not, i.e. we now match anything other than what is inside `[]`.

```
In [12]: p = "[^<^/]*?" # reject closing tags </
```

```
s = "<a href=test.com><font size=1>Some text</font></a>"
```

```
re.findall(p, s)
```

```
Out[12]: ['<a href=test.com>', '<font size=1>']
```

Example: hexadecimal

As we know, a hexadecimal digit is a digit 0-9 or letter a-f (or A-F). Here is an RE to match 1 or more hexadecimal digits.

```
In [77]: p = "[0-9a-fA-F]{1}"
```

```
s = "ACDC1979, Bach1741, ABBA1980"
```

```
re.findall(p, s)
```

```
Out[77]: ['ACDC1979', 'Bach', '1741', 'ABBA1980']
```

Abbreviations for common classes

- `\d`: any digit, i.e. `[0-9]`
- `\D`: any nondigit, i.e. `[^0-9]`
- `\s`: any whitespace
- `\S`: any non-whitespace
- `\w`: alphanumeric (including `_`) suitable for use in variable names;
- `\W`: non-alphanumeric

Exercise: Strong Password Detection. Given a password as a string, say how strong it is. Give it "+20% strength" for each:

- Longer than 8 characters
- Contains lower-case
- Contains upper-case
- Contains one or more digits
- Contains one or more non-alphanumeric characters

```
In [114]: import re
```

```
def strong_pw(s):
```

```
    strength = 0
```

```
    if len(s) > 8:
```

```
        strength += 20
```

```
    if re.search(r"\d", s) is not None:
```

```
        strength += 20
```

```
    if re.search(r"\w", s) is not None:
```

```
        strength += 20
```

```
    if re.search(r"\W", s) is not None or "_" in s:
```

```
        strength += 20
```

```
    return strength
```

```
strong_pw("MyTeStPasswOrd")
```

```
Out[114]: 100
```

Anchors `^` and `$`

These characters "anchor" the match to the start and the end of a line respectively.

```
In [91]: s = "a b c"
```

```
p = "r'b"
```

```
re.findall(p, s)
```

```
Out[91]: ['b']
```

```
In [92]: s = "a b c"
```

```
p = "r'^b"
```

```
re.findall(p, s)
```

```
Out[92]: []
```

Example: valid variable names

Which of these are valid variable names in Python?

```
In [83]: ss = ["x", "x17", "n_chars", "n_chars", "_", "_iters", "17x"]
```

The first character must be a letter or underscore, and after that letters, digits and underscores are allowed.

```
In [97]: ss = ["x", "x17", "n_chars", "n_chars", "_", "_iters", "17x"]
```

```
p = r"^\w+\w*$"
```

```
for s in ss:
```

```
    print(repr(s), ":", re.match(p, s))
```

```
Out[97]: ['x', 'x17', 'n_chars', 'n_chars', '_', '_iters', '17x']
```

```
'x': <re.Match object; span=(0, 1), match='x'>
```

```
'x17': <re.Match object; span=(0, 3), match='x17'>
```

```
'n_chars': <None>
```

```
'n_chars': <None>
```

```
'_iters': <None>
```

```
'_iters': <None>
```

```
'17x': <None>
```

<pre

eval and exec

If we have a string `s = "5"` and we type `i = int(s)`, it converts `s` to `int`.

If we have a string `s = "5.0"` and we type `x = float(s)`, it converts `s` to `float`.

But if we're **at the command line** (IPython or Jupyter notebook) and we just type `i = 5`, "it just works" -- `i` gets the right type without any special conversion step.

Since Python knows how to do this, we might ask: why can't we get Python to convert *any* string to **whatever value and type it would have been**, if we had just typed that string as input on the command-line?

That's what `eval` does. If `s = "5"`, then `eval(s)` is the same as `int(s)`. If `s = "5.0"`, then `eval(s)` is the same as `float(s)`. If `s` is a complex expression then `eval(s)` will evaluate `s` just as if it had been typed at the command-line.

In fact, a command-line (not only in Python) is often called a *read-eval-print* loop for this reason.

An eval application

Remember we mentioned when studying Scikit-Learn that when we create a logistic regression `lr = LogisticRegression()`, that object has a string representation like this:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                    intercept_scaling=1, l1_ratio=None, max_iter=100,
                    multi_class='warn', n_jobs=None, penalty='l2',
                    random_state=None, solver='warn', tol=0.0001, verbose=0,
                    warm_start=False)
```

That string has a nice property: we could copy and paste it into IPython or Jupyter Notebook to re-create the object: `lr = <copy and paste here!>`

That doesn't seem much more useful than just typing `lr = LogisticRegression()`.

But if we had set many parameters with non-default values it could be very useful:

```
lr = LogisticRegression(C=10.0, solver='lbfgs', max_iter=1000)
```

Then just typing `lr` will give us a nice, simple textual representation of the model we created. We could save it and later copy-and-paste it into IPython to recreate the model with the same hyperparameters.

However, what if we wanted to do this for many models in a loop with different hyperparameters? In other words we have created many models and saved their text representations, and later want to do something else with them. We can't just copy-and-paste them in a loop.

But that's where `eval` comes in:

```
for fname in fnames:
    s = open(fname).read()
    lr = eval(s)
```

Python ~ JSON

`eval` works well with all sorts of Python *literals*, e.g. `int`'s, `float`'s, `str`'s, `{}` for dictionaries, `[]` for lists, and so on.

Therefore, we might save a Python object to disk just by taking the string representation, and then load it back in later, effectively treating Python as if it was JSON. This can be very useful in **logging** from a long-running program.

```
In [9]: x = {"a": [3, 4, 5], "b": [5, 6, 7]}
open("data/eval_test.txt", "w").write(repr(x))
y = eval(open("data/eval_test.txt").read())
y
```

```
Out[9]: {'a': [3, 4, 5], 'b': [5, 6, 7]}
```

(There is also the `json` module in the standard library if we want to deal with JSON properly.)

`__str__` and `__repr__`

Notice that above we used `repr(x)`, not `str(x)`.

- `__str__` is supposed to give a descriptive string for human consumption;
- `__repr__` is supposed to give a string which can usually be `eval`-ed to recreate the original object.

For many classes, including `dict`, these are just the same thing. If an object `c` has `__repr__` but not `__str__`, then `str(c)` will default to calling `c.__repr__()`.

Security with eval

`eval` is dangerous, because it executes a piece of code. If you write a program with `eval` and provide the input, e.g. from a file you saved earlier, that's fine. If you write a program with `eval` and accept input from anonymous users over the internet, you might have some problems.

exec

As we have seen, `eval` allows us to evaluate a Python **expression**.

An expression in Python that has a single value, like a generalisation of a **formula** in maths. It can have arithmetic, Boolean operators, function calls, object constructors, and so on. It does not have assignment statements, conditionals, loops, function definitions (except maybe `lambda`).

But sometimes, we might have Python code `s` which is not a single expression. In such a case, `eval` doesn't work. It expects a single expression only:

```
In [3]: s = """
def f(x):
    if x > 3:
        print("the result is excessive")
for i in range(5):
    f(i)
"""

c = eval(s)
```

```
Traceback (most recent call last):

File "/home/jmmcd/anaconda3/lib/python3.7/site-packages/IPython/core/interactiveshell.py", line 3325, in r
un_code
    exec(code_obj, self.user_global_ns, self.user_ns)

File "<ipython-input-3-2583246614cb>", line 8, in <module>
    c = eval(s)

File "<string>", line 2
    def f(x):
        ^
SyntaxError: invalid syntax
```

There is another function, `exec`, which **executes** arbitrary Python code.

```
In [4]: exec(s)
```

```
the result is excessive
```

An arbitrary piece of code doesn't have a value, in contrast to an expression. So, `exec` doesn't return a value. (There is a simple hack to access values calculated inside `exec`, but we won't go any further with this.)

Everything we said about `eval` security applies even more so to `exec`, because it has even greater flexibility!

Exercise

- Can you think of a Python object which is *not* correctly and fully re-created if we write out its `repr` and then read it in and `eval` the result?

Solution

- E.g. in Scikit-Learn we can recreate a `LogisticRegression` in this way, but if we have `fit()` ted then the internal

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js . Use `pickle` or similar instead.

Grammars

A **grammar** is a computational device which can:

- generate strings from a given formal language, or
- parse strings to check that they come from a given language.

There are a few different types of grammar, but we will only study the **context-free grammar**, written in the **Backus-Naur Form (BNF)** notation.

A grammar is a set of *rewrite rules*. Here is an example in BNF:

```
In [15]: fname = "code/boolean.bnf"
print(open(fname).read())
<expr> ::= (<expr> <biop> <expr>) | <uop> <expr> | <var> | <const>
<biop> ::= and | or
<uop> ::= not
<var> ::= x[0] | x[1] | x[2]
<const> ::= True | False
```

Each rule has a left-hand side and a right-hand side separated by the symbol `::=` interpreted as *rewrites to*. The LHS is a single string enclosed in angle brackets. The RHS consists of one or more *productions*, separated by the pipe symbol `|`, interpreted as "or".

A *non-terminal* is a string enclosed in angle brackets. A *terminal* is any other string.

One of the non-terminals is the *start symbol*. In our convention, the non-terminal of the first rule is the start symbol.

Parsing a string

Parsing a string means starting with a string, and iteratively finding out what rule could have given rise to it, until either:

- We find the complete structure -- a *parsing* -- of the string, showing that it is a valid string in the language; or
- We find that the string is not in the language.

Compilers and interpreters use grammars in this way, e.g. the Python interpreter **parses** your code using a grammar. You can see the Python grammar here: <https://docs.python.org/3/reference/grammar.html>.

Recall that a piece of Python code corresponds to an Abstract Syntax Tree, like the one shown here. When we say that the Python interpreter parses a piece of code according to the Python grammar, and **finds its structure**, what we mean is that it creates an AST.



Generating a string from a grammar

Going in the opposite direction also has many applications in AI (we'll see one shortly). By this we mean starting with a grammar and generating a random string from the language:

1. We start with a *derivation* (a string) consisting of just the start symbol;
2. We find the first (left-most) non-terminal in the derivation, and find the rule where that non-terminal is the LHS.
3. We choose one of the productions on the RHS of that rule, and in the derivation, replace the non-terminal with that production.
4. If there are now no non-terminals in the derivation, we stop, else go to 2.

For example (just making **random** choices):

```
1. <expr>
2. -> (<expr> <biop> <expr>)
3. -> (<var> <biop> <expr>)
4. -> (x[0] <biop> <expr>)
5. -> (x[0] and <expr>)
6. -> (x[0] and <const>)
7. -> (x[0] and False)
```

The result is a string in the language. We see that this grammar defines a language consisting of Boolean expressions in up to three variables.

Implementing a grammar for generating strings

The file `code/grammar.py` contains code for reading a grammar in `.bnf` form in, and generating strings from it.

```
In [5]: # this is just a hack so we can import from a file in the code directory.
# you don't need to understand this!
import sys
sys.path.insert(1, "code")

In [ ]: from grammar import Grammar # import from code/grammar.py

In [2]: fname = "code/boolean.bnf"
g = Grammar(file_name=fname)

In [3]: print(g) # notice repr gives a useful output!
```

Grammar(terminals=['x[1]', 'and', 'or', 'x[2]', '(', 'True', ')', 'not', ' ', 'False', 'x[0]', non_terminal_s=['<expr>', '<var>', '<const>', '<uop>', '<biop>'], rules={'<expr>': [['(', '<expr>', ')', '<biop>', ')', '<expr>', ')'], ['<uop>', ' ', '<expr>'], ['<var>'], ['<const>']], '<biop>': [['and'], ['or']], '<uop>': [['not']], '<var>': [['x[0]'], ['x[1]'], ['x[2]']], '<const>': [['True'], ['False']]], start_rule='<expr>')

In []:

We can now derive (generate) a string at random:

```
In [19]: for i in range(5):
    print(g.derive_string())

x[0]
True
False
(False and True)
(x[1] and False)
```

In fact, we can even turn a string into a piece of code, using `eval` as before, and then run it with appropriate arguments:

```
In [36]: s = g.derive_string()
print(s)
f = eval("lambda x: " + s)
f(False, True, False)

(not True and (not not False and x[1]))
```

Out[36]: False

Grammars in natural language

Grammars are used in natural language too. For example, this grammar can generate a few simple sentences.

```
<sentence>      ::= <noun_phrase> <verb_phrase>
<noun_phrase>  ::= <noun> | <determiner> <noun>
<determiner>   ::= the | a
<noun>          ::= lion | tiger | antelope
<verb_phrase>  ::= <trans_verb> <noun_phrase> | <intrans_verb>
<trans_verb>   ::= ate | saw | chased
<intrans_verb> ::= slept | rested
```

```
In [38]: g = Grammar(file_name="code/animal_sentences.bnf")
```

```
In [39]: for i in range(5):
    print(g.derive_string())

a lion saw antelope
a tiger slept
the lion chased the lion
lion saw a antelope
a lion slept
```

Randomness and `random.seed`

The `derive_string` generates a string at random. At each step it uses `random.choice(productions)` to choose a random production.

That is often useful, just as generating random numbers is often useful. But in a scientific context, we often want **replicability** as well. How can we reconcile these ideas?

The good news is that internally, the random number generator (RNG) is deterministic -- it is only **pseudo-random**. An arithmetical transform is applied to each number it outputs, to get the next. That transform is complicated but deterministic.

That allows us to achieve replicability. If we start twice from the same state, we'll get the same sequence of random numbers both times.

We can start from a fixed state by **setting the random seed**. The random seed is a number which is like the "start state" of the random number generator (RNG).

```
In [44]: random.seed(73)
for i in range(5):
    print(g.derive_string())

antelope slept
the tiger slept
lion rested
a antelope rested
lion slept
```

By the way, the `random` module has several other useful functions:

- `random.random()` - a random float in $[0, 1]$
- `random.randrange(a, b)` - a random int in $[a, b]$
- `random.choice(L)` - a random element of the list `L`
- `random.shuffle(L)` - shuffle the list `L` in-place

Generative art - by hand

Let's make a piece of generative art by inventing a formula $p(x, y)$ and simply plotting p over the unit square.

p should be purely numerical, e.g. $p(x, y) = \sin(20x)$.

We'll use `linspace`, `meshgrid`, and `imshow` (recall our notebook/video on fractals in an earlier week) to make the grid. Here we won't iterate a formula and calculate escape time or anything like that.

```
In [15]: import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
```

```
In [16]: n = 200
xs = np.linspace(0, 1, n)
ys = np.linspace(0, 1, n)
x, y = np.meshgrid(xs, ys)
```

```
In [17]: def p(x, y): return np.sin(20 * x) * np.sin(30 * (y+0.5)) * x * y
plt.axis('off'); plt.imshow(p(x, y), cmap=cm.viridis);
```


Generative art - using grammars

"I'd rather write programs that write programs than write programs" -- Richard Sites

Notice that we can also write functions like `p(x, y)` as pure strings, using `eval` as follows:

```
In [18]: ps = "np.sin(40 * x) * np.sin(30 * (y+0.5)) * x * y"
p = eval("lambda x, y: " + ps)
plt.axis('off'); plt.imshow(p(x, y));
```


Exercise

Make a new grammar for *arithmetical expressions* in x and y . Start by copying `code/boolean.bnf` to a new file `code/arithmetic.bnf`. Generate random expressions from that, turn them into functions using `eval`, and visualise them. Try out different colormaps with `cmap`. Think of some more interesting primitive functions to use other than `*`, `sin`, and friends.

You can even create more variables (beyond x and y), e.g. polar coordinates r and θ .

Solution

Make a new grammar for *arithmetical expressions* in x and y . Start by copying `code/boolean.bnf` to a new file `code/arithmetic.bnf`. Generate random expressions from that, turn them into functions using `eval`, and visualise them. Try out different colormaps with `cmap`. Think of some more interesting primitive functions to use other than `*`, `sin`, and friends.

You can even create more variables (beyond x and y), e.g. polar coordinates r and θ .

The file `code/arithmetic.bnf` file is supplied with this week's materials.

Introspection

Introspection means looking inward to the self. In programming, it refers to a few ways in which a program can inspect its own properties **while running** (as distinct from **at compile-time**).

- Finding out the type of an object;
- Accessing docstrings, function names, and other function properties;
- Accessing filenames and line numbers of the source file;
- Accessing the call stack (what functions called what functions to lead us to *here*);
- Accessing the configuration of the interpreter.

A lot of people think that introspection is a key ingredient in consciousness and intelligence! So it's relevant to AI in this sense. But we will think of introspection as a handy tool for programming in general.

In C and other older languages, even detecting the length of an array at runtime was not possible. So to a C programmer, Python's `len(a)` would seem like introspection.

Given an arbitrary object, there are several easy ways to find out about it -- some we've seen already -- and these are more like true introspection:

- `type` : what type is it?
- `dir` : what methods does it have?
- `repr` : see a useful string representation, may be valid Python code.
- `help` : access the docstring.

Some new ones, quite self-explanatory:

- `callable(f)` : is `f` callable (i.e. is `f(...)` allowed)?
- `issubclass(D, C)` : is `D` a subclass of `C`? (Works on classes, not objects.)
- `isinstance(c, C)` : check if `c` is an instance of `C` (allowing for subclasses).

```
In [2]: def f(): pass
x = 17
class C:
    def __call__(self):
        return "Objects can quack like functions"
c = C()
print(callable(f))
print(callable(x))
print(callable(C))
print(callable(c))
c()
```

```
True
False
True
True
```

```
Out[2]: 'Objects can quack like functions'
```

```
In [2]: class C: pass
class D(C): pass
class E(D): pass
print(issubclass(D, C))
print(issubclass(E, C))
print(issubclass(C, D))
```

```
True
True
False
```

```
In [68]: c = C()
d = D()
print(isinstance(c, C))
print(isinstance(d, D))
print(isinstance(c, D))
print(isinstance(d, C))
```

```
True
True
False
True
```

Classes and functions have official names. The official name is the one written in the source following `class` or `def`. This is distinct from the object's variable name!

- `x.__name__` : get the "official" name of `x` which must be a function or a class

```
In [61]: def f(a, b):
    return a + b
g = f # g is f, and has the same official name:
print(f.__name__)
print(g.__name__)
```

```
f
f
```

This could be useful e.g. when testing some Scikit-Learn models:

```
In [62]: from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier

for clf in [LogisticRegression(solver="lbfgs"),
            KNeighborsClassifier(n_neighbors=1)]:
    s = clf.fit([[3, 4], [3, 5]], [0, 1]).score([[4, 4]], [0])
    print(clf.__class__.__name__, s)
```

```
LogisticRegression 1.0
KNeighborsClassifier 1.0
```

Accessing variables and values in a namespace

Each returns a `dict`:

- `globals()` : variables in the module namespace
- `locals()` : variables in the current namespace
- `vars()` : same as `locals()`
- `vars(d)` : variables in namespace `d` where e.g. `d` is some module.

hasattr

"Frameworks use introspection frequently, to discover the capabilities of objects the user has passed; for example, "does this object's class have a `do_something()` method? If so, call the object's `do_something()`; otherwise call the `do_something_similar()` framework function with the object as an argument."

-- <http://archive.oreilly.com/oreillyschool/courses/Python4/Python4-09.html>.

`hasattr(obj, attr)` tells us whether an object has a particular attribute.

Using introspection and `hasattr`, we check that our code will work before running it.

```
In [13]: def check_initial(x):
    if hasattr(x, "startswith"):
        print(x.startswith("C"))
    else:
        print(x.__class__.__name__.startswith("C"))
check_initial("CDEFG")
```

```
True
```

In the **duck typing** style, we instead assume `x` has a (and maybe catch a possible exception). We say "it is easier to ask forgiveness than permission" (EAFP):

```
In [14]: def check_initial(x):
    try:
        print(x.startswith("C"))
    except AttributeError:
        print(x.__class__.__name__.startswith("C"))
check_initial(C())
```

```
True
```

Version numbers for Python and libraries

These are useful:

1. To catch possible incompatibilities between your code and an old version of a library;
2. When reporting bugs on Github and asking questions on Stackoverflow.

```
In [34]: import sys
print(sys.version)
import numpy as np
print(np.version)
import sklearn
print(sklearn.__version__)
```

```
3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0]
<module 'numpy.version' from '/home/jmmcd/anaconda3/lib/python3.7/site-packages/numpy/version.py'>
0.21.2
```

Some more possibilities

- Find out what source file an object was defined in, and at what line number (`inspect.getmembers()`);
- Find out which function called the function we are currently in (`inspect.getmembers()`);
- Given a function, find out what arguments (and types) it expects (`inspect.Signature`).

Example: getting at the data in a JSON file

Suppose we have a JSON file and we don't know anything about its structure. We can use the `json` module to read it in to native Python datatypes. We'll use introspection to understand the structure.

```
In [8]: import json
d = json.load(open("data/students.json"))
```

Notice that (just like `pickle.load` and `pickle.dump` which we saw before), `json.load` expects an open file, not a filename. Same for `json.dump`.

```
In [9]: d
```

```
Out[9]: [{"name": "Bruce Wayne",
  'age': 34,
  'ID': '1234',
  'modules': {'CT5123': {'grades': [55, 68],
    'attendance': [False, True, True, True, True]},
   'CT5234': {'grades': [45, 90],
    'attendance': [True, False, False, False, True]}},
  {"name": "Peter Parker",
  'age': 21,
  'ID': '0126',
  'modules': {'CT5123': {'grades': [90, 90, 90],
    'attendance': [False, True, True, True, True]},
   'CT5234': {'grades': [60, 74],
    'attendance': [False, True, True, True, True]}}}]
```

```
In [10]: type(d)
```

```
Out[10]: list
```

```
In [12]: type(d[0])
```

```
Out[12]: dict
```

```
In [13]: d[0].keys()
```

```
Out[13]: dict_keys(['name', 'age', 'ID', 'modules'])
```

```
In [14]: d[0]["name"]
```

```
'Bruce Wayne'
```

```
In [15]: d[0]["age"]
```

```
34
```

```
In [17]: d[0]["ID"]
```

```
'1234'
```

```
In [18]: d[0]["modules"]
```

```
Out[18]: {'CT5123': {'grades': [55, 68], 'attendance': [False, True, True, True, True]},
 'CT5234': {'grades': [45, 90], 'attendance': [True, False, False, False, True]}}
```

```
In [19]: type(d[0]["modules"])
```

```
Out[19]: dict
```

... and so on.

Downey provides a nice function `structshape` for doing this automatically and summarising the result. See example below and see Think Python, p. 120. The code is in `code/structshape.py`.

```
In [23]: from code.structshape import structshape
structshape(students)
```

```
Out[23]: 'list of (dict of 4 str->(int, dict of 2 str->dict of 2 str->(list of 2 int, list of 5 bool), str), dict of 4 str->(int, dict of 2 str->(dict of 2 str->(list of 3 int, list of 5 bool), dict of 2 str->(list of 2 int,
```

```
Loading [MathJax]/jax/output/CommonHTML/fonts/Tex/fontdata.js'
```


n-grams, randomness, and the Aaronson oracle

The Aaronson oracle is about the simplest possible computer game, but it really makes you think. We'll describe it and see how to get the computer to play it using a very broadly applicable technique, *n*-grams. We'll start with motivation.

Which of these is random?



From [telescopr](#) via [Discover magazine](#)

The left-hand image is random. The right-hand one is too evenly-spaced to be truly random.

It turns out that humans are very bad at detecting randomness. We're so good at detecting patterns that we detect them even when they're not there!

The same thing could happen in other settings. Which of these is random?

- 10011000110100011000111010100010110000110001111110
- 010010100101010001001001011010101101010110101100

The [Aaronson oracle](#) is a game invented by Scott Aaronson (a top researcher in computational complexity and quantum computing, and he has a [great blog](#)) to probe the idea of randomness.

- The human types '0' or '1' (or 'f' or 'd', or left-arrow or right-arrow, whatever) -- should be fairly quick.
- The computer has made a guess which it will be.
- We repeat many times (the human should type fairly quickly). The computer has access to the history.
- If the computer's accuracy is near 50%, the human wins. If high (e.g. 65%), the computer wins.

The program is just a loop which makes a prediction (let's just predict randomly for now), reads in the user's move, saves both to a history, and calculates a running accuracy.

```
move_hist = []
pred_hist = []
while True:
    # generate a prediction, just placeholder
    prediction = random.choice("01")

    # read user's move
    user_move = getch()

    # store to history
    move_hist.append(user_move)
    pred_hist.append(prediction)

    # stats
    accuracy = np.mean([p == u for p, u in
                        zip(pred_hist, move_hist)])
    print(f"prediction: {prediction}; " +
          f"user played {user_move}; " +
          f"accuracy {accuracy}")
```

input and getch

In Python, there is a function `input()` (a component of the read-eval-print loop!) which gets a line from the user.

But that won't work here because the user would have to type 0 `<return>` or 1 `<return>` each time which would be slow and could change their behaviour.

Instead, there is a small module `code/getchar.py` which reads one character at a time, available here:

- <https://stackoverflow.com/q/510357/86465>
- <http://code.activestate.com/recipes/134892/>

Our remaining task is to implement a method of predicting the user's move. There are many possible approaches, each of which makes some implicit assumptions about the patterns that will be present despite the user's intentions. We'll use a very simple method which performs amazingly well: *n*-grams.

- Suppose we observe that whenever the user types `01010` they usually type `1` next. That means that whenever we observe `01010` we should predict `1`.

n-grams

An *n*-gram is an *n*-tuple of consecutive objects drawn from a sequence. For example in the sentence "it was the best of times, it was the worst of times", we have the following 3-grams:

(it was the), (was the best), (the best of), (best of times), etc.

n-grams have applications in language modelling and generation, both for natural language and formal languages, and even the "language" of music.

Here, we will use *n*-grams to make predictions. Let's choose $n = 5$. For every possible 5-gram, we'll track how often it is followed by a `0` and how often by a `1`.

Suppose we have the string `010101001`. Then we get these observations:

```
01010 -> 1
10101 -> 0
01010 -> 0
10100 -> 1
```

We'll represent each observation as a 2-tuple, e.g. `("01010", "1")`. We'll just put the tuples in a `Counter`, which is a specialised `defaultdict`.

```
from collections import Counter
c = Counter()
c["01010", "1"] += 1
c["10101", "0"] += 1
```

etc.

Then whenever we observe a string `"01010"`, we just check which has the highest frequency in the data so far: `("01010", "0")`, or `("01010", "1")`.

Amazingly, this is good enough to perform quite well! The full program is in `code/aaronson_oracle.py`:

```
$ cd code
$ python aaronson_oracle.py
```

Exercises

- Play the game a few times and see how you get on. Remember, you're supposed to play quickly.
- Tell it to read the *n*-grams from a previously-saved file if it exists, and to write an updated version at the end, e.g. `python aaronson_oracle.py ngrams.txt`.
- Look at the Counter that is saved in `ngrams.txt` and see if you can see your own weakness.
- Change n and see if it gets better or worse.

Exercises (only for fun):

1. Write an ML program using Scikit-Learn which can perform better than the *n*-gram program, ie better than 60% against a typical human.

2. Write a deterministic player which can defeat the program, ie keep it close to 50%.

3. Try playing your deterministic player (2) against your friend's enhanced predictor (1).

4. Try playing against your friends manually.

The Fourier Transform and Spectrogram

Fourier: any signal is a sum of sinusoidal signals.

Helmholtz: any sound is a sum of sinusoidal tones.

Tangent: "Fourier's consideration of the possibility that the Earth's atmosphere might act as an insulator of some kind is widely recognized as the first proposal of what is now known as the greenhouse effect" (Wikipedia)

The Fourier transform is a core building-block of the study of signals. It decomposes a signal into a sum of sinusoidal signals of different frequencies. We can then see how much energy is present at each frequency.

<http://mriquestions.com/fourier-transform-ft.html>

The **Discrete** Fourier Transform is a definition of the FT, suitable for use with time series (i.e. signals which are sampled at discrete, regular intervals, rather than continuous).

The **Fast** Fourier Transform (Cooley & Tukey, 1965) computes the Discrete Fourier Transform, fast. It was named as one of the top 10 algorithms of the 20th century!

<https://www.computer.org/csdm/magazine/cs/2000/01/c1022/13rRUxBJhBm>

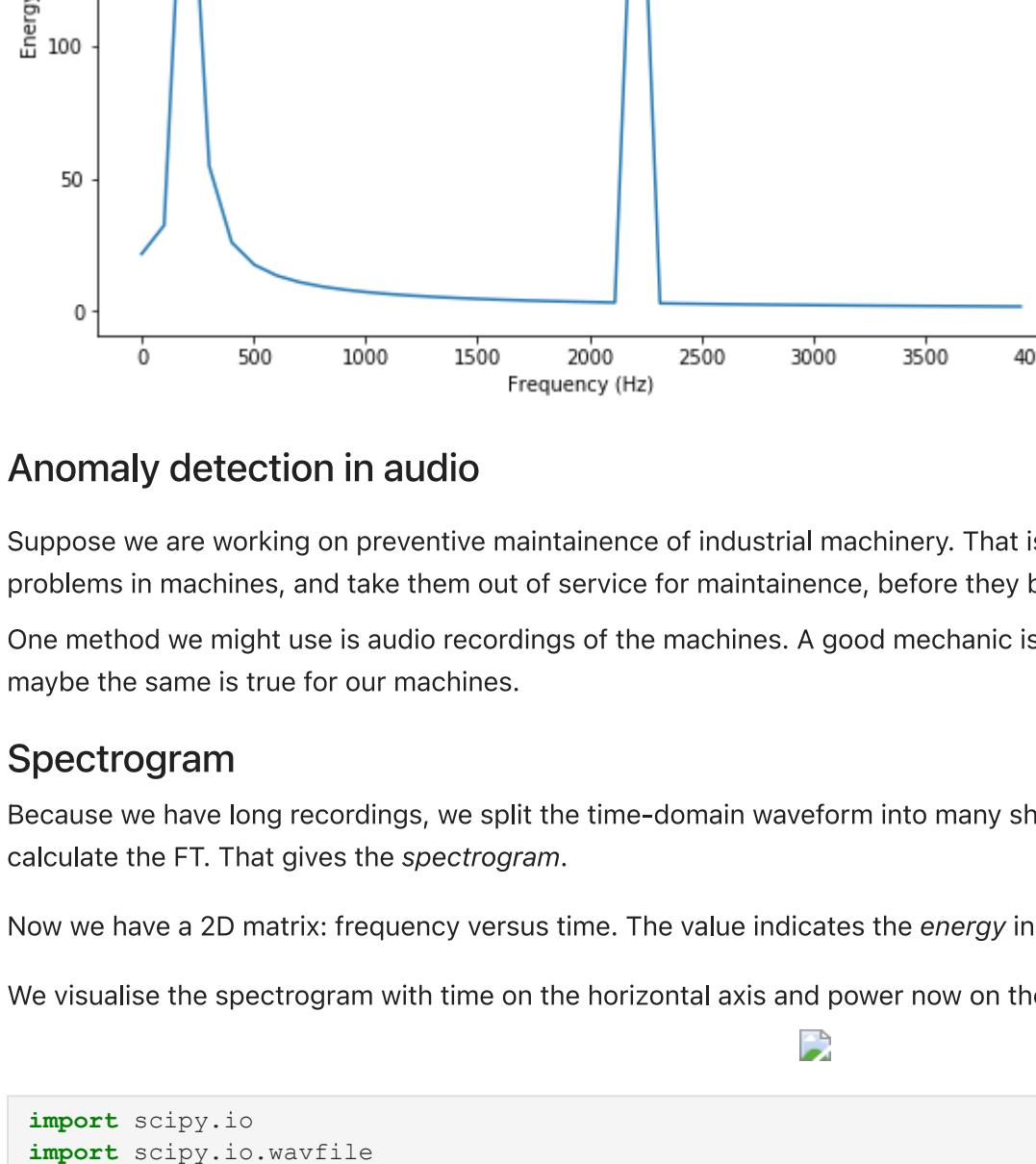
This amazing animation shows how the Fourier transform can arise in unexpected places: <http://www.jezzamon.com/fourier/>

First, we'll see how to run the FFT on a short time series of generated audio.

```
In [5]: import numpy as np
import matplotlib.pyplot as plt
import scipy.fftpack
%matplotlib inline
```

```
In [9]: L = 0.01 # 0.01s of audio
fs = 44100 # sampling rate
N = int(L * fs) # number of samples
# time values, linearly spaced
t = np.linspace(0, L, N, endpoint=False)
f1, f2 = 220.0, 2200.0
x = (np.sin(2 * np.pi * t * f1) +
     np.sin(2 * np.pi * t * f2))
```

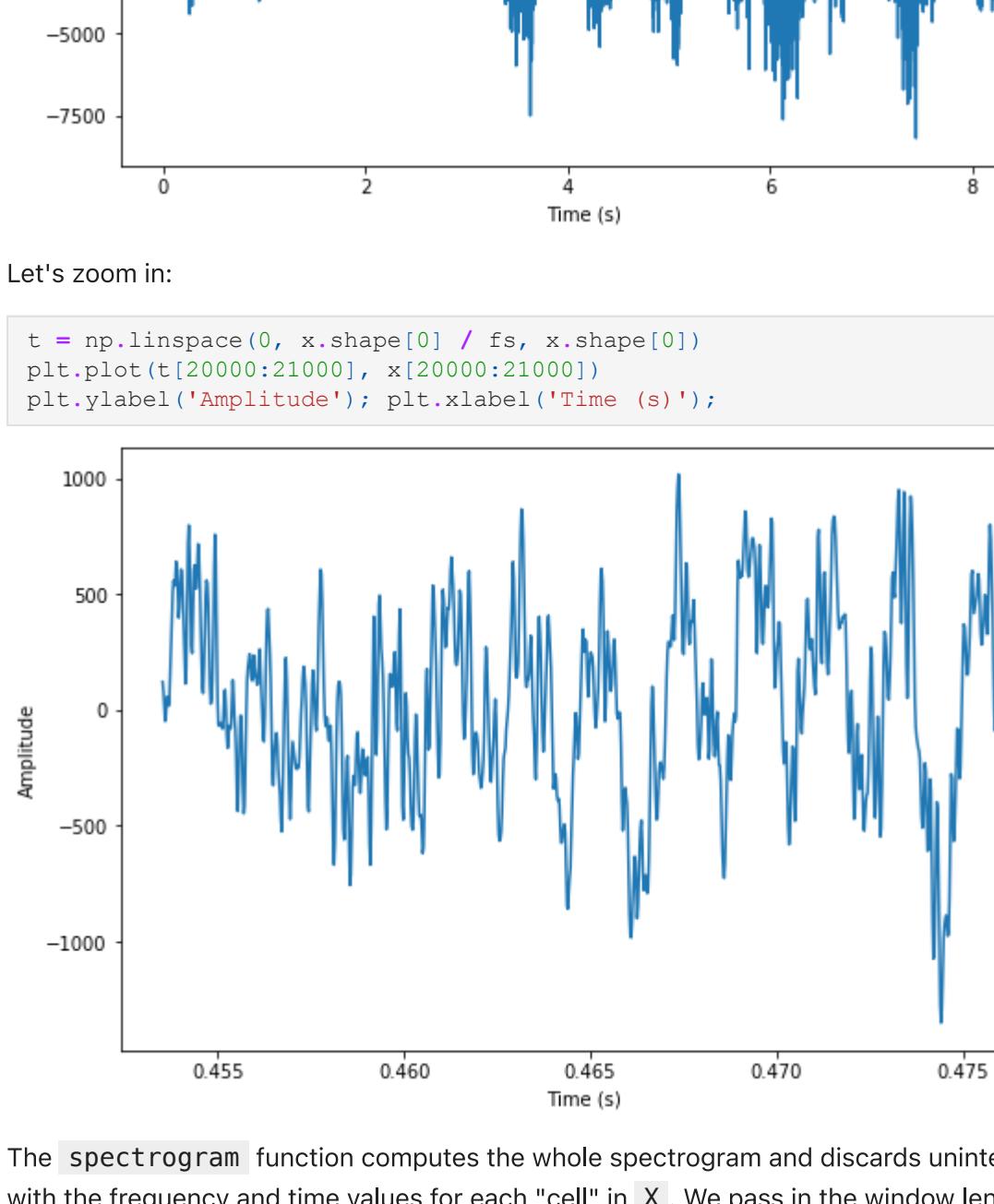
```
In [10]: plt.plot(t, x)
plt.xlabel("Time (s)")
plt.ylabel("x (waveform)");
```



```
In [12]: X = scipy.fftpack.fft(x) # FT
# discard the second half of FT with [:N//2]
# and discard the imaginary part (phase) with abs
X = np.abs(X[:N//2])
print(X.shape)
```

(220,)

```
In [13]: # calculate the frequency for each DFT bin
F = np.linspace(0.0, fs / 2, N//2)
# plot only the low-frequency components
plt.plot(F[:40], X[:40])
plt.xlabel(r"Frequency (Hz)"); plt.ylabel("Energy");
```



Anomaly detection in audio

Suppose we are working on preventive maintenance of industrial machinery. That is: we want to detect and predict possible problems in machines, and take them out of service for maintenance, before they break down.

One method we might use is audio recordings of the machines. A good mechanic is often able to diagnose car problems by ear, so maybe the same is true for our machines.

Spectrogram

Because we have long recordings, we split the time-domain waveform into many short (overlapping) windows and for each, calculate the FT. That gives the *spectrogram*.

Now we have a 2D matrix: frequency versus time. The value indicates the *energy* in each frequency bin, at each time-step.

We visualise the spectrogram with time on the horizontal axis and power now on the vertical axis.

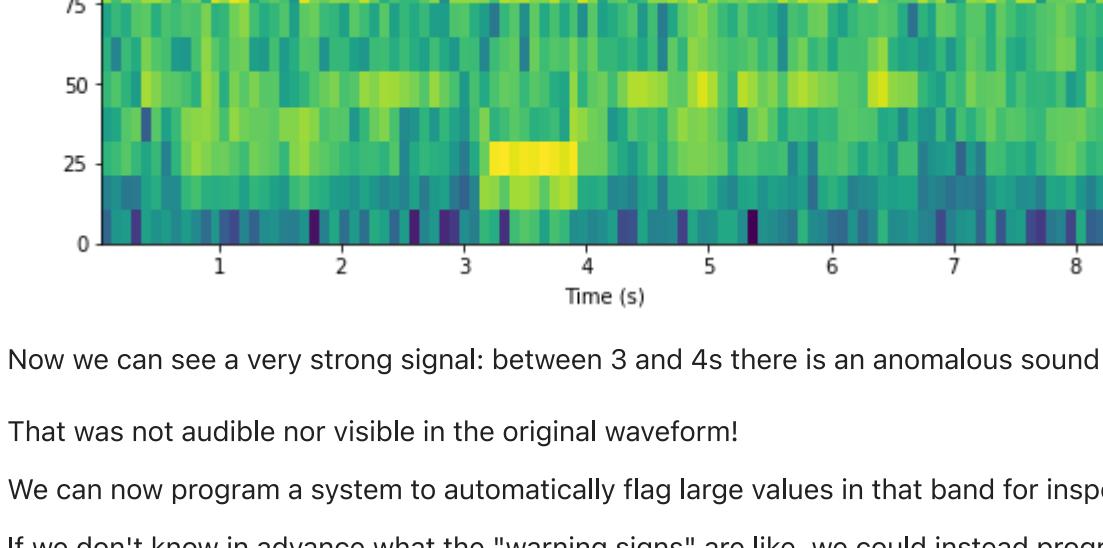
```
In [14]: import scipy.io
import scipy.io.wavfile
import scipy.signal
```

```
In [15]: fs, x = scipy.io.wavfile.read("data/furnace.wav")
print(fs) # sampling rate
print(x.shape)
```

44100
(367616,)

Let's look at the audio recording:

```
In [16]: t = np.linspace(0, x.shape[0] / fs, x.shape[0])
plt.plot(t, x)
plt.ylabel('Amplitude'); plt.xlabel('Time (s)');
```



Let's zoom in:

```
In [17]: t = np.linspace(0, x.shape[0] / fs, x.shape[0])
plt.plot(t[20000:21000], x[20000:21000])
plt.ylabel('Amplitude'); plt.xlabel('Time (s)');
```


The `spectrogram` function computes the whole spectrogram and discards uninteresting parts to give `X`. It gives `f` and `t` with the frequency and time values for each "cell" in `X`. We pass in the window length in samples (usually a power of two).

```
In [18]: f, t, X = scipy.signal.spectrogram(x, fs=fs,
                                         nperseg=4096)
print(f.shape)
print(t.shape)
print(X.shape)
```

(2049,)
(102,)
(2049, 102)

```
In [19]: plt.pcolormesh(t, f, np.log(X));
plt.ylabel('Frequency (Hz)');
plt.xlabel('Time (s)');
```


Now we can see a very strong signal: between 3 and 4s there is an anomalous sound around 25Hz not present elsewhere.

That was not audible nor visible in the original waveform!

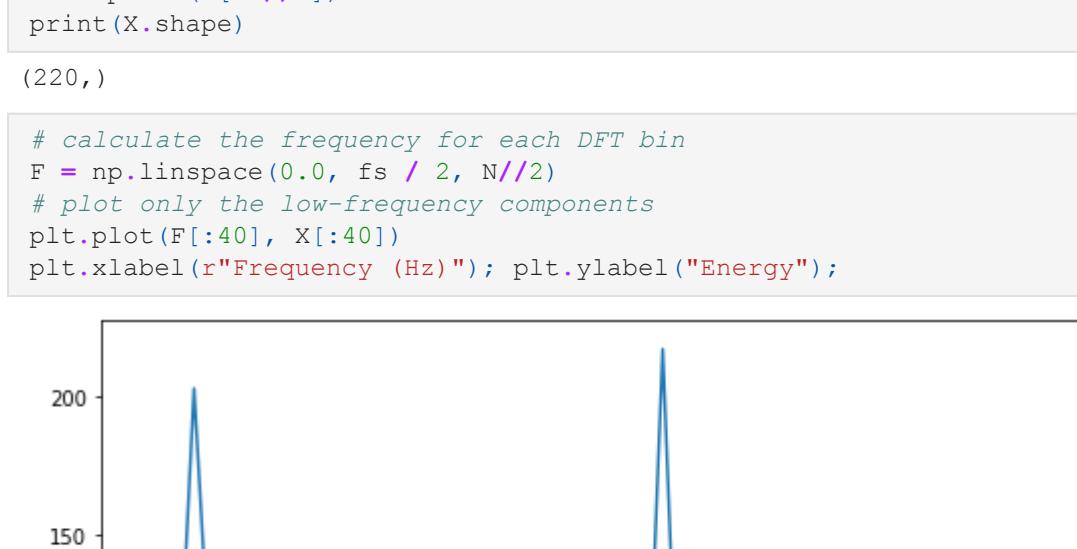
We can now program a system to automatically flag large values in that band for inspection and maintenance.

If we don't know in advance what the "warning signs" are like, we could instead program an ML algorithm to detect any type of anomaly.

What else can we use this for?

- The spectrogram contains more meaningful ("high-level") features than the raw data, so:
- Machine learning on audio (e.g. speech processing) usually starts with the spectrogram, eg:

Loading [MathJax]/jax/output/CommonHTML/fonts/Tex/fontdata.js in noisy environments) uses peaks in the spectrogram.



Now we can see a very strong signal: between 3 and 4s there is an anomalous sound around 25Hz not present elsewhere.

That was not audible nor visible in the original waveform!

We can now program a system to automatically flag large values in that band for inspection and maintenance.

If we don't know in advance what the "warning signs" are like, we could instead program an ML algorithm to detect any type of anomaly.

- Also non-audio signals: weather, stock markets, astronomy, ...

Introduction to R

James McDermott

NUI Galway



NUI Galway
OÉ Gaillimh

Programming and Tools for Artificial Intelligence

Dr James McDermott

Introduction to R

R is a language for statistical computing. It is based on an older, commercial language S. Like most of the software studied in this MSc, R is open-source. Research statisticians develop new algorithms in R because it is high-quality open-source. Professional data scientists use it because many statistical algorithms become available in R first, and because the ecosystem, especially tools like RStudio, R Markdown, ggplot, the tidyverse, and Shiny, are excellent.



R Ecosystem

- RStudio: a nice IDE for R
- R Markdown: a text-based format for writing reports with integrated R code, code outputs, and plots
- ggplot: best-in-class plotting
- The tidyverse: a collection of packages for manipulating data according to rational principles of “tidy data”
- Shiny: web-based dashboards

Sources

- Our R lessons are based partly on Hadley Wickham's *R for Data Science* <https://r4ds.had.co.nz>
- We also draw on Dr Jim Duggan's NUI Galway module CT474
- The materials are written in "R Markdown". I'll distribute both the .Rmd source and the .pdf slide output.
<https://rmarkdown.rstudio.com/lesson-1.html>

Further reading

- Venables, Smith and the R Core Team, *An Introduction to R*
<https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf>
- Wickham, *Advanced R* <https://adv-r.hadley.nz>
- Kabacoff, *Quick-R* <https://www.statmethods.net/>

Cheatsheets

- <https://rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf>
- <https://rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf>
- <https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>
- <https://rstudio.com/resources/cheatsheets/>

RStudio

“RStudio is an integrated development environment, or IDE, for R programming. Download and install it from <http://www.rstudio.com/download>.” - R4DS

R Basics

James McDermott

NUI Galway



NUI Galway
OÉ Gaillimh

Programming and Tools for Artificial Intelligence

Dr James McDermott

R Basics

Basic R

R **syntax** looks a bit different to Python. Many people think it's not as clean. But the fundamental **concepts** are mostly the same: line-by-line execution, primitive data types, compound data types, assignment, function calls, iteration and conditionals, classes.

Numerical data

```
x <- 5
typeof(x) # vector of length 1 of type double!
## [1] "double"

x <- 5L
typeof(x) # using 'L', a vector of length 1 of type int
## [1] "integer"
```

```
s <- "abcdefghijklmnopqrstuvwxyz" # assignment using "<-
x <- 5
while (x > 0) { # curly brackets
  if (x %% 2 == 0) { # operators may differ from Python
    print(c(x, substr(s, x, x))) # c() means concatenate
  }
  x = x - 1 # assignment using "="
}
## [1] "4" "d"
## [1] "2" "b"
```

Special values

- NA: not applicable/missing (common in data read from e.g. CSV files)
- NaN: “not a number”, as in Python
- -Inf, Inf: infinite values, as in Python

```
c(NA, -1, 0, 1) / 0
```

```
## [1] NA -Inf NaN Inf
```

Special functions

```
is.finite(), is.infinite(), is.na(), is.nan()
```

What's this about <- and =?

They usually do the same thing – assignment – but there are a few places where = is not allowed. The R community tends to stick to <-.

- <https://stackoverflow.com/questions/1741820/what-are-the-differences-between-and-in-r>

Vectors: a compound data type

```
x <- c("one", "two", "three", "four", "five")
x[1] # BTW R indexes from 1, not from 0
## [1] "one"
x[[1]]
## [1] "one"
```

What's the difference between [] and [[]]?

Both exist and sometimes do the same thing, sometimes different!

```
x <- c("one", "two", "three", "four", "five")
x[1] # BTW R indexes from 1, not from 0
## [1] "one"

x[[1]]
## [1] "one"

x[c(3, 2, 5)] # single [] for selecting a subset of elements
## [1] "three" "two"   "five"
# x[[c(3, 2, 5)]] # double [[]] doesn't work here
```

- <https://stackoverflow.com/questions/1169456/the-difference-between-bracket-and-double-bracket-for-accessing-the-el>
- <http://adv-r.had.co.nz/Subsetting.html>

Lists with named elements

A list with named elements is a bit like a dict:

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
a[["b"]] # double square brackets
## [1] "a string"
```

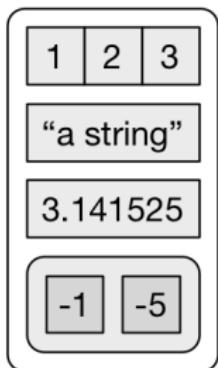
\$ is the same as [[]] but only for named elements:

```
a$b # notice, no quotation marks
## [1] "a string"
```

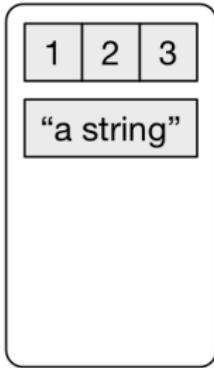
List subsetting

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```

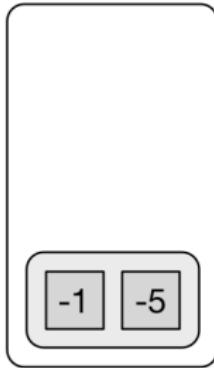
a



a[1:2]



a[4]



a[[4]]



a[[4]][1]



a[[4]][[1]]



(From R4DS)

List subsetting

See also Wickham's pepper pot: <https://r4ds.had.co.nz/vectors.html>
(Ctrl-F pepper)

Compound data types

- R vector -> Python list or Numpy ndarray
- R list -> Python tuple
- R data.frame -> Pandas DataFrame
- R tibble -> Pandas DataFrame

Inspecting compound data types

The `str` function gives you the *structure* of an item:

```
str(a)
```

```
## List of 4
## $ a: int [1:3] 1 2 3
## $ b: chr "a string"
## $ c: num 3.14
## $ d:List of 2
##   ..$ : num -1
##   ..$ : num -5
```

The `typeof` and `length` functions are self-explanatory:

```
typeof(a)
```

```
## [1] "list"
```

```
length(a)
```

```
## [1] 4
```

Ranges, columns, for-in

```
xs = 1:10 # range 1, 2, ... 10
print(xs)
## [1] 1 2 3 4 5 6 7 8 9 10

for (x in xs) {
  print(x^2 %% 2)
}

## [1] 1
## [1] 0
## [1] 1
## [1] 0
## [1] 1
## [1] 0
## [1] 1
## [1] 0
## [1] 1
## [1] 0
```

But don't use for-loops!

R is vectorised, like Numpy:

```
xs = 1:10
xs = xs ^ 2
ys = xs %% 2
ys
## [1] 1 0 1 0 1 0 1 0 1 0
```

Vectorised if-else

```
v1 = 1:5
v2 = v1 ^ 2
v3 = ifelse(v2 %% 2 == 0, "Even", "Odd")
v3
## [1] "Odd"  "Even" "Odd"  "Even" "Odd"
Compare to Numpy np.where.
```

Recycling

```
1:10 * 1:2
```

```
## [1] 1 4 3 8 5 12 7 16 9 20
```

The shorter vector is recycled. But this is ugly: don't use it!

Functions

function is the equivalent of Python lambda.

```
normalise <- function(x) {  
  # no "return" statement: last value is returned  
  (x - min(x)) / (max(x) - min(x))  
}  
normalise(1:10)  
## [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556  
## [8] 0.7777778 0.8888889 1.0000000
```

Exercises

- 1 Write the Factorial function in R, eg `fact(5)` gives 120.
- 2 Given `x <- "John"`, calculate the length in characters of `x`. Use `nchar()`.
- 3 Given `xs <- c("John", "Paul", "George", "Ringo")`, calculate the length of each name, using vectorisation (not a for-loop).
- 4 Calculate whether each name is shorter than 5 characters.
- 5 Index `xs` to keep just the names shorter than 5 characters.
- 6 Write a function which unit-norms a vector, ie normalises it so that the vector length equals 1. Eg `unit_norm(c(10, 10, 10, 10))` gives
0.5 0.5 0.5 0.5.
- 7 Write a function which standardises a vector, ie gets the z-score, ie maps it to have mean 0 and standard deviation 1. Eg `z_score(c(10, 6, 12, 12))` gives
0.0000000 -1.4142136 0.7071068
0.7071068.

Solutions

```
fact <- function(n) { # Exercise 1
  if (n <= 1) {
    1 # remember, no return statement!
  } else {
    n * fact(n-1)
  }
}
fact(5)
## [1] 120
```

```
x <- "John"  
nchar(x) # Exercise 2  
## [1] 4  
  
xs <- c("John", "Paul", "George", "Ringo")  
nchar(xs) # Exercise 3  
## [1] 4 4 6 5  
  
nchar(xs) < 5 # Exercise 4  
## [1] TRUE TRUE FALSE FALSE  
  
xs[nchar(xs) < 5] # Exercise 5  
## [1] "John" "Paul"
```

```
unit_norm <- function(x) { # Exercise 6
  x / sqrt(sum(x**2))
}
unit_norm(c(10, 10, 10, 10))
## [1] 0.5 0.5 0.5 0.5

z_score <- function(x) { # Exercise 7
  (x - mean(x)) / sd(x)
}
z_score(c(10, 6, 12, 12))
## [1] 0.0000000 -1.4142136  0.7071068  0.7071068
```

Tidy Data in R

James McDermott

NUI Galway



NUI Galway
OÉ Gaillimh

Programming and Tools for Artificial Intelligence

Dr James McDermott

Tidy Data in R

Tidy Data

- “Tidy Data” is a set of data-organising principles common in R.
- Proposed by Hadley Wickham, author of important R packages, recently won the 2019 COPSS President’s Award (“the Fields medal of statistics”).
- “when your data is tidy, each column is a variable, and each row is an observation”.
- Much of this and the following R notebooks draw on Wickham’s *R for Data Science* <https://r4ds.had.co.nz>.

Base R and the Tidyverse

- The “Tidyverse” is the “Tidy data universe”, a set of packages which try to adhere to tidy data principles.
- Tidyverse packages are all about working on *tibbles* (improved data frames), rather than vectors.
- The tidyverse packages provide new functionality and cleaner interfaces to existing functionality.
- “Base R” is the name used to mean R without the tidyverse packages.

Installing and using packages

To install the Tidyverse packages, copy this code to your R Console in RStudio.

```
> install.packages("tidyverse")
```

To actually use these packages, we have to import as follows:

```
library(tidyverse)

## -- Attaching packages -----
## v ggplot2 3.2.1     v purrr    0.3.2
## v tibble   2.1.3     v dplyr    0.8.3
## v tidyverse 1.0.0     v stringr  1.4.0
## v readr    1.3.1     vforcats  0.4.0
## -- Conflicts -----
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

tibbles are improved data.frames

- In Base R, a `data.frame` is (as in Pandas) a data structure for rectangular data, with column headers – a *table*
- A `tibble` is an improved dataframe for the tidyverse
- Docs: <https://r4ds.had.co.nz/tibbles.html>

Converting a data.frame to a tibble

```
d <- read.csv("data/mpg_extract.csv")
head(d, n=3) # just look at 3 rows

##   X..year miles.per.gallon
## 1      70              18
## 2      70              15
## 3      70              18

dt = as_tibble(d)
head(dt, n=3)

## # A tibble: 3 x 2
##   X..year miles.per.gallon
##       <dbl>             <dbl>
## 1      70              18
## 2      70              15
## 3      70              18
```

Why are tibbles named tibbles?



Making a tibble by hand: use tribble

```
d = tribble(  
  ~x, ~y, ~z, # here ~ indicates that these are formulas  
  #---/---/----  
  "a", 2, 3.6,  
  "b", 1, 8.5  
)
```

Reading in a tibble

`read_csv` (not `read.csv`) is part of the Tidyverse `readr` package.

```
d <- read_csv("data/prices.csv")  
  
## Parsed with column specification:  
## cols(  
##   Date = col_character(),  
##   AAPL = col_double(),  
##   MSFT = col_double(),  
##   YHOO = col_double(),  
##   NFLX = col_double(),  
##   BMW = col_double(),  
##   F = col_double(),  
##   FB = col_double(),  
##   GOOG = col_double(),  
##   GM = col_double(),  
##   GE = col_double(),  
##   LULU = col_double(),
```

Was everything read in ok?

```
head(d) # take a quick look

## # A tibble: 6 x 14
##   Date     AAPL    MSFT    YHOO    NFLX    BMW      F      FB      GOOG
##   <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>    <dbl>   <dbl>   <dbl>
## 1 27/03/~  76.8   39.4   35.6   52.0   91.0   15.2   61.0   557.
## 2 28/03/~  76.7   40.3   35.9   51.3   92.0   15.4   60.0   558.
## 3 29/03/~  76.7   40.3   35.9   51.3   92.0   15.4   60.0   558.
## 4 30/03/~  76.7   40.3   35.9   51.3   92.0   15.4   60.0   558.
## 5 31/03/~  76.7   41.0   35.9   50.3   91.5   15.6   60.2   555.
## 6 01/04/~  77.4   41.4   36.5   52.1   92.2   16.3   62.6   566.
## # ... with 2 more variables: C <dbl>, JPM <dbl>
```

Was everything read in ok?

```
glimpse(d) # another way of taking a quick look

## Observations: 1,010
## Variables: 14
## $ Date <chr> "27/03/2014", "28/03/2014", "29/03/2014", "30/03/2014"
## $ AAPL <dbl> 76.7800, 76.6943, 76.6943, 76.6943, 76.6771, 76.6771
## $ MSFT <dbl> 39.36, 40.30, 40.30, 40.30, 40.99, 41.42, 41.35, 41.35
## $ YHOO <dbl> 35.5900, 35.9000, 35.9000, 35.9000, 35.9000, 35.9000, 35.9000, 35.9000
## $ NFLX <dbl> 52.026, 51.267, 51.267, 51.267, 50.290, 52.099, 52.099, 52.099
## $ BMW <dbl> 91.05, 92.02, 92.02, 92.02, 91.49, 92.25, 92.75, 92.75
## $ F <dbl> 15.25, 15.45, 15.45, 15.45, 15.60, 16.32, 16.45, 16.45
## $ FB <dbl> 60.970, 60.010, 60.010, 60.010, 60.240, 62.620, 62.620, 62.620
## $ GOOG <dbl> 556.931, 558.456, 558.456, 558.456, 555.445, 555.445, 555.445, 555.445
## $ GM <dbl> 34.51, 34.73, 34.73, 34.73, 34.42, 34.34, 34.85, 34.85
## $ GE <dbl> 25.81, 25.88, 25.88, 25.88, 25.89, 25.87, 26.00, 26.00
## $ LULU <dbl> 51.20, 51.89, 51.89, 51.89, 52.59, 52.98, 54.45, 54.45
## $ C <dbl> 47.45, 47.25, 47.25, 47.25, 47.60, 47.80, 48.25, 48.25
```

Manually specifying column types when reading

It looks as if Date has type <chr>. But R has a special date type.

```
d <- read_csv("data/prices.csv",
               col_types=cols(Date=col_date(
                 format="%d/%m/%Y"))))
head(d, n=3)

## # A tibble: 3 x 14
##   Date      AAPL    MSFT    YHOO    NFLX    BMW     F      FB      GOO
##   <date>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 2014-03-27  76.8   39.4   35.6   52.0   91.0   15.2   61.0   557
## 2 2014-03-28  76.7   40.3   35.9   51.3   92.0   15.4   60.0   558
## 3 2014-03-29  76.7   40.3   35.9   51.3   92.0   15.4   60.0   558
## # ... with 3 more variables: LULU <dbl>, C <dbl>, JPM <dbl>
```

See <https://r4ds.had.co.nz/data-import.html> for more examples like this.

Accessing subsets of a tibble

The \$ operator gets a named column as a vector:

```
d$MSFT
```

```
##      [1] 39.360 40.300 40.300 40.300 40.990 41.420 41.350 41.  
##     [10] 39.870 39.870 39.820 39.820 40.470 39.360 39.190 39.  
##    [19] 39.180 39.750 40.400 40.010 40.010 40.010 40.010 39.  
##   [28] 39.690 39.860 39.910 39.910 39.910 40.870 40.510 40.  
##   [37] 39.690 39.690 39.690 39.430 39.060 39.400 39.640 39.  
##   [46] 39.540 39.970 40.420 40.240 39.600 39.830 39.830 39.  
##   [55] 39.680 40.350 40.100 40.120 40.120 40.120 40.120 40.  
##   [64] 40.340 40.940 40.940 40.940 40.790 40.290 40.320 41.  
##   [73] 41.480 41.480 41.270 41.110 40.860 40.580 41.230 41.  
##   [82] 41.500 41.680 41.650 41.510 41.670 41.670 41.670 41.  
##   [91] 42.030 41.720 42.250 42.250 42.250 41.700 41.870 41.  
##  [100] 41.800 41.800 41.800 42.010 41.780 41.670 41.685 42.  
##  [109] 42.090 42.140 42.450 44.080 44.530 44.710 44.710 44.  
##  [118] 44.830 44.870 44.400 44.500 44.500 44.500 43.970 43.  
##  [127] 44.830 44.870 44.400 44.500 44.500 44.500 43.970 43.
```

Accessing subsets of a tibble

Square brackets, name as string => 1-column tibble:

```
d["MSFT"]
```

```
## # A tibble: 1,010 x 1
##       MSFT
##   <dbl>
## 1 39.4
## 2 40.3
## 3 40.3
## 4 40.3
## 5 41.0
## 6 41.4
## 7 41.4
## 8 41.0
## 9 39.9
## 10 39.9
## # ... with 1,000 more rows
```

Accessing subsets of a tibble

Square brackets and column number as int, same effect:

```
d[2]
```

```
## # A tibble: 1,010 x 1
##       AAPL
##   <dbl>
## 1 76.8
## 2 76.7
## 3 76.7
## 4 76.7
## 5 76.7
## 6 77.4
## 7 77.5
## 8 77.0
## 9 76.0
## 10 76.0
## # ... with 1,000 more rows
```

Accessing subsets of a tibble

Square brackets and “slice” – select several columns of d:

```
d[2:4]
```

```
## # A tibble: 1,010 x 3
##       AAPL    MSFT    YHOO
##   <dbl> <dbl> <dbl>
## 1 76.8  39.4  35.6
## 2 76.7  40.3  35.9
## 3 76.7  40.3  35.9
## 4 76.7  40.3  35.9
## 5 76.7  41.0  35.9
## 6 77.4  41.4  36.5
## 7 77.5  41.4  36.6
## 8 77.0  41.0  35.8
## 9 76.0  39.9  34.3
## 10 76.0  39.9  34.3
## # ... with 1,000 more rows
```

Accessing subsets of a tibble

Get the first 10 rows of MSFT column:

```
d$MSFT[1:10]  
## [1] 39.36 40.30 40.30 40.30 40.99 41.42 41.35 41.01 39.87
```

Non-tidy data (1)

	treatmenta	treatmentb
John Smith	—	2
Jane Doe	16	11
Mary Johnson	3	1

Non-tidy data (2)

	John Smith	Jane Doe	Mary Johnson
treatmenta	—	16	3
treatmentb	2	11	1

Tidy Data

person	treatment	result
John Smith	a	—
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

"In tidy data...

Every value belongs to a variable and an observation. [...]

- 1 Each variable forms a column.
- 2 Each observation forms a row.
- 3 Each type of observational unit forms a table.

This is Codd's 3rd normal form" – R4DS

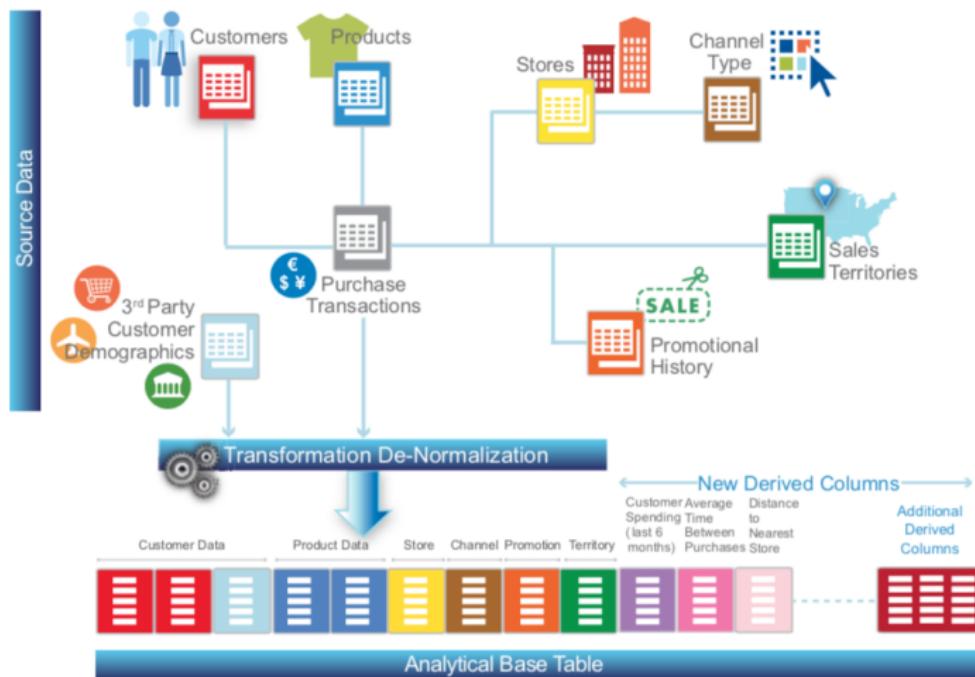
- *Is it?* (See code/tidy_data.xlsx)

Forms of data

- Tidy Data versus Codd's 3rd Normal Form (3NF), studied in database theory
- Tidy Data versus analytical base table (ABT)
- Some theory: <http://www.jstatsoft.org/v59/i10/paper>.

Analytical base table

A DB in 3NF has multiple tables and no repetition of data. An ABT is one table with repetition and derived data. (Image from SAS).



Why Tidy Data?

- Consistent underlying structure => consistent tools (ggplot, dplyr, etc.) => easier to learn
- Suits R's vectorisation

Another important motivation: if they are regularly queried together, we gain ease of querying. We may lose efficiency of storage, may re-introduce potential anomalies (relative to 3NF).

gather

country	year	cases	country	1999	2000
Afghanistan	1999	745	Afghanistan	745	2666
Afghanistan	2000	2666	Brazil	37737	80488
Brazil	1999	37737	China	212258	213766
Brazil	2000	80488			
China	1999	212258			
China	2000	213766			

table4

The diagram illustrates the process of gathering data. It shows two tables side-by-side. The first table has columns for country, year, and cases. The second table has columns for country, 1999, and 2000. Arrows point from the 'cases' values in the first table to the corresponding '1999' and '2000' values in the second table, demonstrating how the 'cases' variable is being mapped to two different years.

```
gather(table4b, key="year", value="cases", c("1999", "2000"))
```

spread

The diagram illustrates the 'spread' operation. On the left, a wide table has four columns: country, year, key, and value. On the right, a long table has four columns: country, year, cases, and population. Arrows point from each row in the wide table to the corresponding row in the long table, showing how the 'key' and 'value' columns are mapped into the 'cases' and 'population' columns of the long table.

country	year	key	value	country	year	cases	population
Afghanistan	1999	cases	745	Afghanistan	1999	745	19987071
Afghanistan	1999	population	19987071	Afghanistan	2000	260	20595360
Afghanistan	2000	cases	2666	Brazil	1999	37737	172006362
Afghanistan	2000	population	20595360	Brazil	2000	80488	174504898
Brazil	1999	cases	37737	China	1999	212258	1272915272
Brazil	1999	population	172006362	China	2000	213766	1280428583
Brazil	2000	cases	80488				
Brazil	2000	population	174504898				
China	1999	cases	212258				
China	1999	population	1272915272				
China	2000	cases	213766				
China	2000	population	1280428583				

table2

```
spread(table2, key="key", value="value")
```

Some more tidying functions

- `separate`: separate a column which encodes two variables into two columns, e.g. `name` "James McDermott" -> `first_name` "James", `second_name` "McDermott"
- `unite`: the opposite
- `complete`: add missing rows (and use NA as the missing value)

Exercises

- 1 Recall our experiment on running time for sorting an array of different sizes. The original data (before we added extra columns) is available in `data/sort_times_original.csv`. Read it in to a tibble. (You might need to set the working directory first.)
- 2 Use `glimpse` to take a look. What types do the columns have?
- 3 In what way is this *not* tidy data? Use `gather` to fix it. Hint: the result should have shape 50×3 with columns `n`, `run_number`, `run_time`.
- 4 It would be nicer if `run_number` was just an integer, eg 0, instead of `run0`. Use `separate` to split it into two parts. Hint: use `into=c("dummy", "run_number")`.
- 5 Look again at the result. We don't need that "dummy" column. Use `NA` to omit it. Hint: see `?separate` for help on `into`.
- 6 Look again – `run_number` is still not an integer! Fix this. Hint: `separate` can guess the correct type to convert to, but see `?separate` again to see how to ask it to.
- 7 Write it to a file `data/sort_times_tidy.csv` using `write_csv()`.

Solutions

Exercise 1

```
d <- read_csv("data/sort_times_original.csv")  
  
## Parsed with column specification:  
## cols(  
##   n = col_double(),  
##   run0 = col_double(),  
##   run1 = col_double(),  
##   run2 = col_double(),  
##   run3 = col_double(),  
##   run4 = col_double()  
## )
```

Exercise 2

```
glimpse(d) # All columns of type `dbl`, which is ok

## Observations: 10
## Variables: 6
## $ n      <dbl> 1e+06, 2e+06, 3e+06, 4e+06, 5e+06, 6e+06, 7e+06, 8e+06, 9e+06, 1e+07
## $ run0   <dbl> 0.09924603, 0.19706607, 0.30280304, 0.44548678, 0.58392051, 0.72235425, 0.86078798, 0.99922172, 1.13765545, 1.27608918
## $ run1   <dbl> 0.1099961, 0.1945050, 0.3008888, 0.5314040, 0.7620278, 0.9926516, 1.2232754, 1.4538992, 1.6845230, 1.9151468
## $ run2   <dbl> 0.1018548, 0.2033260, 0.3165970, 0.4161611, 0.5158342, 0.6155072, 0.7151803, 0.8148534, 0.9145265, 1.0141995
## $ run3   <dbl> 0.1004527, 0.1933441, 0.3653409, 0.4889781, 0.6125159, 0.7362531, 0.8600903, 0.9839275, 1.1077647, 1.2316019
## $ run4   <dbl> 0.1140921, 0.2565329, 0.3853610, 0.4850140, 0.6137518, 0.7474896, 0.8812274, 1.0149652, 1.1487030, 1.2824408
```

Exercise 3

```
d <- gather(d, key="run_number", value="run_time",
             run0, run1, run2, run3, run4)
```

Exercise 4

```
separate(d, run_number,
          into=c("dummy", "run_number"), sep=3)

## # A tibble: 50 x 4
##       n dummy run_number run_time
##   <dbl> <chr>  <chr>      <dbl>
## 1 1000000 run    0         0.0992
## 2 2000000 run    0         0.197 
## 3 3000000 run    0         0.303 
## 4 4000000 run    0         0.445 
## 5 5000000 run    0         0.584 
## 6 6000000 run    0         0.771 
## 7 7000000 run    0         1.54  
## 8 8000000 run    0         0.982 
## 9 9000000 run    0         1.24  
## 10 10000000 run   0         1.38 
## # ... with 40 more rows
```

Exercise 5

```
separate(d, run_number,
          into=c(NA, "run_number"), sep=3)

## # A tibble: 50 x 3
##       n run_number run_time
##   <dbl> <chr>        <dbl>
## 1 1000000 0        0.0992
## 2 2000000 0        0.197 
## 3 3000000 0        0.303 
## 4 4000000 0        0.445 
## 5 5000000 0        0.584 
## 6 6000000 0        0.771 
## 7 7000000 0        1.54  
## 8 8000000 0        0.982 
## 9 9000000 0        1.24  
## 10 10000000 0      1.38 
## # ... with 40 more rows
```

Exercise 6

```
d <- separate(d, run_number,
               into=c(NA, "run_number"), sep=3,
               convert=TRUE)

d
## # A tibble: 50 x 3
##       n run_number run_time
##   <dbl>     <int>     <dbl>
## 1 1000000     0    0.0992
## 2 2000000     0    0.197
## 3 3000000     0    0.303
## 4 4000000     0    0.445
## 5 5000000     0    0.584
## 6 6000000     0    0.771
## 7 7000000     0    1.54
## 8 8000000     0    0.982
## 9 9000000     0    1.24
## 10 10000000    0    1.38
```

Exercise 7

```
write_csv(d, "data/sort_times_tidy.csv")
```

R dplyr

James McDermott

NUI Galway



Programming and Tools for Artificial Intelligence

Dr James McDermott

dplyr

dplyr

dplyr is a package for relational operations on data. That is, it does stuff similar to SQL, which many students will be familiar with (also comparable to Excel and Pandas). In particular:

- `filter` (choose rows, like SQL `where`)
- `arrange` (sort rows)
- `select` (choose columns, like SQL `select`)
- `mutate` (add columns)
- `summarise` (condense multiple values)
- `sample_n`, `sample_frac` (for taking a quick look at a sub-sample, see also `head`)
- `inner_join`, `left_join`, `right_join`, `full_join` (join two tables, like SQL `join`)

dplyr and the pipe operator

All the dplyr verbs (select, etc.) have three things in common (from <https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html>):

- 1 The first argument is a data frame [actually, a tibble].
- 2 The subsequent arguments describe what to do with the data frame.
You can refer to columns in the data frame directly without using \$.
- 3 The result is a new data frame

Therefore, it is natural to *chain* operations. That is where the *pipe* operator comes in.

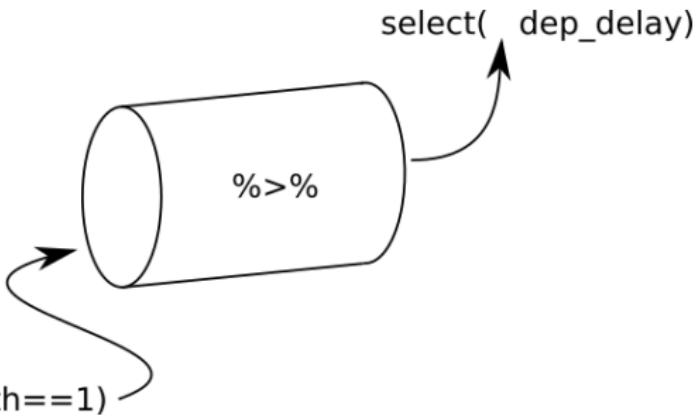
The pipe %>%



(Magritte)

- In Unix, the *pipe* symbol | is used to pass data from one command to another, e.g. ls | grep -v "#"
- In `magrittr`, the pipe %>% passes the output of its left-hand side to become the first argument of its right-hand side.

The pipe %>%



```
filter(d, month==1) %>% select(dep_delay)
```

The pipe %>%

In R, special operators are often named with double % symbols. The *pipe* operator %>% is an example.

t %>% f() means precisely f(t), i.e. the output of the LHS becomes the first argument of the RHS.

It is useful to avoid complicated nested expressions:

t %>% f("abc") %>% g("x", "y") is easier to read than

g(f(t, "abc"), "x", "y")

(isn't it?)

Example: NYC Flights

```
# uncomment if not already installed
# install.packages("nycflights13")
library(nycflights13)
library(tidyverse)

## -- Attaching packages -----
## v ggplot2 3.2.1           v purrr   0.3.2
## v tibble   2.1.1           v dplyr    0.8.0.1
## v tidyverse 0.8.3          v stringr  1.4.0
## v readr    1.3.1           vforcats  0.4.0

## -- Conflicts -----
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Take a quick look at data

```
flights
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_
##   <int> <int> <int>     <int>             <int>      <dbl> <
## 1 2013     1     1       517              515        2
## 2 2013     1     1       533              529        4
## 3 2013     1     1       542              540        2
## 4 2013     1     1       544              545       -1
## 5 2013     1     1       554              600       -6
## 6 2013     1     1       554              558       -4
## 7 2013     1     1       555              600       -5
## 8 2013     1     1       557              600       -3
## 9 2013     1     1       557              600       -3
## 10 2013    1     1       558              600       -2
## # ... with 336,766 more rows, and 12 more variables: sched_
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <
## #   ... with 336,766 more rows, and 12 more variables: sched_
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <
```

filter

Let's see just flights on Jan 1:

```
flights %>% filter(month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_
##   <int> <int> <int>     <int>             <int>      <dbl> <...
## 1 2013     1     1       517              515        2
## 2 2013     1     1       533              529        4
## 3 2013     1     1       542              540        2
## 4 2013     1     1       544              545       -1
## 5 2013     1     1       554              600       -6
## 6 2013     1     1       554              558       -4
## 7 2013     1     1       555              600       -5
## 8 2013     1     1       557              600       -3
## 9 2013     1     1       557              600       -3
## 10 2013    1     1       558              600       -2
## # ... with 832 more rows, and 12 more variables: sched_arr_
## #   time, arr_delay, origin, dest, distance, hour, minute,
## #   carrier, tailnum, flight, tailnum, arr_time
```

filter

Remember, this is internally translated to:

```
filter(flights, month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_
##   <int> <int> <int>     <int>             <int>     <dbl> <...
## 1 2013     1     1      517              515        2
## 2 2013     1     1      533              529        4
## 3 2013     1     1      542              540        2
## 4 2013     1     1      544              545       -1
## 5 2013     1     1      554              600       -6
## 6 2013     1     1      554              558       -4
## 7 2013     1     1      555              600       -5
## 8 2013     1     1      557              600       -3
## 9 2013     1     1      557              600       -3
## 10 2013    1     1      558              600       -2
## # ... with 832 more rows, and 12 more variables: sched_arr_
## #   time, arr_delay, origin, dest, distance, hour, minute,
## #   carrier, tailnum, flight, tailnum, arr_time
```

filter

Which flight departure was delayed the longest? We can use filter again:

```
flights %>% filter(dep_delay == max(dep_delay, na.rm=TRUE))

## # A tibble: 1 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>     <int>         <int>     <dbl>    <int>
## 1 2013     1     9       641          900      1301      1301
## # ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
## #   time_hour <dttm>
```

Notice that `na.rm=TRUE` *inside* `max()` is essential: consider `max(flights$dep_delay)` to see why.

Syntax notes

- 1 `na.rm` argument
- 2 We can refer to columns with no special syntax (not even quotes)
- 3 Remember `==` for equality (I put spaces), but `=` for passing keyword arguments (I don't put spaces, as in Python).

filter examples

Boolean AND: use comma as we already saw

```
flights %>% filter(month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_
##   <int> <int> <int>     <int>           <int>     <dbl>   <
## 1 2013     1     1       517             515        2
## 2 2013     1     1       533             529        4
## 3 2013     1     1       542             540        2
## 4 2013     1     1       544             545       -1
## 5 2013     1     1       554             600       -6
## 6 2013     1     1       554             558       -4
## 7 2013     1     1       555             600       -5
## 8 2013     1     1       557             600       -3
## 9 2013     1     1       557             600       -3
## 10 2013    1     1       558             600       -2
## # ... with 832 more rows, and 12 more variables: sched_
## # ... with 832 more rows, and 12 more variables: sched_
```

filter examples

Boolean OR: use |

```
flights %>% filter(month == 1 | month == 12)
```

```
## # A tibble: 55,139 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_
##   <int> <int> <int>     <int>          <int>     <dbl>   <
## 1 2013     1     1       517            515        2
## 2 2013     1     1       533            529        4
## 3 2013     1     1       542            540        2
## 4 2013     1     1       544            545       -1
## 5 2013     1     1       554            600       -6
## 6 2013     1     1       554            558       -4
## 7 2013     1     1       555            600       -5
## 8 2013     1     1       557            600       -3
## 9 2013     1     1       557            600       -3
## 10 2013    1     1       558            600       -2
## # ... with 55,129 more rows, and 12 more variables: sched_
## # ... with 55,129 more rows, and 12 more variables: sched_
## # ... with 55,129 more rows, and 12 more variables: sched_
```

filter examples

%in% operator does the same as above:

```
flights %>% filter(month %in% c(1, 12))
```

```
## # A tibble: 55,139 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_
##   <int> <int> <int>    <int>          <int>     <dbl>   <
## 1 2013     1     1      517            515        2
## 2 2013     1     1      533            529        4
## 3 2013     1     1      542            540        2
## 4 2013     1     1      544            545       -1
## 5 2013     1     1      554            600       -6
## 6 2013     1     1      554            558       -4
## 7 2013     1     1      555            600       -5
## 8 2013     1     1      557            600       -3
## 9 2013     1     1      557            600       -3
## 10 2013    1     1      558            600       -2
## # ... with 55,129 more rows, and 12 more variables: sched_
## #   arr_time, carrier, tailnum, flight, origin, dest, distance,
## #   hour, minute, cancellations, diverted, reason, origin_is_lax
```

arrange

dplyr becomes like a programmatic interface to Excel,
e.g. sort-by-column:

```
arrange(flights, dep_delay) # sort-by-column
```

```
## # A tibble: 336,776 x 19
##       year month   day dep_time sched_dep_time dep_delay arr...
##       <int> <int> <int>    <int>           <int>     <dbl>   ...
## 1 2013     12     7      2040            2123      -43
## 2 2013      2     3      2022            2055      -33
## 3 2013     11    10      1408            1440      -32
## 4 2013      1    11      1900            1930      -30
## 5 2013      1    29      1703            1730      -27
## 6 2013      8     9      729             755      -26
## 7 2013     10    23      1907            1932      -25
## 8 2013      3    30      2030            2055      -25
## 9 2013      3     2      1431            1455      -24
## 10 2013     5     5      934             958      -24
```

arrange

Descending order:

```
arrange(flights, desc(dep_delay))
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr...
##   <int> <int> <int>    <int>          <int>     <dbl>   <...
## 1 2013     1     9      641            900     1301
## 2 2013     6    15     1432           1935     1137
## 3 2013     1    10     1121           1635     1126
## 4 2013     9    20     1139           1845     1014
## 5 2013     7    22      845           1600     1005
## 6 2013     4    10     1100           1900      960
## 7 2013     3    17     2321            810      911
## 8 2013     6    27      959           1900      899
## 9 2013     7    22     2257            759      898
## 10 2013    12     5      756           1700      896
## # ... with 336,766 more rows, and 12 more variables: sched...
```

select chooses columns

```
select(flights, day, month, year)

## # A tibble: 336,776 x 3
##       day month year
##   <int> <int> <int>
## 1     1     1  2013
## 2     1     1  2013
## 3     1     1  2013
## 4     1     1  2013
## 5     1     1  2013
## 6     1     1  2013
## 7     1     1  2013
## 8     1     1  2013
## 9     1     1  2013
## 10    1     1  2013
## # ... with 336,766 more rows
```

select chooses columns

Select all columns from year to day (behind the scenes, the columns are *numbers*):

```
select(flights, year:day)
```

```
## # A tibble: 336,776 x 3
##       year   month   day
##   <int> <int> <int>
## 1 2013     1     1
## 2 2013     1     1
## 3 2013     1     1
## 4 2013     1     1
## 5 2013     1     1
## 6 2013     1     1
## 7 2013     1     1
## 8 2013     1     1
## 9 2013     1     1
## 10 2013    1     1
```

Using select to deselect

```
select(flights, -(month:minute))
```

```
## # A tibble: 336,776 x 2
##       year time_hour
##   <int> <dttm>
## 1  2013 2013-01-01 05:00:00
## 2  2013 2013-01-01 05:00:00
## 3  2013 2013-01-01 05:00:00
## 4  2013 2013-01-01 05:00:00
## 5  2013 2013-01-01 06:00:00
## 6  2013 2013-01-01 05:00:00
## 7  2013 2013-01-01 06:00:00
## 8  2013 2013-01-01 06:00:00
## 9  2013 2013-01-01 06:00:00
## 10 2013 2013-01-01 06:00:00
## # ... with 336,766 more rows
```

More ways to select columns

```
select(flights, starts_with("d"))

## # A tibble: 336,776 x 5
##       day dep_time dep_delay dest   distance
##   <int>     <int>     <dbl> <chr>     <dbl>
## 1     1        1      517  IAH     1400
## 2     1        1      533  IAH     1416
## 3     1        1      542  MIA     1089
## 4     1        1      544  BQN     1576
## 5     1        1      554  ATL      762
## 6     1        1      554  ORD      719
## 7     1        1      555  FLL     1065
## 8     1        1      557  IAD      229
## 9     1        1      557  MCO      944
## 10    1        1      558  ORD      733
## # ... with 336,766 more rows
```

mutate

```
select(flights, year:day, ends_with("delay"),
       distance, air_time) %>%
  mutate(gain=dep_delay - arr_delay) %>%
  mutate(speed=distance / air_time * 60)
```

```
## # A tibble: 336,776 x 9
##   year month   day dep_delay arr_delay distance air_time
##   <int> <int> <int>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 2013    1     1        2        11      1400      227
## 2 2013    1     1        4        20      1416      227
## 3 2013    1     1        2        33      1089      160
## 4 2013    1     1       -1       -18      1576      183
## 5 2013    1     1       -6       -25      762       116
## 6 2013    1     1       -4        12      719       150
## 7 2013    1     1       -5        19     1065      158
## 8 2013    1     1       -3       -14      229       53
## 9 2013    1     1       -3        -8      944      140
```

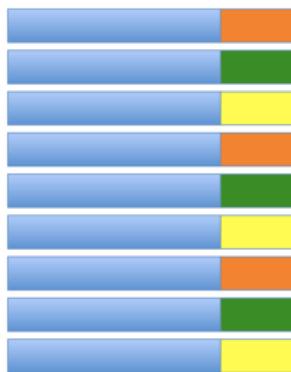
mutate

Remember none of these operations change the tibble itself, just return a new one. So we may decide to save the result in a new variable.

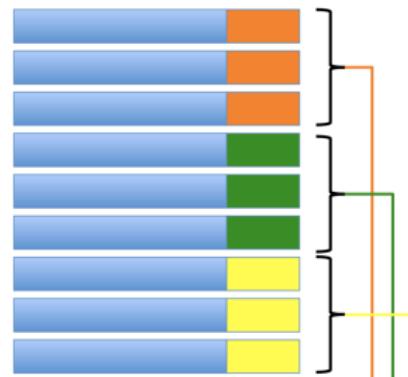
```
sml <- select(flights, year:day, ends_with("delay"),  
               distance, air_time) %>%  
  mutate(gain=dep_delay - arr_delay) %>%  
  mutate(speed=distance / air_time * 60)
```

group_by

Original data frame



Grouped data frame



summarise()



(R4DS)

group_by and summarise go well together

```
group_by(flights, carrier) %>%  
  summarise(dep_delay=mean(dep_delay, na.rm=TRUE))
```

```
## # A tibble: 16 x 2  
##   carrier dep_delay  
##   <chr>     <dbl>  
## 1 9E        16.7  
## 2 AA        8.59  
## 3 AS        5.80  
## 4 B6        13.0  
## 5 DL        9.26  
## 6 EV        20.0  
## 7 F9        20.2  
## 8 FL        18.7  
## 9 HA        4.90  
## 10 MQ       10.6  
## 11 OO       12.6
```

A bigger example: who

who is a dataset from the World Health Organisation which needs a lot of cleaning:

```
who %>%
  # gather many columns to 1
  gather(key, value, new_sp_m014:newrel_f65,
         na.rm = TRUE) %>%
  # fix inconsistent spelling
  mutate(key = stringr::str_replace(key, "newrel", "new_rel"))
  # split eg "new_sp_m014" -> "new", "sp", "m014"
  separate(key, c("new", "var", "sexage")) %>%
  # remove redundant/unneeded columns
  select(-new, -iso2, -iso3) %>%
  # split eg "m014" -> "m", "014"
  separate(sexage, c("sex", "age"), sep = 1)
```

A tibble: 76 046 x 6

A bigger example: who

See <https://r4ds.had.co.nz/tidy-data.html> for detailed explanation.

Exercise: look at the output of each step of this transformation, starting at `who` itself, to understand the need for the next.

Some more handy functions (some from dplyr)

- Offset a vector of values: `lead` and `lag`
- Cumulative calculations: `cumsum`, `cummax`, etc.
- Where does each value come in a sort? `min_rank`
- Counts: `n`, `n_distinct`

Functional programming in R

- dplyr and the pipe are already examples of functional programming!
E.g. all these operations don't change their input, just return a new version.
- Our friend map also exists in R. It makes a list.
- map_dbl and friends may be more useful since they return vectors.
Compare map and map_dbl in the following.

```
d = 1:5  
# R uses the opposite argument ordering, compared to Python  
map(d, sqrt)  
  
d = 1:5  
map_dbl(d, sqrt)
```

Functional programming in R

```
d %>% map_dbl(sqrt) # equivalent, using pipe
```

Functional programming in R

`map` and friends come from the `purrr` package, well-documented here:
<https://r4ds.had.co.nz/iteration.html#the-map-functions>

- The Joy of Functional Programming ACM Tech Talk webcast with Hadley Wickham can be viewed here:
<https://learning.acm.org/techtalks/functionalprogramming>
(Prerequisites: basic R, tibbles, distinction between lists, vectors, dataframes)

Summary

- tidy data: columns are variables, rows are observations
- tibbles
- pipe %>%
- verbs including select, filter, mutate, arrange, rename, gather, spread

Let's look at a cheatsheet for dplyr:

<https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>

Exercises

- Exercise 1: Our sort times data is available in tidy format as `sort_times_tidy.csv`. Use `group_by` and `summarise` to get the mean and the standard deviation for each `n`, and then for each `run_number`.
- A dataset of characters in *Star Wars* is available as `dplyr::starwars`. Exercise 2: Find all the human females. Exercise 3: Find the characters who are human *or* Wookiee. Exercise 4: Find the shortest character. Hint: recall we might need `na.rm`. Exercise 5: Add a new column called `BMI` giving the body mass index, where the formula is $BMI = m/h^2$ for mass m in kg and height h in metres. https://en.wikipedia.org/wiki/Body_mass_index. Exercise 6: Which character has the highest `BMI`?

Solutions

Exercise 1

```
d <- read_csv("data/sort_times_tidy.csv")

## Parsed with column specification:
## cols(
##   n = col_double(),
##   run_number = col_double(),
##   run_time = col_double()
## )

d %>% group_by(n) %>% summarise(mean_run_time=mean(run_time),

## # A tibble: 10 x 3
##       n  mean_run_time  sd_run_time
##   <dbl>      <dbl>        <dbl>
## 1 1000000  0.105       0.00654
## 2 2000000  0.209       0.0269
## 3 3000000  0.334       0.0387
```

Exercise 1

Notice that the mean and stddev for $n = 7$ million are anomalously high. One way this could occur is if our computer had a spike in CPU usage during the experiment, e.g. due to a browser loading a video.

Exercise 1

```
d %>% group_by(run_number) %>%
  summarise(mean_run_time=mean(run_time),
            sd_run_time=sd(run_time))
```

```
## # A tibble: 5 x 3
##   run_number mean_run_time sd_run_time
##       <dbl>          <dbl>        <dbl>
## 1 0           0.754        0.512
## 2 1           0.644        0.368
## 3 2           0.604        0.353
## 4 3           0.648        0.369
## 5 4           0.678        0.416
```

No major anomalies this time.

Exercise 2

```
sw <- dplyr::starwars
sw %>% filter(species == "Human", gender == "female") # human

## # A tibble: 9 x 13
##   name    height  mass hair_color skin_color eye_color birth_
##   <chr>    <int> <dbl> <chr>      <chr>      <chr>
## 1 Leia~     150     49 brown     light      brown
## 2 Beru~     165     75 brown     light      blue
## 3 Mon ~    150     NA auburn    fair       blue
## 4 Shmi~    163     NA black     fair       brown
## 5 Cordé    157     NA brown    light      brown
## 6 Dormé    165     NA brown    light      brown
## 7 Joca~    167     NA white    fair       blue
## 8 Rey       NA      NA brown    light      hazel
## 9 Padm~    165     45 brown    light      brown
## # ... with 5 more variables: homeworld <chr>, species <chr>
## #   vehicles <list>, starships <list>
```

Exercise 3

```
sw %>% filter(species == "Human" | species == "Wookiee") # hu
```

```
## # A tibble: 37 x 13
```

```
##   name   height  mass hair_color skin_color eye_color birth
##   <chr>   <int> <dbl> <chr>      <chr>      <chr>
## 1 Luke~    172     77 blond      fair       blue
## 2 Dart~    202    136 none       white      yellow
## 3 Leia~    150     49 brown      light      brown
## 4 Owen~    178    120 brown, gr~ light      blue
## 5 Beru~    165     75 brown      light      blue
## 6 Bigg~    183     84 black      light      brown
## 7 Obi~~   182     77 auburn, w~ fair      blue-gray
## 8 Anak~    188     84 blond      fair       blue
## 9 Wilh~    180     NA auburn, g~ fair       blue
## 10 Chew~   228    112 brown      unknown    blue
## # ... with 27 more rows, and 5 more variables: homeworld <ch
## #   species <chr>  films <list>  vehicles <list>  starships <
```

Exercise 4

```
sw %>% filter(height == max(height, na.rm=TRUE))

## # A tibble: 1 x 13
##   name    height  mass hair_color skin_color eye_color birth_
##   <chr>    <int> <dbl> <chr>       <chr>       <chr>
## 1 Yara~     264     NA none       white       yellow
## # ... with 5 more variables: homeworld <chr>, species <chr>,
## #   vehicles <list>, starships <list>
```

Exercise 5

```
# NB convert height from cm to metres before squaring
BMI <- function(h, m) {m / (h / 100)^2}
sw <- sw %>% mutate(bmi=BMI(height, mass))
```

Exercise 6

```
sw %>% filter(bmi == max(bmi, na.rm=TRUE))

## # A tibble: 1 x 14
##   name    height  mass hair_color skin_color eye_color birth_
##   <chr>    <int> <dbl> <chr>       <chr>       <chr>
## 1 Jabb~     175   1358 <NA>        green-tan~ orange
## # ... with 6 more variables: homeworld <chr>, species <chr>,
## #   vehicles <list>, starships <list>, bmi <dbl>
```

dplyr joins

James McDermott

NUI Galway



NUI Galway
OÉ Gaillimh

Programming and Tools for Artificial Intelligence

Dr James McDermott

dplyr joins

Relational databases

The main ideas of relational databases (SQL) are probably familiar to all:

- A database consists of tables
- A table consists of a set of columns
- A column has a type, and maybe some constraints (e.g. positive integer)
- Some column(s) may be designated as a key for the table

Joins

- As we know, in Relational Databases, it is good practice to use *normalisation*: splitting a table up into multiple tables, to avoid duplication of information and the possibility of *update anomalies*. 3NF is the result of normalisation.
- Doing ML/stats/analytics may require *de-normalisation* – re-joining – eventually to export to our ML/stats/analytics system.

Before normalisation

Movie rental DB

Date	Movie	Genre	Customer	Address
01-Jan	Amelie	Romance	Bob	11, Haight St
02-Jan	The Matrix	Sci-fi	Frida	Oxford Circus
02-Jan	Amelie	Romance	Carrie	99, Fifth Ave
05-Jan	Skyfall	Adventure	Bob	11, Haight St
05-Jan	Avengers	Sci-fi	Frida	Oxford Circus

After normalisation: 3rd Normal Form (3NF)

Rentals table

Date	Movie ID	Customer ID
01-Jan	102	1
02-Jan	101	2
02-Jan	102	3
05-Jan	103	1
05-Jan	104	2

Customer table

Customer ID	Name	Address
1	Bob	11, Haight St
2	Frida	Oxford Circus
3	Carrie	99, Fifth Ave

Movie table

Movie ID	Name	Genre
101	The Matrix	Sci-fi
102	Amelie	Romance
103	Skyfall	Adventure
104	Avengers	Sci-fi

Key columns

After normalisation, the link between data is via key columns – in this case, the Customer ID and Movie ID columns. It is possible to put the original table back together using a **join**. We say that we join **on** the key column.

SQL

In SQL, a JOIN might be something like this. This is an *implicit join*:

```
SELECT * FROM RENTALS, CUSTOMER  
WHERE RENTALS.CustomerID = CUSTOMER.CustomerID;
```

This is an equivalent *explicit join*:

```
SELECT * FROM RENTALS JOIN CUSTOMER  
ON RENTALS.CustomerID = CUSTOMER.CustomerID;
```

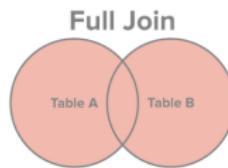
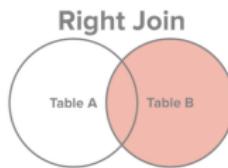
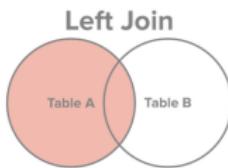
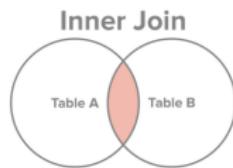
(This is not examinable.)

What is a join, really?

- Think of join as an *operator* whose left and right operands are tables, and whose result is a table formed as the union of their columns
- The *Cross join* is a good place to start. Conceptually, a cross join is a *Cartesian product of rows*. For every row in T1, we put it side by side with every row in T2. Think of that as a new joined table. Now we can select columns from it and filter rows using ON. In particular, we'll probably filter for rows where a key column in one table matches a key column in the other, discarding the large majority of this cross product.
- Other joins just restrict the “every row in T1” and “every row in T2” parts depending on which matches actually exist.

Different types of joins

There are a few types of joins. To distinguish them, many textbooks and cheatsheets proceed to Venn diagrams, e.g. <http://www.sql-join.com/sql-join-types> (below). These are helpful as mnemonics but the language of Venn diagrams is not sufficient to define the different joins.



Different types of joins (Data Wrangling Cheatsheet)

a	b
x1	x2
A	1
B	2
C	3

+

x1	x3
A	T
B	F
D	T

=

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

`dplyr::left_join(a, b, by = "x1")`

Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

`dplyr::right_join(a, b, by = "x1")`

Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

`dplyr::inner_join(a, b, by = "x1")`

Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

`dplyr::full_join(a, b, by = "x1")`

Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

`dplyr::semi_join(a, b, by = "x1")`

All rows in a that have a match in b.

x1	x2
C	3

`dplyr::anti_join(a, b, by = "x1")`

All rows in a that do not have a match in b.

Further reading

- Most people working in industry in the fields of AI, ML, Data Science, Statistics, etc., use relational databases and SQL a lot.
- We don't teach it, because it is usually seen as a topic for undergrad level. This MOOC is recommended as an optional catch-up or refresher:
 - Stanford Databases
<https://lagunita.stanford.edu/courses/DB/2014/SelfPaced/about>
 - (The following topics in the MOOC are recommended for a "short version": Introduction, JSON, Relational Algebra (Section 1), SQL, Relational Design Theory (Section 1), Unified Modelling Language, Online Analytical Processing)

Exercises

- 1 Read the three data files `rentals.csv`, `movies.csv`, `customers.csv`, all in the `data/` directory, as tibbles.
- 2 Optional: get R to read the Date column correctly. Hint:
https://readr.tidyverse.org/reference/parse_datetime.html
- 3 Using a `dplyr` join command, create a table showing the customer name and address for every rental.
- 4 Piping the result into another join command, recreate the full original table as shown under “Before Normalisation” above.
- 5 Notice the columns `Name.x` and `Name.y` which appear because there is a `Name` column in each of the `Movies` and `Customers` tables. Rename them.
- 6 Calculate the number of movies Frida watched of the Sci-fi genre.

Solutions

```
library(tidyverse)

## -- Attaching packages -----
## v ggplot2 3.2.1      v purrr    0.3.2
## v tibble   2.1.3      v dplyr    0.8.3
## v tidyverse 1.0.0      v stringr  1.4.0
## v readr    1.3.1      v forcats  0.4.0

## -- Conflicts -----
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Exercises 1 and 2:

```
rentals <- read_csv("data/rentals.csv",
                     col_types=cols(Date=col_date(
                         format="%d-%b-%Y"))))
movies <- read_csv("data/movies.csv")

## Parsed with column specification:
## cols(
##   MovieID = col_double(),
##   Name = col_character(),
##   Genre = col_character()
## )

customers <- read_csv("data/customers.csv")

## Parsed with column specification:
## cols(
##   CustomerID = col_double(),
##   Name = col_character(),
##   Address = col_character()
```

Customer name and address for each rental

```
inner_join(rentals, customers, by="CustomerID")  
## # A tibble: 5 x 5  
##   Date      MovieID CustomerID Name    Address  
##   <date>     <dbl>     <dbl> <chr>  <chr>  
## 1 2018-01-01     102       1 Bob    11, Haight St  
## 2 2018-01-02     101       2 Frida  Oxford Circus  
## 3 2018-01-02     102       3 Carrie 99, Fifth Ave  
## 4 2018-01-05     103       1 Bob    11, Haight St  
## 5 2018-01-05     104       2 Frida  Oxford Circus
```

Recreate original table

```
inner_join(rentals, customers, by="CustomerID") %>%  
  inner_join(movies, by="MovieID")  
  
## # A tibble: 5 x 7  
##   Date      MovieID CustomerID Name.x Address     Name.y  
##   <date>     <dbl>    <dbl> <chr>   <chr>     <chr>  
## 1 2018-01-01     102        1 Bob    11, Haight St Amelie  
## 2 2018-01-02     101        2 Frida  Oxford Circus The Ma...  
## 3 2018-01-02     102        3 Carrie 99, Fifth Ave Amelie  
## 4 2018-01-05     103        1 Bob    11, Haight St Skyfall  
## 5 2018-01-05     104        2 Frida  Oxford Circus Avenger
```

Rename columns

```
t = inner_join(rentals, customers, by="CustomerID") %>%
  inner_join(movies, by="MovieID") %>%
  rename(CustomerName=Name.x, MovieTitle=Name.y)
t
## # A tibble: 5 x 7
##   Date      MovieID CustomerID CustomerName Address    ...
##   <date>     <dbl>     <dbl> <chr>       <chr>    ...
## 1 2018-01-01     102       1 Bob        "11, Haight~ A...
## 2 2018-01-02     101       2 Frida     "Oxford Cir~ TH...
## 3 2018-01-02     102       3 Carrie    "99, Fifth ~ A...
## 4 2018-01-05     103       1 Bob        "11, Haight~ SK...
## 5 2018-01-05     104       2 Frida     "Oxford Cir~ AV...
```

Filter and count

```
t %>% filter(CustomerName=="Frida", Genre=="Sci-fi") %>%
  count()

## # A tibble: 1 x 1
##       n
##   <int>
## 1     2
```

Filter and count

The following is a solution to the problem, but it requires the programmer to do all the work in their head. That's not scalable or flexible and it's error-prone, so don't do this.

```
# Frida is CustomerID 2
# Movies 101 and 104 are Sci-fi
rentals %>% filter(CustomerID == 2,
                      MovieID %in% c(101, 104)) %>%
  count()

## # A tibble: 1 x 1
##       n
##   <int>
## 1     2
```

Plotting in R

James McDermott

NUI Galway



NUI Galway
OÉ Gaillimh

Programming and Tools for Artificial Intelligence

Dr James McDermott

Plotting in R

Plotting in R

Plotting is a central part of the data analysis loop. `ggplot2` is a great library that works well with `dplyr` and the rest of the Tidyverse.

```
library(tidyverse)
```

```
## -- Attaching packages -----
```

```
## v ggplot2 3.2.1      v purrr   0.3.2
## v tibble  2.1.1      v dplyr    0.8.0.1
## v tidyr   0.8.3      v stringr  1.4.0
## v readr   1.3.1      vforcats  0.4.0
```

```
## -- Conflicts -----
```

```
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
```

Example (mpg dataset)

We previously saw a small extract of this dataset `data/mpg_extract.csv`.
The full dataset is built-in to the `ggplot2` library.

```
dt <- ggplot2::mpg  
dt  
  
## # A tibble: 234 x 11  
##   manufacturer model displ year cyl trans drv cty  
##   <chr>        <chr>  <dbl> <int> <int> <chr> <chr> <int>  
## 1 audi         a4      1.8  1999    4 auto~ f     18  
## 2 audi         a4      1.8  1999    4 manu~ f     21  
## 3 audi         a4      2    2008    4 manu~ f     20  
## 4 audi         a4      2    2008    4 auto~ f     21  
## 5 audi         a4      2.8  1999    6 auto~ f     16  
## 6 audi         a4      2.8  1999    6 manu~ f     18  
## 7 audi         a4      3.1  2008    6 auto~ f     18  
## 8 audi         a4 q~  1.8  1999    4 manu~ 4     18  
## 9 audi         a4 q~  1.8  1999    4 auto~ 4     16
```

Example (mpg dataset)

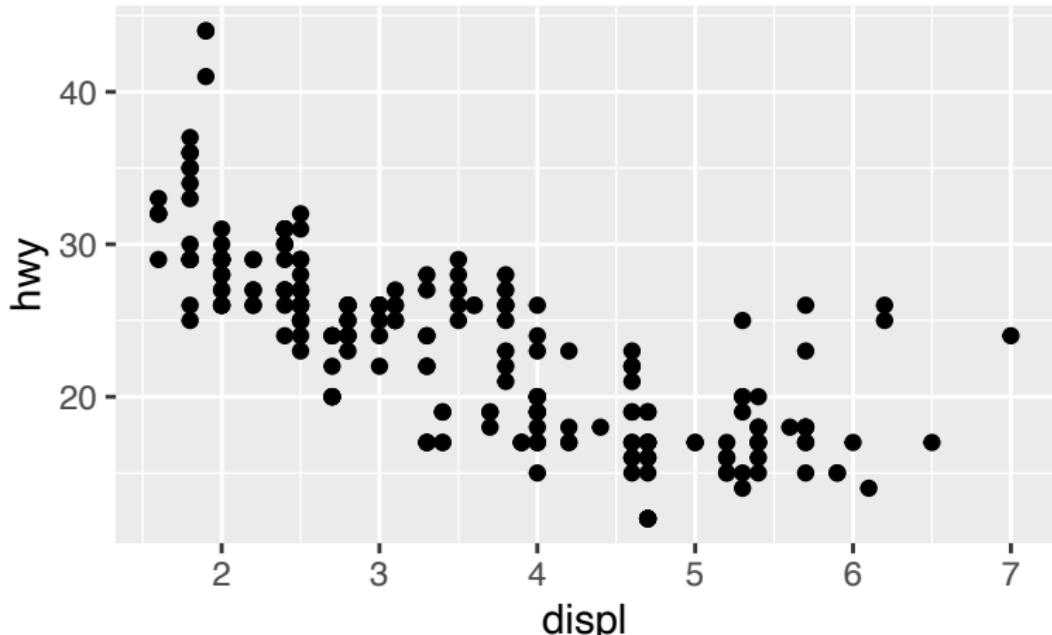
Do cars with big engines use more fuel than cars with small engines?
Among the variables are:

- `displ`, a car's engine volume ("displacement") in litres
- `hwy`, a car's fuel efficiency on the highway in miles per gallon

Scatterplot using geom_point

The library is called ggplot2, but the function is called ggplot.

```
ggplot(data = dt) +  
  geom_point(mapping = aes(x=displ, y=hwy))
```



Seeing/saving

- In R Studio, the plot appears in the bottom-right panel.
- If editing R Markdown, it appears inline.
- If we want to save:

```
ggsave("img/mpg_test.png")
```

```
## Saving 4 x 2.5 in image
```

General recipe for ggplot

```
ggplot(data = <DATA>) +  
<GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

The Grammar of Graphics

- Wilkinson, L. (2005), *The Grammar of Graphics* (2nd ed.). Statistics and Computing, New York: Springer.
- Wickham calls it “The most important modern work in graphical grammars”
- Every graph: a data set, a coordinate system, and visual marks representing data
- Wickham wrote the `ggplot2` package, an implementation of the grammar of graphics, which is used by most R practitioners.

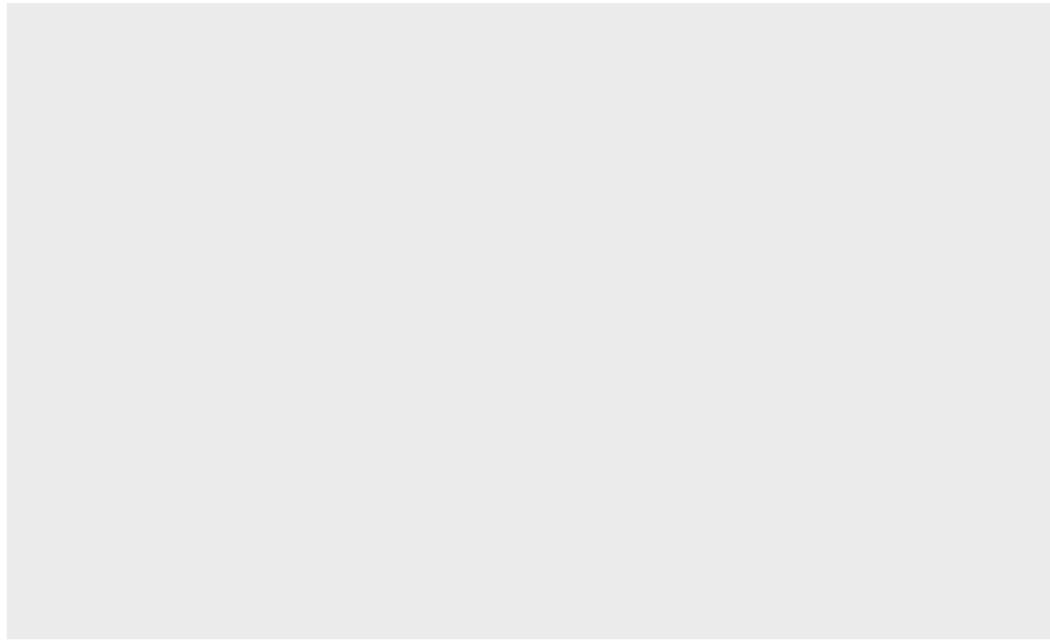
ggplot2 concepts

- Layers
- Aesthetic mappings
- Geometric objects

Layers

We can create a blank plot with something like this.

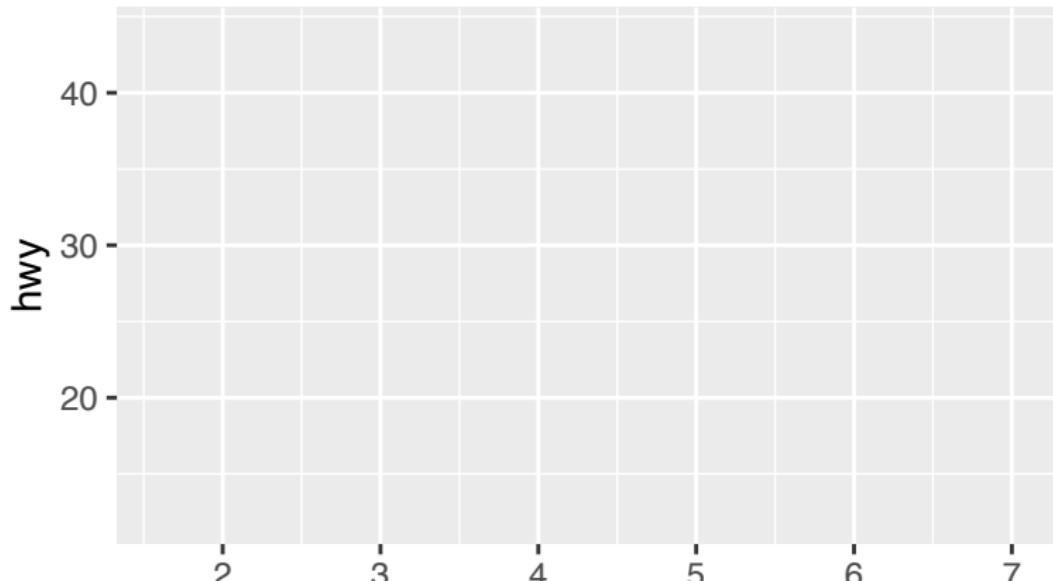
```
ggplot(mpg)
```



Aesthetics

aes says what variable maps to what aesthetic property e.g. colour, or position on an axis. It still doesn't have any *layers*. (Notice it is allowed to put aes inside ggplot.)

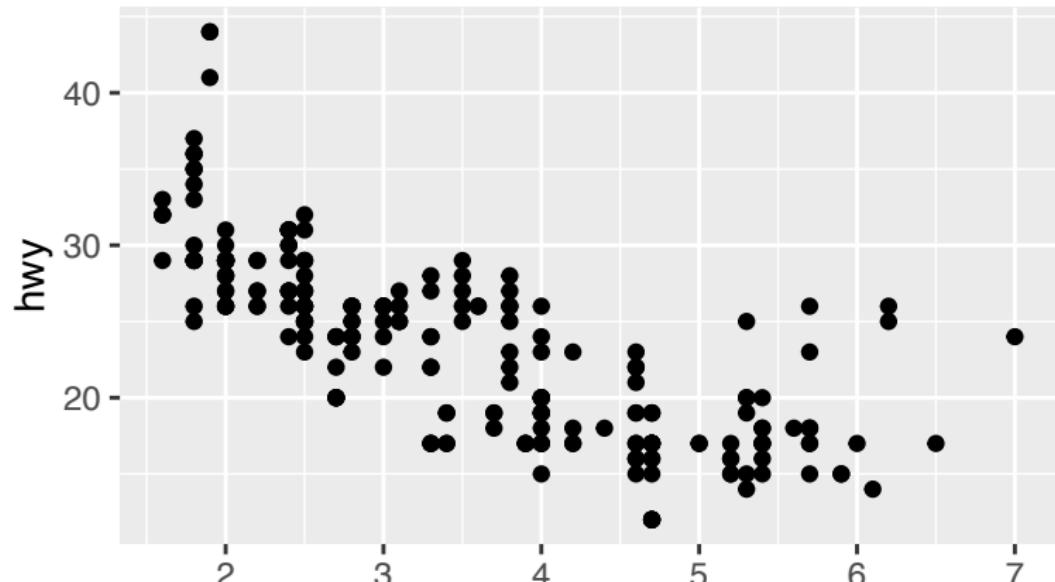
```
ggplot(mpg, aes(x=displ, y=hwy))
```



geoms

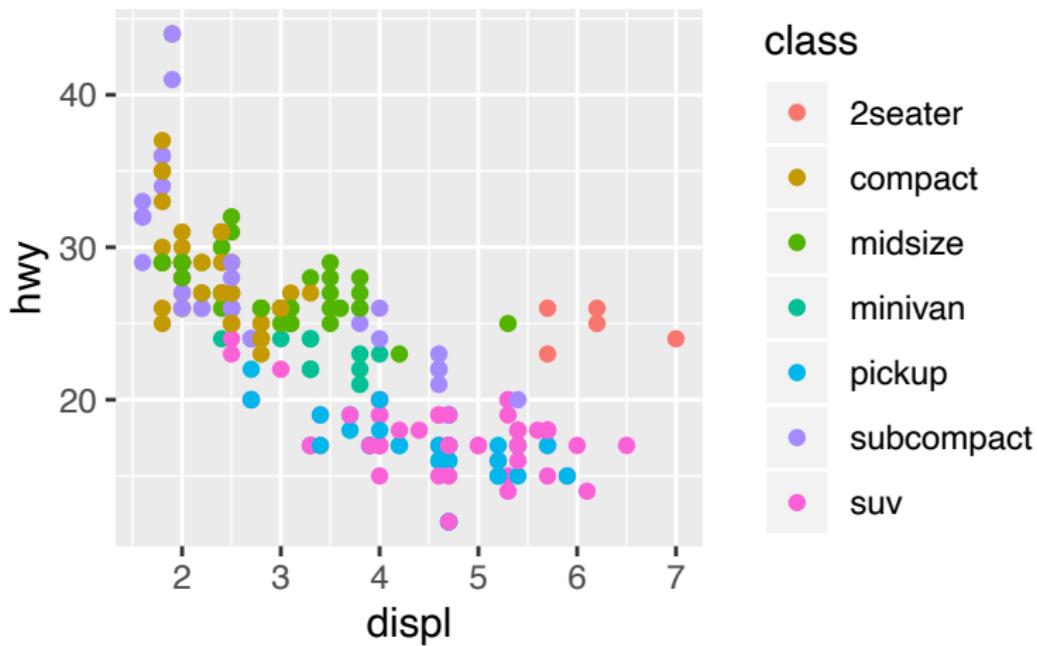
A *geom* is a way of translating a value to marks in the plot. We write + geom, and **that adds a layer**.

```
ggplot(data = dt) +  
  geom_point(mapping = aes(x=displ, y=hwy))
```



Adding an extra aesthetic

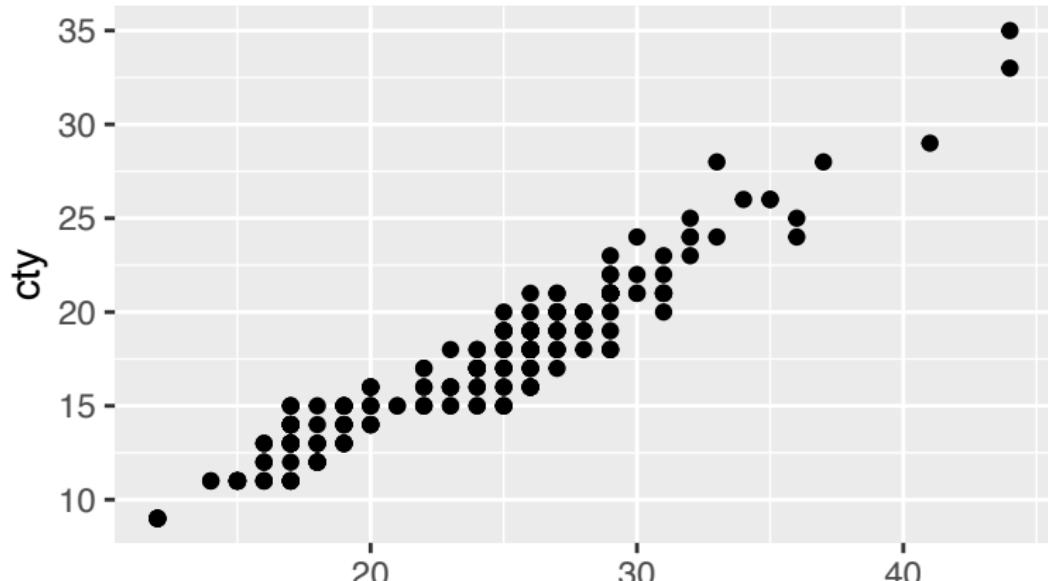
```
ggplot(data = dt) +  
  geom_point(mapping = aes(x=displ, y=hwy, colour=class))
```



More geoms

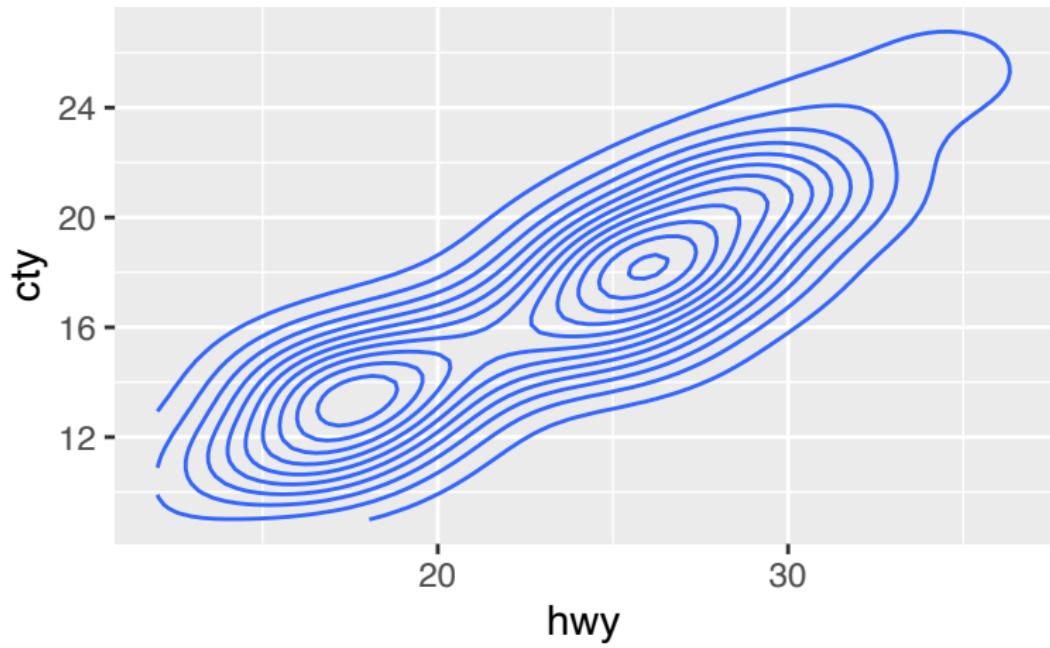
There are several different geoms. We already saw `geom_point`. Next, we'll look at how we can change appearance.

```
ggplot(mpg, aes(x=hwy, y=cty)) +  
  geom_point()
```



More geoms

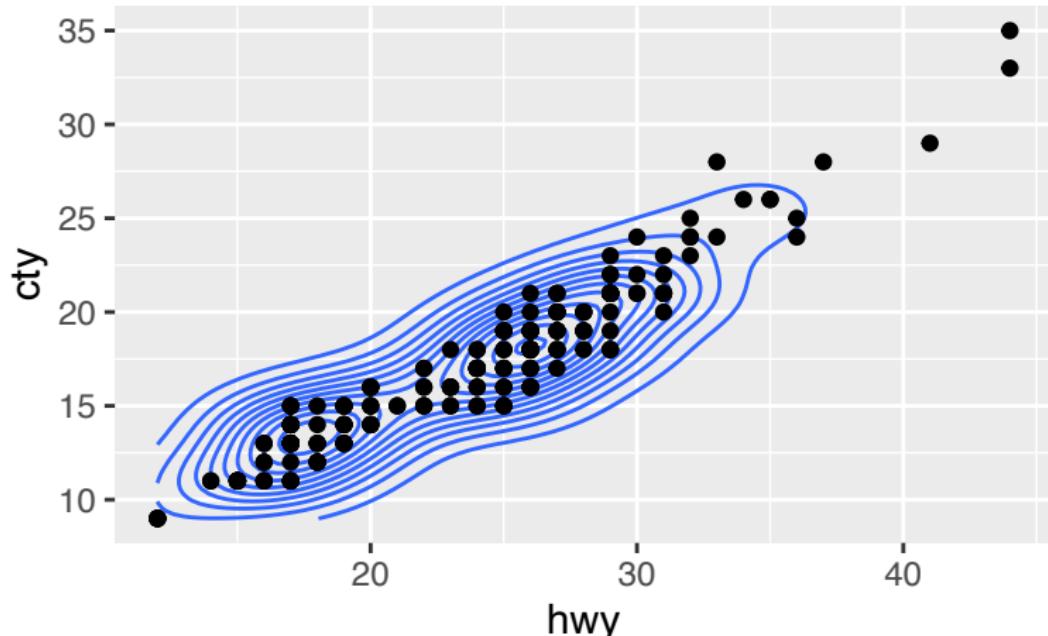
```
ggplot(mpg, aes(x=hwy, y=cty)) +  
  geom_density2d()
```



Multiple geoms

We can just write `+` to add multiple geoms to a plot (i.e. multiple layers).

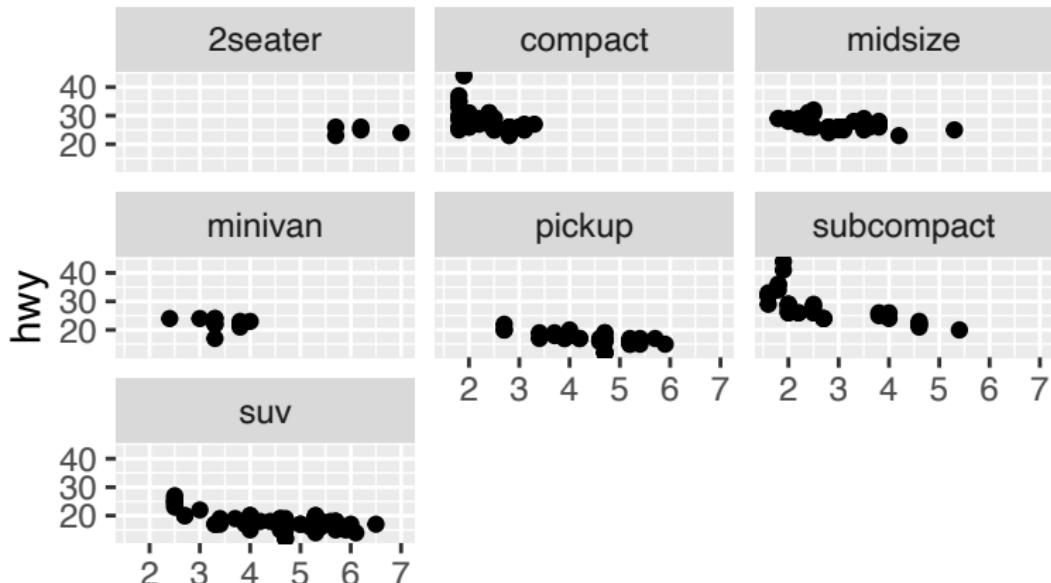
```
ggplot(mpg, aes(x=hwy, y=cty)) +  
  geom_density2d() + geom_point()
```



Faceting

Maybe a nicer way is instead to split the data into one graph per class.

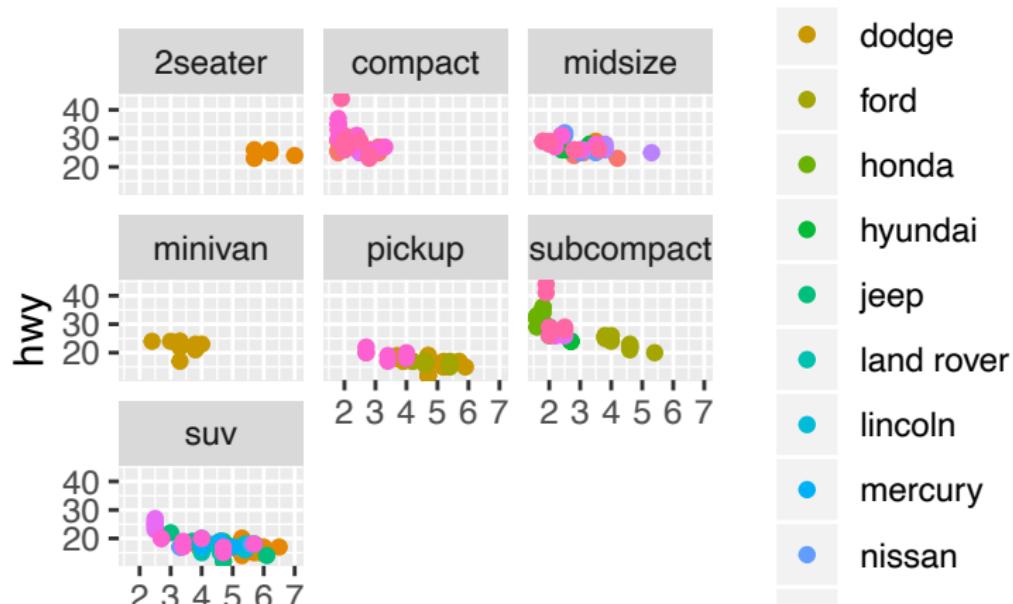
```
ggplot(data = dt) +  
  geom_point(mapping = aes(x=displ, y=hwy)) +  
  facet_wrap(~class) # notice tilde ~ for a *formula*
```



Adding another variable

Another variable is manufacturer. We are now showing 4 variables.

```
ggplot(data = dt) +  
  geom_point(mapping=aes(x=displ, y=hwy, colour=manufacturer))  
  facet_wrap(~class)
```



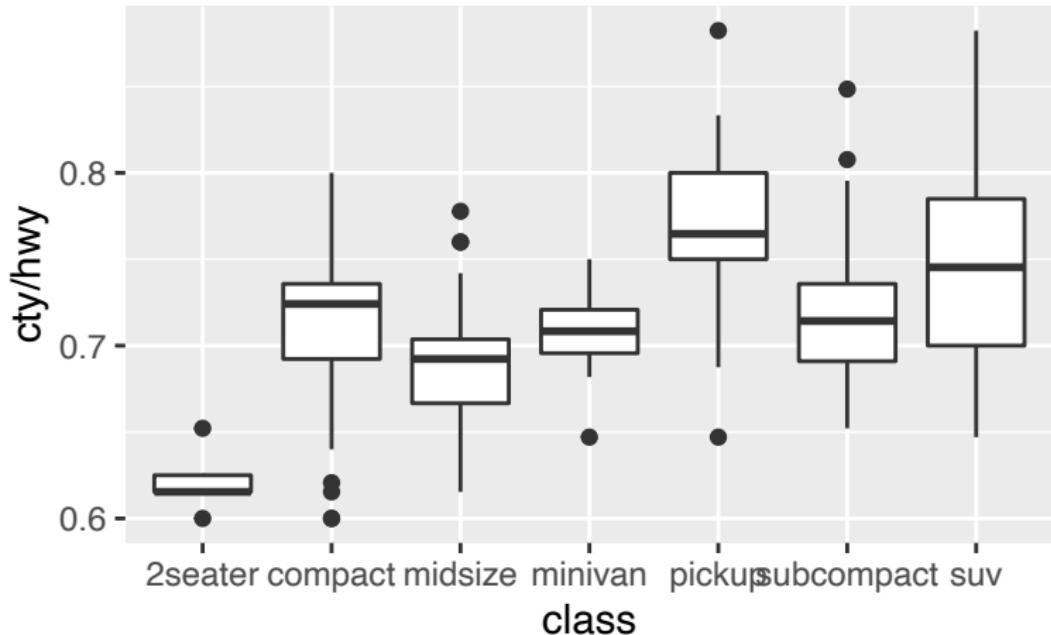
New variables

We can create a new variable inside the aes call.

New variables

Which car types emphasise city fuel efficiency over highway fuel efficiency?

```
ggplot(mpg, aes(x=class, y=cty/hwy)) +  
  geom_boxplot()
```

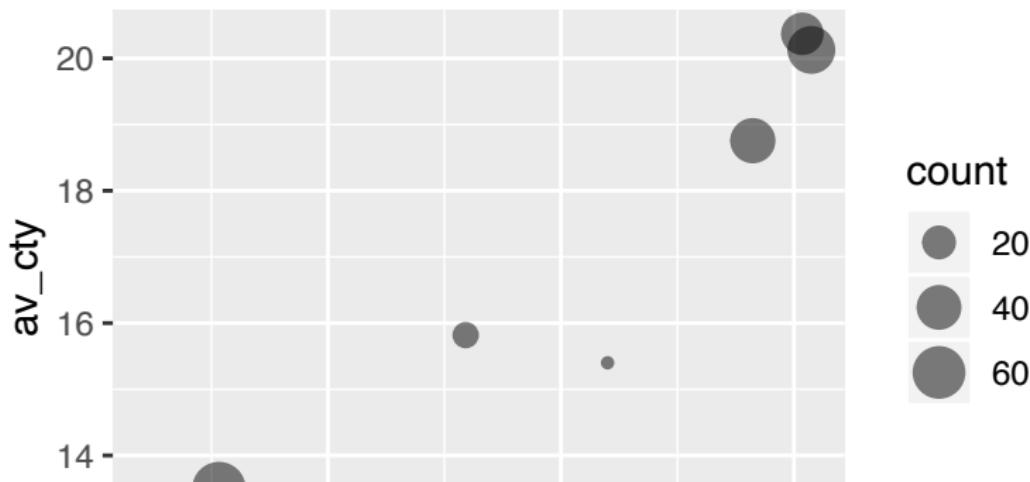


dplyr with ggplot

We can pipe the output of some `dplyr` manipulation straight into `ggplot` and then use aesthetics and `geom` commands to refine the plot.

dplyr with ggplot

```
mpg %>% group_by(class) %>%
  summarise(count=n(),
    av_hwy=mean(hwy),
    av_cty=mean(cty)) %>%
  ggplot(mapping = aes(x=av_hwy, y=av_cty)) +
  geom_point(aes(size=count), alpha=0.5)
```



Collision modifiers

These are methods of preventing marks (e.g. dots) from overlapping with each other.

- `dodge` (“smart” displacement of dots)
- `jitter` (random displacement of dots)
- `nudge` (manual displacement of dots)

There's an example with `nudge` in Assignment 2.

More references

- Manual <https://ggplot2.tidyverse.org/reference/>
- Cheatsheet
<https://github.com/rstudio/cheatsheets/blob/master/data-visualization-2.1.pdf>
- BBC using R for data journalism with a “house style” <https://medium.com/bbc-visual-and-data-journalism/how-the-bbc-visual-and-data-journalism-team-works-with-graphics-in-r-ed0b35693535>

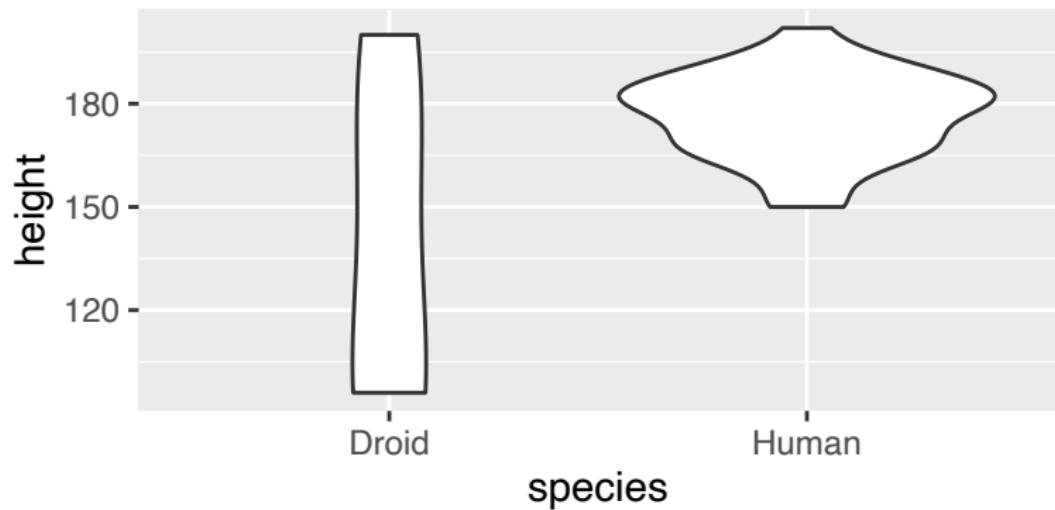
Exercises

Here are a few plots from well-known datasets. The exercise is to reproduce them using ggplot

Exercise 1

Character height in Star Wars using a “violin plot”. A violin plot is like a sideways, smoothed histogram, or like a box plot.

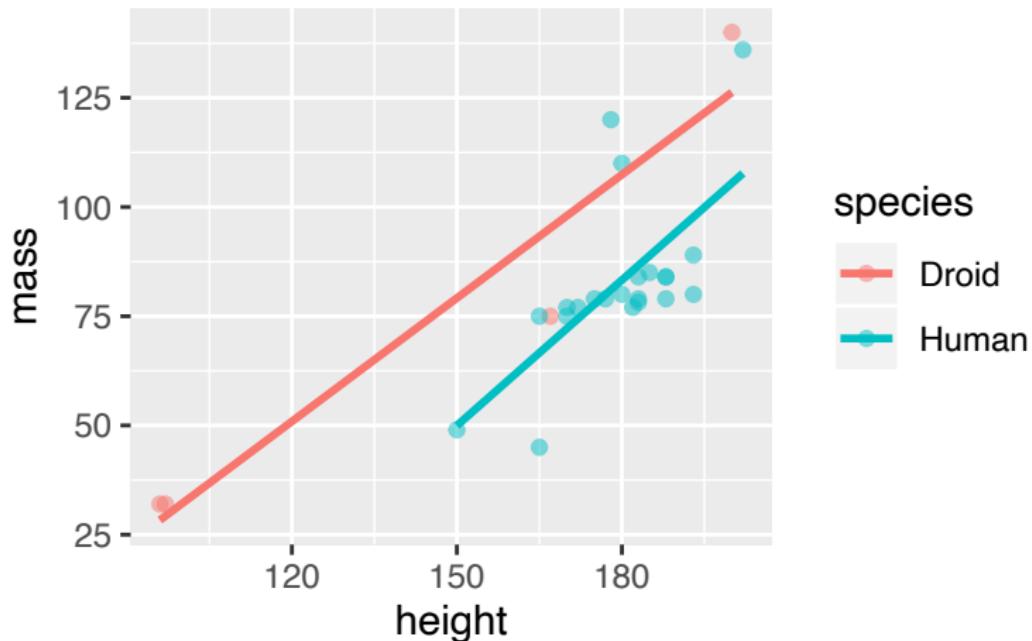
The dataset is `dplyr::starwars`.



Exercise 2

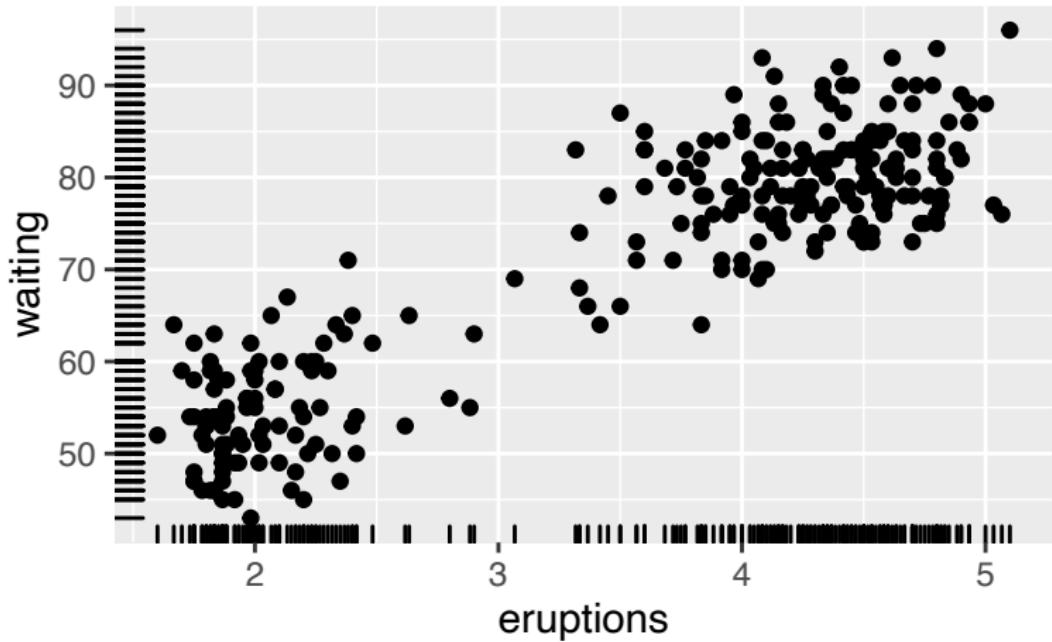
Character height and mass in Star Wars, with trendlines.

```
## Warning: Removed 14 rows containing non-finite values (statistic)
## Warning: Removed 14 rows containing missing values (geom_point)
```



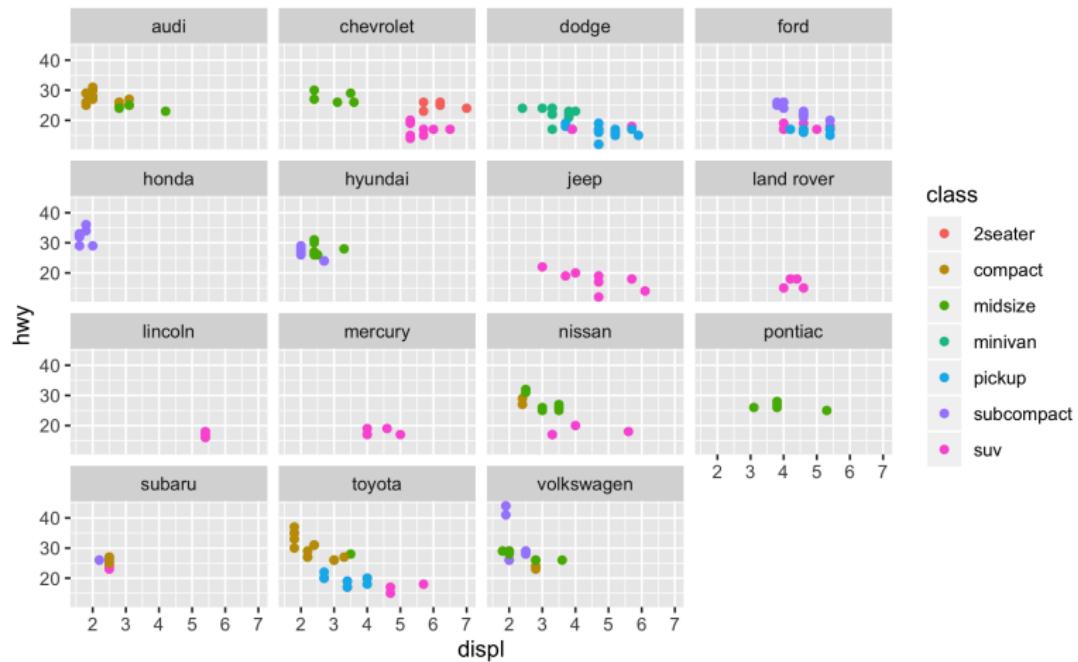
Exercise 3

Eruption length versus waiting time at the Old Faithful geyser. The dataset is `faithful`, built-in to R as a `data.frame`. Recall we can use `as_tibble` to convert to a tibble. Hint: this is called a “rug plot”.



Exercise 4

Like a previous plot, but now faceting manufacturer and showing class as colour.



Solutions

Exercise 1

```
s = dplyr::starwars  
s %>% select(height, mass, species) %>%  
  filter(height != "na.rm") %>%  
  filter(species %in% c("Human", "Droid")) %>%  
  ggplot(mapping=aes(x=species, y=height)) + geom_violin()
```

Exercise 2

```
s %>% filter(species == "Human" | species == "Droid") %>%
  ggplot(aes(x=height, y=mass, color=species)) +
  geom_point(alpha=0.5) + geom_smooth(method=lm, se=FALSE)
```

Exercise 3

```
f = as_tibble(faithful)
f %>% ggplot(aes(x=eruptions, y=waiting))
+ geom_point() + geom_rug()
```

Exercise 4

```
dt <- ggplot2::mpg  
ggplot(data = dt) +  
  geom_point(mapping = aes(x=displ, y=hwy, colour=class)) +  
  facet_wrap(~manufacturer)  
ggsave("img/R_mpg_displ_hwy_manu_class.png")
```

Statistics in R

James McDermott

NUI Galway



NUI Galway
OÉ Gaillimh

Programming and Tools for Artificial Intelligence

Dr James McDermott

Statistics in R

Load Tidyverse as usual

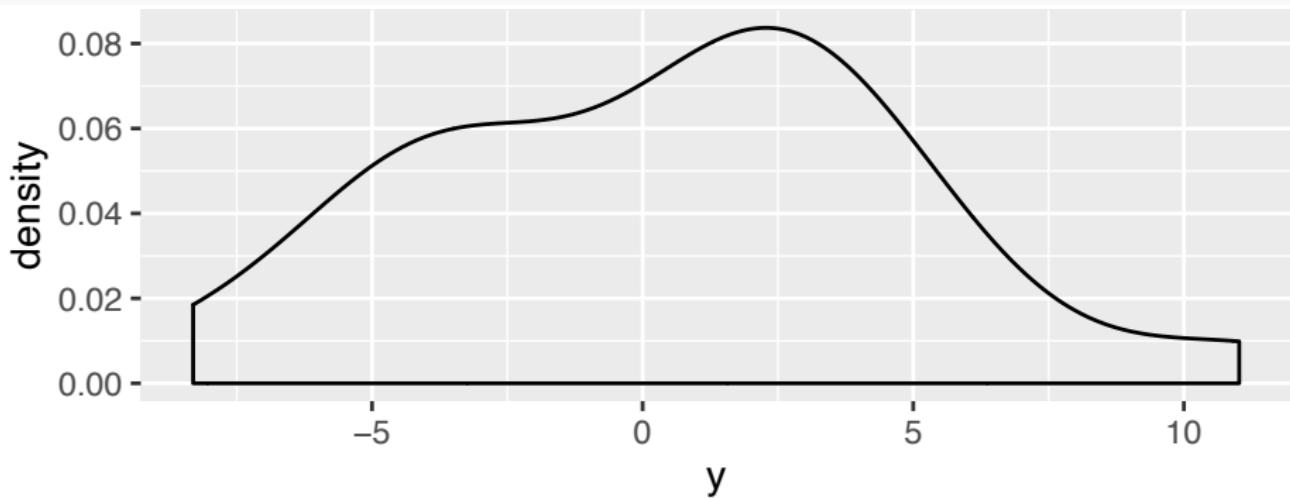
```
library(tidyverse)

## -- Attaching packages -----
## v ggplot2 3.2.1      v purrr    0.3.2
## v tibble   2.1.3      v dplyr    0.8.3
## v tidyr     1.0.0      v stringr  1.4.0
## v readr     1.3.1      vforcats  0.4.0

## -- Conflicts -----
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Random numbers

```
x1 <- runif(20, min=0, max=2) # random uniform with bounds  
x2 <- rnorm(20) # random normal with mean 0, sd 1  
y <- x1 + x2 * rnorm(20, mean=5, sd=2)  
ggplot(tibble(y), aes(x=y)) + geom_density()
```



Basic statistics

```
for (f in c(min, max, mean, median, sd, var, IQR, mad)) {  
  print(f(y))  
}  
  
## [1] -8.304924  
## [1] 11.02246  
## [1] 0.4374315  
## [1] 1.325795  
## [1] 4.580337  
## [1] 20.97949  
## [1] 5.971987  
## [1] 4.859902
```

More data summaries

```
for (f in c(range, quantile, summary, fivenum)) {  
  print(f(y))  
}  
  
## [1] -8.304924 11.022464  
##      0%      25%      50%      75%      100%  
## -8.304924 -3.058389  1.325795  2.913598 11.022464  
##   Min. 1st Qu. Median   Mean 3rd Qu.   Max.  
## -8.3049 -3.0584  1.3258  0.4374  2.9136 11.0225  
## [1] -8.304924 -3.309386  1.325795  2.951253 11.022464
```

Correlations

```
cor(x1, y) # get the correlation  
## [1] 0.295735
```

Correlations: statistical test

```
cor.test(x1, y) # run a test

##
## Pearson's product-moment correlation
##
## data: x1 and y
## t = 1.3134, df = 18, p-value = 0.2055
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## -0.1688880  0.6528217
## sample estimates:
##       cor
## 0.295735
```

Correlations: using results

```
res = cor.test(x1, y) # save the result
names(res) # see result structure

## [1] "statistic"    "parameter"    "p.value"      "estimate"
## [6] "alternative"  "method"       "data.name"    "conf.int"

R = res['statistic'] # extract values...
p = res['p.value'] # ...from the result
```

Null hypothesis significance testing

Independent 2-sample 2-sided t-test

Test whether difference in means is different from 0

```
t.test(x1, y)

##
##  Welch Two Sample t-test
##
## data: x1 and y
## t = 0.61544, df = 19.607, p-value = 0.5453
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -1.520858 2.791563
## sample estimates:
## mean of x mean of y
## 1.0727840 0.4374315
```

More t-tests

The `t.test` function also has options for:

- 1-sided tests
- paired tests
- 1-sample tests.

Regression models

The `lm` (linear model) function and variants are used for regression.

```
df = tibble(x1, x2, y)
head(df)

## # A tibble: 6 x 3
##       x1      x2     y
##   <dbl>  <dbl> <dbl>
## 1 1.94 -0.220  1.07
## 2 1.62  0.702  5.72
## 3 0.514  0.505  2.88
## 4 1.27  0.384  4.29
## 5 0.475 -0.964 -4.36
## 6 0.132 -0.512 -2.81
```

Formulas

R provides a special formula syntax involving the tilde ~. It's used to specify a regression model. The left-hand side is the dependent variable, y. The right-hand side gives the independent variables, interactions, and transformations. So, ~ means something like "is modelled as".

$y \sim x_1 + x_2$

This says: run the formula $y = a + b_1x_1 + b_2x_2$

Using a formula in a regression

```
res <- lm(y ~ x1 + x2, data=df)
summary(res) # show results

##
## Call:
## lm(formula = y ~ x1 + x2, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0111 -0.6745  0.2500  0.6999  2.7058
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -0.7509    0.7119  -1.055   0.3063    
## x1          1.2890    0.5867   2.197   0.0422 *  
## x2          4.5939    0.3722  12.343  6.53e-10 *** 
## ---
```

Formulas with interaction

If we changed + to *, we would add the interaction effect, ie we would run the formula

$$y = a + b_1x_1 + b_2x_2 + b_{12}x_1x_2$$

Use ?formula for more on this special syntax.

Formulas with interaction

```
res <- lm(y ~ x1 * x2, data=df)
summary(res) # show results

##
## Call:
## lm(formula = y ~ x1 * x2, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.6828 -0.4217  0.2195  0.7291  2.2952
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -0.6443    0.7051  -0.914   0.3744    
## x1          1.2612    0.5774   2.184   0.0442 *  
## x2          5.7138    0.9635   5.930 2.11e-05 *** 
## x1:x2     -1.0724    0.8535  -1.257   0.2270    
##
```

Formulas with transformation

We could also use transformations. For example:

```
res <- lm(y ~ x1 + log(x2), data=df)
## Warning in log(x2): NaNs produced
summary(res) # show results

##
## Call:
## lm(formula = y ~ x1 + log(x2), data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.9823 -1.7985 -0.0445  0.8793  4.2214
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 4.4794    2.0797   2.154   0.0747 .
## x1          0.9525    1.6607   0.514   0.6257
```

One-way analysis of variance (ANOVA)

Like t-test for multiple groups, again using a formula.

```
res = aov(height ~ gender * species, data=dplyr::starwars)
summary(res)
```

```
##                                Df Sum Sq Mean Sq F value    Pr(>F)
## gender                  3   1674   558.1   8.421 0.000254 ***
## species                 34  73457  2160.5  32.599 < 2e-16 ***
## gender:species          2     196     97.9   1.477 0.242539
## Residuals                34   2253    66.3
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
## 13 observations deleted due to missingness
```

Beyond Base R: the caret package

- k-nearest neighbours
- Linear regression
- Support vector machines
- Classification/regression trees
- Perceptrons
- Ensembles, including forests, bagging, boosting

<https://topepo.github.io/caret>

The caret package

The main Python competitor is `scikit-learn` which we will study later.

We won't go into detail on ML algorithms in this class.

Further reading

- <https://www.statmethods.net/stats/ttest.html>
- <https://www.statmethods.net/stats/regression.html>
- <https://www.statmethods.net/stats/anova.html>

Exercises

- 1 In the mpg dataset (part of the tidyverse), calculate the mean and standard deviation of the highway fuel efficiency.
- 2 Using group_by, calculate the mean and standard deviation of the highway fuel efficiency per manufacturer.
- 3 Calculate the correlation between highway fuel efficiency and engine size.
- 4 What was the average highway fuel efficiency in 1999 and in 2008?
- 5 Carry out a two-sample independent t-test between highway fuel efficiency in 1999 and 2008 and interpret the result.
- 6 Carry out a regression on highway fuel efficiency by displacement.

Solution 1

```
library(tidyverse)
```

```
mean(mpg$hwy)
```

```
## [1] 23.44017
```

```
sd(mpg$hwy)
```

```
## [1] 5.954643
```

Solution 2

```
mpg %>% group_by(manufacturer) %>%
  summarise(mean=mean(hwy), sd=sd(hwy))

## # A tibble: 15 x 3
##   manufacturer     mean     sd
##   <chr>        <dbl>  <dbl>
## 1 audi         26.4   2.18
## 2 chevrolet    21.9   5.11
## 3 dodge        17.9   3.57
## 4 ford          19.4   3.33
## 5 honda         32.6   2.55
## 6 hyundai       26.9   2.18
## 7 jeep          17.6   3.25
## 8 land rover    16.5   1.73
## 9 lincoln       17     1
## 10 mercury      18     1.15
## 11 nissan        24.6   5.09
```

Solution 3

```
cor(mpg$hwy, mpg$displ)
```

```
## [1] -0.76602
```

Solution 4

```
mpg %>% group_by(year) %>%  
  summarise(mean=mean(hwy), sd=sd(hwy))
```

```
## # A tibble: 2 x 3  
##   year   mean     sd  
##   <int> <dbl> <dbl>  
## 1 1999   23.4   6.08  
## 2 2008   23.5   5.85
```

Solution 5

```
mpg1999 <- mpg %>% filter(year == 1999)
mpg2008 <- mpg %>% filter(year == 2008)
t.test(mpg1999$hwy, mpg2008$hwy)

##
##  Welch Two Sample t-test
##
## data: mpg1999$hwy and mpg2008$hwy
## t = -0.032864, df = 231.64, p-value = 0.9738
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -1.562854 1.511572
## sample estimates:
## mean of x mean of y
## 23.42735 23.45299
```

Solution 6

```
res = lm(hwy ~ displ, data=mpg)
summary(res)

##
## Call:
## lm(formula = hwy ~ displ, data = mpg)
##
## Residuals:
##       Min     1Q Median     3Q    Max
## -7.1039 -2.1646 -0.2242  2.0589 15.0105
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 35.6977    0.7204   49.55 <2e-16 ***
## displ      -3.5306    0.1945  -18.15 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '
```