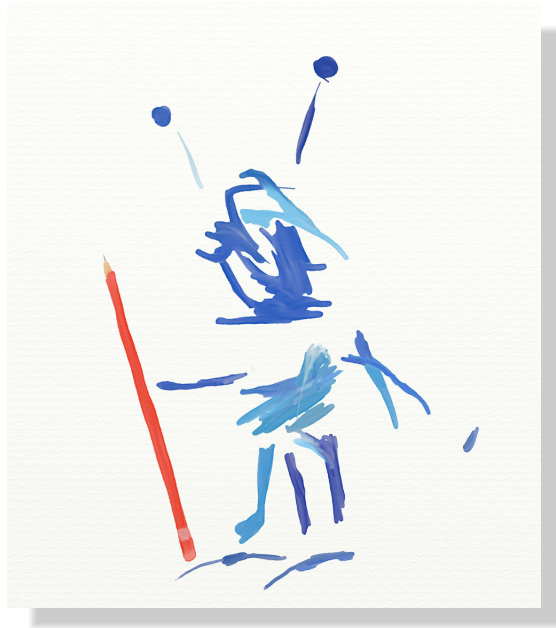


IA Scientific Computing 2019 / 2020

Damon Wischik, Computer Science, Cambridge University



Contents

1	Using Python	1
1.1	A first session	1
1.2	Basic Python expressions	2
1.2.1	Maths and logic	2
1.2.2	Strings and formatting	2
1.3	Collections and control flow	4
1.3.1	Lists and tuples	4
1.3.2	Slicing	5
1.3.3	Dictionaries	5
1.3.4	Control flow	5
1.3.5	Comprehensions *	6
1.4	Python as a programming language *	7
1.4.1	Functions and functional programming	7
1.4.2	Generators	8
1.4.3	None and Maybe, and Enumeration types	8
1.4.4	Dynamic typing	9
1.4.5	Object-oriented programming	10
2	Numerical computation	11
2.1	Preamble	11
2.2	Vectorized thinking	11
2.2.1	A first session	11
2.2.2	Vectorized library routines	12
2.2.3	‘for’ loops considered harmful	13
2.3	Arrays	14
2.4	Numerical optimization and fitting	15
2.5	Simulation	17
3	Working with data	19
3.1	Preamble	19
3.2	What data looks like	19
3.2.1	Missing values	20
3.3	Importing, exporting, and creating dataframes	21
3.4	Selecting and modifying data	22
3.5	Tabulations and indexed arrays	24
3.5.1	Dataframe → indexed array	24
3.5.2	Indexed array → dataframe	25
3.6	Database-style joins	26
3.7	Plotting	27
3.7.1	Basic principles	27
3.7.2	Plot gallery	28

1. Using Python

1.1. A first session

We can use Python interactively like a calculator. Here are some simple expressions and their values. Try entering these yourself, in your own notebook, then press shift+enter or choose Cell | Run Cells from the menu.

```
3 + 8
```

```
1.618 * 1e5
```

```
x = 3
y = 2.2
z = 1
x * y + z
```

```
(x,y,z) = (3, 2.2, 1) # We can assign multiple values at once
x,y,z = 3, 2.2, 1    # (brackets are optional)
x * y + z            # long lines can be split
```

If we want to type in a very long line, we can split it using a backslash.

```
"Few things are more distressing to a well regulated mind " \
+ "than to see a boy who ought to know better, " \
+ "disporting himself at improper moments."
```

Jupyter will only show the output from the last expression in a cell. If we want to see multiple values, print them explicitly.

```
print(x * y + z)
print(x * (y + z))
```

Python does its best to print out helpful error messages. When something goes wrong, look first at the last line of the error message to see what type of error it was, then look back up to see where it happened.

```
x = 'hello'
y = x + 5
y
```

```
1 x = 'hello'
----> 2 y = x + 5
      3 y
```

```
TypeError: must be str, not int
```

1.2. Basic Python expressions

1.2.1. MATHS AND LOGIC

All the usual mathematical operators work, though watch out for division which uses different syntax to Java.

```
7 / 3                # floating point division
7 // 3              # integer division (rounds down)
min(3,4), max(3,4)
abs(-10), abs(3+4j) # 3+4j is a complex number
round(7.4), round(-7.4), round(3.4567, 2)
3**2                # power
5 << 1, 5 >> 2      # bitwise shifting
7 & 1, 6 | 1        # bitwise operations
(3+4j).real, (3+4j).imag # complex numbers
```

The usual logical operators work too, though the syntax is wordier than other languages. Python's truth values are True and False.

```
(x,y,z) = (5, 12, False)
x < y or y < 10        # precedence: (x < y) or (y < 10)
x < y and not y < 15   # precedence: (x < y) and (not (y < 15))
(x == y) == z
'lower' if x < y else 'higher' # same as Java's (x < y) ? 'lower' : 'higher'
```

Some useful maths functions are found in the maths module. To use them you need to run import math. (It's common to put your import statements at the top of the notebook, as they only need to be run once per session, but they can actually appear anywhere.)

```
import math
math.floor(-3.4), math.ceil(-3.4)
math.pow(9, 0.5), math.sqrt(9)
math.exp(2), math.log(math.e), math.log(101, 10)
math.sin(math.pi*1.3), math.atan2(3,4)

import cmath # for functions on complex numbers
cmath.sqrt(-9)
cmath.exp(math.pi * 1j) + 1

import random # for generating random numbers
random.random(), random.random()
```

1.2.2. STRINGS AND FORMATTING

Python strings can be enclosed by either single quotes or double quotes. Strings (like everything else in Python) are objects, and they have methods for various string-processing tasks. See the String Methods documentation¹ for a full list.

```
"shout".upper()        # "SHOUT"
"hitchhiker".replace('hi', 'ma') # "matchmaker"
'i' in 'team'          # False

x = '''
Also, a multi-line string can be
entered with triple-quotes.
'''
```

To control how values are printed, use f-strings i.e. strings with f before the opening quote. Each

¹<https://docs.python.org/3/library/stdtypes.html#string-methods>

chunk of the string enclosed in `{·}` is evaluated, and the result is spliced back into the string.

```
name, age = 'Zaphod', 27
f"My name is {name} and I will be {age+1} next year"
```

The replacement chunk can specify the output format. The documentation² describes more format specifiers.

```
f"The value of  $\pi$  to 3 significant figures is {math.pi:.3}"
```

If you do any serious data processing in Python, you will likely find yourself needing regular expressions³. The appendix illustrates using regular expressions for data cleanup.

```
import re
s = 'In 2019 there will be an election'
re.search('(\d+)', s)[0]           # '2019'
re.sub('a(n?) (\w+)ion', 'calamity', s) # 'In 2019 there will be calamity'
```

²https://docs.python.org/3/reference/lexical_analysis.html#f-strings

³<https://docs.python.org/3/library/re.html>

1.3. Collections and control flow

Python has four common types for storing collections of values: tuples, lists, dictionaries, and sets.

In IA courses on OCaml and Java we learnt about lists versus arrays. In those courses, and in IA Algorithms, we study the efficiency of various implementation choices. In Python, you *shouldn't* think about these things, at least not in the first instance. The Pythonic style is to just go ahead and code, and only worry about efficiency after we have working code. As the famous computer scientist Donald Knuth said,

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

Only when we have special requirements should we switch to a dedicated collection type, such as a deque⁴ or a heap⁵ or the specialized numerical types we'll learn about in section 2.

1.3.1. LISTS AND TUPLES

Python lists and Python tuples are both used to store sequences of elements. They both support iterating over the elements, concatenation, random access, and so on. They're a bit like lists, and a bit like arrays.

```
a = [1, 2, 'buckle my shoe']      # a list
b = (3, 4, 'knock at the door')  # a tuple

len(a), len(b)
a[0], a[1], b[2]                 # indexes start at 0
a[-1], a[-2]                    # negative indexes count from the end
[a, 'then', b]
3 in a, 3 in b                   # is this item contained in the collection?

a + list(b)                      # l1+l2 concatenates two lists
tuple(a) + b                     # t1+t2 concatenates two tuples
list(zip(a,b))                   # zip(l1,l2) gives [(l1[0],l2[0]), (l1[1],l2[1]), ...]
```

As you see, both lists and tuples can hold mixed types, including other lists or tuples. You can convert a list to a tuple and vice versa, and extract elements. The difference is that lists are mutable

```
a[0] = 5
a.append('then')
a.extend(b)
a
[5, 2, 'buckle my shoe', 'then', 3, 4, 'knock at the door']
```

whereas tuples are immutable.

```
b[0] = 5

4 print(a)
5
----> 6 b[0] = 5 # error
7 print(b)

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

To sort a list, we have a choice between sorting in-place or returning a new sorted list without changing the original.

```
names = ['bethe', 'alpher', 'gamov']
```

⁴<https://docs.python.org/3/library/collections.html#collections.deque>

⁵<https://docs.python.org/3/library/heapq.html>

```
sorted(names)      # ['alpher', 'bethe', 'gamov'], returns a new list
names              # ['bethe', 'alpher', 'gamov'], unchanged from before

names.sort()
names              # ['alpher', 'bethe', 'gamov'], sorted in-place
```

Another common operation is to concatenate a list of strings. Python's syntax for this is unusual:

```
', '.join(names) + ' wrote a famous paper on nuclear physics'
' alpher, bethe, gamov wrote a famous paper on nuclear physics'
```

1.3.2. SLICING

We can pick out subsequences using the *slice* notation, `x[start:end:sep]`.

```
x = list(range(10))      # [0,1,2,3,4,5,6,7,8,9]

x[1:3]                  # start is inclusive and end is exclusive, so x[1:3] == [x[1],x[2]]
x[:2]                   # first two elements
x[2:]                   # everything after the first two
x[-3:]                  # last three elements
x[:-3]                  # everything prior to the last three
x[::4]                  # every fourth element
```

We can assign into slices.

```
x[::4] = [None, None, None]
x
[None, 1, 2, 3, None, 5, 6, 7, None, 9]
```

1.3.3. DICTIONARIES

The other very useful data type is the dictionary, what Java calls a Map or HashMap.

```
room_alloc = {'Adrian': None, 'Laura': 32, 'John': 31}
room_alloc['Guarav'] = 19      # add or update an item
del room_alloc['John']         # remove an item

room_alloc['Laura']            # get an item
room_alloc.get('Alexis', 1)    # get item if it exists, else default to 1
'Alexis' in room_alloc         # does this dictionary contain the key 'Alexis'?
```

To iterate over items in a dictionary, see the next example...

1.3.4. CONTROL FLOW

Python supports the usual control flow statements: for, while, continue, break, if, else.

To iterate over items in a list,

```
for item in list:
    ... # do something with item
```

To iterate over items and their positions in the list together,

```
for i, name in enumerate(['bethe', 'alpher', 'gamov']):
    print(f"Person {name} is in position {i}")
```

To just do something a number of times, if we don't care about the index, it's conventional to call the loop variable `_`.

```
x = 2
```

```
for _ in range(5):
    x *= 2
```

To iterate over two lists simultaneously, zip them.

```
for x,y in zip(['apple', 'orange', 'grape'], ['cheddar', 'wensleydale', 'brie']):
    print(f"{x} goes with {y}")
```

We can also iterate over (key,value) pairs in a dictionary. Suppose we're given a list of room allocations, and we want to find the occupants of each room.

```
room_alloc = {'adrian': 10, 'chloe': 5, 'guarav': 10, 'shay': 11,
              'alexis': 11, 'rebecca': 10, 'zubin': 5}

occupants = {}
for name, room in room_alloc.items(): # iterate over keys and values
    if room not in occupants:
        occupants[room] = []
    occupants[room].append(name)

for room, occupants_here in occupants.items():
    ns = ', '.join(occupants_here)
    print('Room {r} has {ns}'.format(r=room, ns=ns))
```

We can also iterate over just the keys (for name in room_alloc), or just the values (for room in room_alloc.values()).

1.3.5. COMPREHENSIONS *

Python has a distinctive piece of syntax called a *comprehension* for creating lists. It's a very common pattern to write code that transforms lists, e.g.

```
ℓ = ... # start with some list [ℓ0, ℓ1, ...]
f = ... # some function we want to apply, to get [f(ℓ0), f(ℓ1), ...]
res = []
for i in range(len(ℓ)):
    x = ℓ[i]
    y = f(x)
    res.append(y)
```

This is so common that Python has special syntax for it,

```
res = [f(x) for x in ℓ]
```

There's also a version which only keeps elements that meet a criterion,

```
res = [f(x) for x in ℓ if t]
```

Here's a concrete example:

```
xs = range(10)
[x**2 for x in xs if x % 2 == 0]
```

We can also use comprehension to create dictionaries and sets. Here's a dictionary:

```
{x: x**2 for x in xs}
```


1.4. Python as a programming language *

This section of the notes is to compare and contrast the Python language to what you have learnt in the courses so far using OCaml and Java. This section of the course is here for your general interest, and it's not needed for the Scientific Computing course, apart from section 1.4.1 on defining functions.

The development of the Python language is documented in *Python Enhancement Proposals* (PEPs)⁶. Significant changes in the language, or in the standard libraries, are discussed in mailing lists and written up for review as a PEP. They typically suggest several ways a feature might be implemented, and give the reason for choosing one of them. If consensus is not reached to accept the PEP, then the reasons for its rejection are also documented. They are fascinating reading if you are interested in programming language design.

1.4.1. FUNCTIONS AND FUNCTIONAL PROGRAMMING

The code snippet below shows how we define a function in Python. There are several things to note:

- The function is defined with a default argument, `c=0`. You can invoke it by either `roots(2,3,1)` or `roots(2,3)`.
- Functions can be called with named arguments, `roots(b=3, a=2)`, in which case they can be provided in any order.

In scientific computing, we'll come across many functions that accept 10 or more arguments, all of them with sensible defaults, and typically we'll only specify a few of the arguments. This is why defaulting and named arguments are so useful.

```
import math

def roots(a, b, c=0):
    """Return a list with the real roots of c*(x**2) + b*x + a == 0"""
    if b == 0 and c == 0:
        raise Exception("This polynomial is constant")
    if c == 0:
        return [-a/b]
    elif a == 0:
        return [0] + roots(b=c, a=b)
    else:
        discr = b**2 - 4*c*a
        if discr < 0:
            return []
        else:
            return [(-b+s*math.sqrt(discr))/2/c for s in [-1,1]]
```

Some more notes:

- This function either returns a value, or it throws an exception i.e. generates an error message and finishes. If your function finishes without an explicit return statement, it will return `None`. Unlike Java, it's possible for different branches of your function to return values of different types — at risk to your sanity.
- This function returns a single variable, namely a list. If you want to return several variables, return them in a tuple, and unpack the tuple using multiple assignment as shown in section 1.1.
- It's conventional to document your function by providing a documentation string as the first line. You can see help for a function with `?`. If we run `?roots` we're shown

```
Signature: roots(a, b, c=0)
Docstring: Return a list with the real roots of c*(x**2) + b*x + a == 0
File:      /path_to_notebook/<ipython-input-53-6cf3a0af9585>
Type:      function
```

In Python as in OCaml, functions can be returned as results, assigned, put into lists, passed as arguments to other functions, and so on.

See OCaml lecture 8

⁶<https://www.python.org/dev/peps/>

```
import random

def noisifier( $\sigma$ ):
    def add_noise(x):
        return x + random.uniform(- $\sigma$ ,  $\sigma$ )
    return add_noise

fs = [noisifier( $\sigma$ ) for  $\sigma$  in [0.1, 1, 5]]

[f(1.5) for f in fs]
```

```
[1.5041, 1.0309, 6.0885]
```

In this example above, `noisifier` is a function that returns another function. The inner function ‘remembers’ the value of σ under which it was defined; this is known as a *closure*.

We can use `lambda` to define anonymous functions, i.e. functions without names. This often used to fill in arguments.

```
def illustrate_func(f, xs):
    for x in xs:
        print(f"f({x}) = {f(x)}")

illustrate_func(lambda b: roots(1,b,2), range(5))
```

```
f(0) = []
f(1) = []
f(2) = []
f(3) = [-1.0, -0.5]
f(4) = [-1.70710, -0.29289]
```

1.4.2. GENERATORS

A generator (or lazy list, or sequence) is a list where the elements are only computed on demand. This lets us implement infinite sequences. In Python, we can create them by defining a function that uses the `yield` statement:

```
def fib():
    x,y = 1,1
    while True:
        yield x
        x,y = (y, x+y)

fibs = fib()
[next(fibs) for _ in range(10)]
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

When we call `next(fibs)`, the `fib` code runs through until it reaches the next `yield` statement, then it emits a value and pauses. Think of `fibs` as an execution pointer and a call stack: it remembers where it is inside the `fib` function, and calling `next` tells it to resume executing until the next time it hits `yield`.

We can also transform generators using syntax a bit like list comprehension:

```
even_fibs = (x for x in fib() if x % 2 == 0)
[next(even_fibs) for _ in range(10)]
```

```
[2, 8, 34, 144, 610, 2584, 10946, 46368, 196418, 832040]
```

1.4.3. NONE AND MAYBE, AND ENUMERATION TYPES

It’s often handy for functions to be able to return either a value, or a marker that there is no value. For example, `head(list)` should return a value unless the list is empty in which case there’s nothing to return. A common pattern in a language like OCaml is to have a datatype that explicitly supports this, for example we’d define `head` to return an enumeration datatype with a constructor function, `None | Some[a]`. This forces everyone who uses `head` to check whether or not the answer is `None`.

In Python, the return type of a function isn't constrained. It's a common convention to return `None` if you have nothing to return, and a value otherwise, and to trust that the person who called you will do the appropriate checks.

Enumeration types are also used for type restriction, e.g. to limit what can be placed in a list. When we actually do want to achieve this, Python isn't much help. It does have an add-on library for enumeration types⁷ but it's a lot of work for little benefit.

One situation where enumeration types are very useful is when working with categorical values in data. When working with data, the levels of the enumeration are decided at runtime (by the contents of the data we load in), so pre-declared types are no use anyway.

1.4.4. DYNAMIC TYPING

Python uses *dynamic typing*, which means that values are tagged with their types during execution and checked only then. To illustrate, consider the functions

```
def double_items(xs):
    return [x*2 for x in xs]
def goodfunc():
    return double_items([1,2,[3,4]]) + double_items("hello world")
def badfunc():
    return double_items(10)
```

We won't be told of any errors until `badfunc()` is invoked, even though it's clear when we define it that `badfunc` will fail.

Python programmers are encouraged to use *duck typing*, which means that you should test values for what they can do rather than what they're tagged as. "If it walks like a duck, and it quacks like a duck, then it's a duck". In this example, `double_items(xs)` iterates through `xs` and applies `*2` to every element, so it should apply to any `xs` that supports iteration and whose elements all support `*2`. These operations mean different things to different types: iterating over a list returns its elements, while iterating over a string returns its characters; doubling a number is an arithmetical operation, doubling a string or list repeats it. Python does allow you to test the type of a value with e.g. `if isinstance(x, list): ...`, but programmers are encouraged not to do this.

Python's philosophy is that library designers are providing a service, and programmers are adults. If a library function uses comparison and addition, and if the end-user programmer invents a new class that supports comparison and addition, then why on earth shouldn't the programmer be allowed to use the library function? (I've found this useful for simulators: I replaced 'numerical timestamp' with 'rich timestamp class that supports auditing, listing which events depended on which other events', and I didn't have to change a single line of the simulator body.) Some statically typed languages like Haskell and Scala support this via *dynamic type classes*, but their syntax is rather heavy.

To make duck typing useful, Python has a long list of special method names⁸ so that you can create custom classes supporting the same operations as numbers, or as lists, or as dictionaries. For example, if you define a new class with the method `__iter__`⁹ then your new class can be iterated over just like a list.

Example: trees. Suppose we want to define a tree whose leaves are integers and whose branches can have an arbitrary number of children. Actually, in Python, there's nothing to define: we can just start using it, using a list to denote a branch node.

```
x = [1, [[2, 4, 3], 9], [5, [6, 7], 8]]
```

To flatten a list like this we can use duck typing: given a node `n`, try to iterate over its children, and if this fails then the node must be a leaf so just return `[n]`.

```
def flatten(n):
    try:
        return [y for child in n for y in flatten(child)]
    except TypeError as e:
        return [n]
flatten(x)
```

⁷<https://docs.python.org/3/library/enum.html>

⁸<https://docs.python.org/3/reference/datamodel.html#special-method-names>

⁹https://docs.python.org/3/reference/datamodel.html#object.__iter__

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

This would work perfectly well for trees containing arbitrary types — unless the end-user programmer puts in leaves which are themselves iterable, in which case the duck typing test doesn't work — unless that is the user's intent all along, to be able to attach new custom sub-branches ...

A solution is to define a custom class for branch nodes, and use `isinstance` to test each element to see if it's a branch node. This is not very different to the OCaml solution, which is to declare nodes to be of type 'either leaf or branch' — except that Python would still allow leaves of arbitrary mixed type.

1.4.5. OBJECT-ORIENTED PROGRAMMING

Python is an object-oriented programming language. Every value is an object. You can see the class of an object by calling `type(x)`. For example,

```
x = 10
type(x)    # reports int
dir(x)     # gives a list of x's methods and attributes
```

It supports inheritance and multiple inheritance, and static methods, and class variables, and so on. It doesn't support interfaces, because they don't make sense in a duck typing language.

Here's a quick look at a Python object, and at how it might be used for the flatten function earlier.

```
class Branch(object):
    def __init__(self, children):
        self.children = children

def flatten(n):
    if isinstance(n, Branch):
        return [y for child in n.children for y in flatten(child)]
    else:
        return [n]

x = Branch([10, Branch([3, 2]), "hello"])
flatten(x)
```

Every method takes as its first argument a variable referring to the current object, 'this' in Java. Python doesn't support private and protected access modifiers, except by convention: the convention is that attributes and functions whose name begins with an underscore are considered private, and may be changed in future versions of the library.

The next lines of code are surprising. You can 'monkey patch' an object, after it has been created, to change its attributes or give it new attributes. Like so many language features in Python, this is sometimes tremendously handy, and sometimes the source of infuriating bugs.

```
y = Branch([])
y.my_label = "added an attribute"
```

2. Numerical computation

Working with numbers is central to almost all scientific and engineering computing, from deep learning to image processing to climate simulation.

We learnt about Python lists in section 1. We could just use lists to store numbers: a list to store a vector, a list of lists to store a matrix, and so on. Python lists can store mixed data types e.g. integers mixed with strings and sublists and even functions — and this flexibility comes with a price:

- Flexible lists are slow. For scientific computing, it's better to use specialised classes for numeric datatypes. If the machine knows what datatypes to expect, it can compute faster.
- Flexible lists are cumbersome. Mathematicians have good notation for vector algebra, and if we can write code to match then it'll be more concise and easier to debug.
- Putting these two points together... there are many carefully-optimized low-level libraries for doing all the numerical heavy lifting. We'd like to be able to write code at a high level, and rely on the machine to use whatever resources it has to compute efficiently. It might be a GPU, or it might be a cluster of machines in the cloud: we should be able to express our high-level intention, and leave the machine to figure out the best way to achieve it. In effect, we're using Python just as 'glue' to link together high-level syntax with low-level libraries.

The core skill is *vectorized thinking*, which means writing our code in terms of functions that operate on entire vectors (or arrays) at once. Once you get the hang of it, you will write code that is more concise and faster.

2.1. Preamble

There are two main Python libraries for numerical work, NumPy¹⁰ and SciPy¹¹. It's vital to be familiar with these, especially NumPy, for almost any work in machine learning. At the top of almost every piece of scientific computing work, we'll import these standard modules. We'll also give them short aliases so we can write e.g. `np.foo` rather than `numpy.foo`.

```
import numpy as np
import matplotlib.pyplot as plt

# and some others, not quite so universally used
import math, random
import scipy
import scipy.optimize
```

2.2. Vectorized thinking

2.2.1. A FIRST SESSION

```
x = np.array([1,2,5,3,2]) # [1, 2, 5, 3, 2]
y = np.ones(5)           # [1.,1.,1.,1.,1.]
z = np.arange(5)         # [0, 1, 2, 3, 4]

# We can do maths on vectors, applying the operations elementwise
# We can mix vectors and scalars
(x + 1) * 2              # [4, 6, 12, 8, 6]
(x + y) * z              # [0., 3.,12.,12.,12.]

x >= 3                   # [False, False, True, True, False]
np.where(x>=3, x, 3)     # [3, 3, 5, 3, 3]

# Python slicing and indexing notation works
x[:3]                    # [1, 2, 5]
x[x>=3] = -10           # x is now [1, 2, -10, -10, 2]
```

¹⁰<http://www.numpy.org/>

¹¹<https://www.scipy.org/>

All the elements of a numpy vector have to be the same type. It can be integer, or floating point, or boolean. To see the type of a vector, `x.dtype`. To cast to another type, `(y>=3).astype(int)`.

2.2.2. VECTORIZED LIBRARY ROUTINES

To be good at writing vectorized code, we need to know what sorts of calculations we can do on vectors. Here are some useful routines¹²:

To create vectors,

- `np.array([1,2,3])` creates a numpy vector from a Python list
- `np.zeros(n)`, `np.ones(n)`, `numpy.full(n,fill_value)`
- `numpy.ones_like(a)` creates a vector of the same shape as a
- `np.arange` is like Python's range
- `np.linspace(start,stop,n)` creates n evenly-spaced points between start and stop inclusive, very useful for plotting
- `np.random.random(n)`, `np.random.choice(a,n)`, and other random number generators¹³
- see also the many other array creation routines¹⁴

For maths,

- Normal mathematical expressions work on vectors, and you can mix vectors and scalars
- `np.sin`, `np.exp`, `np.floor`, ...
- `x y` gives the dot product, `np.linalg.norm(x)` is the norm
- `np.sum`, `np.mean`, and `np.prod`; also `np.cumsum(x)` gives $[x_0, x_0 + x_1, x_0 + x_1 + x_2, \dots]$
- `np.min` and `np.max` for the overall min and max; `np.minimum(x,y)` for $[\min(x_0, y_0), \min(x_1, y_1), \dots]$
- and many other maths¹⁵ and statistics¹⁶ functions.

Here's a more elaborate example: computing the correlation coefficient¹⁷ between two vectors x and y ,

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2} \sqrt{\sum_i (y_i - \bar{y})^2}}$$

where x and y have the same length N , and

$$\bar{x} = \frac{1}{N} \sum_i x_i, \quad \bar{y} = \frac{1}{N} \sum_i y_i.$$

Here are two implementations, one written in Python-style, one written in scientific computing style¹⁸, to compute ρ . The latter is roughly 15 times faster. (The magic command¹⁹ `%%time` at the start of a cell makes the notebook print out the execution time.)

```
# Set up some parameters.
# We'll seed the random number generators so our test is reproducible.
N = 10000000
rand_seed = 1618033988

%%time
# Python-style code

random.seed(rand_seed)
```

¹²<https://docs.scipy.org/doc/numpy/reference/routines.html#routines>

¹³<https://docs.scipy.org/doc/numpy/reference/routines.random.html>

¹⁴<https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html#array-creation-routines>

¹⁵<https://docs.scipy.org/doc/numpy/reference/routines.math.html#mathematical-functions>

¹⁶<https://docs.scipy.org/doc/numpy/reference/routines.statistics.html>

¹⁷https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

¹⁸Of course, if we really knew our way round numpy, we'd just use `np.corrcoef`, <https://docs.scipy.org/doc/numpy/reference/generated/numpy.corrcoef.html#numpy.corrcoef>

¹⁹<http://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-time>

```
# Create two lists of random numbers, xs and ys
xs = [random.random() for i in range(N)]
ys = [xs[i] + random.random() for i in range(N)]
# Compute the various terms involved in the formula
xbar = sum(xs) / N # sum(list) is built into Python
ybar = sum(ys) / N
sxy = sum([(x-xbar)*(y-ybar) for x,y in zip(xs,ys)])
sxx = sum([(x-xbar)**2 for x in xs])
syy = sum([(y-ybar)**2 for y in ys])
ρ = sxy / math.sqrt(sxx) / math.sqrt(syy)
print(ρ)
```

```
0.707063527537949
CPU times: user 5.34 s, sys: 2.78 s, total: 8.12 s
Wall time: 8.25 s
```

```
%%time
# Vectorized code

np.random.seed(rand_seed)
# Create two random vectors x and y
x = np.random.random(N)
y = x + np.random.random(N)
# Compute the terms in the formula. Note: @ means "dot product"
xbar = np.sum(x) / N
ybar = np.sum(y) / N
ρ = ((x-xbar) @ (y-ybar)) / math.sqrt(np.sum((x-xbar)**2) * np.sum((y-ybar)**2))
print(ρ)
```

```
0.707201664199036
CPU times: user 359 ms, sys: 797 ms, total: 1.16 s
Wall time: 899 ms
```

2.2.3. 'FOR' LOOPS CONSIDERED HARMFUL

Vectorized thinking isn't just for mathematical formulae—there are all sorts of programming constructs that can be vectorized also. In general, whenever you find yourself writing a for loop or a Python list comprehension, stop and see if you can vectorize your code. You'll usually end up with something more flexible for scientific computing.

[list comprehension](#)
page 6

Functions for indexing:

- the usual slice notation works on numpy vectors, e.g. $x[:10]$ or $x[10:]$ or $x[:-3]$
- `np.where(b)` gives a vector of indexes at which the boolean vector `b` is True
- index a vector using a vector of booleans, e.g. $x[y>5]$
- index a vector using a vector of integers, e.g. $i=np.where(y>5)$; $x[i]$
- update part of a vector, e.g. $x[x<3] = x[x<3] + 10$
- `np.concatenate([v1,v2])` concatenates two or more vectors

[Slice notation](#), page 5

General programming functions:

- `len(x)` gives the length of a vector
- `np.any`, `np.all`, and other logic functions²⁰
- `np.unique` returns the unique elements of a vector
- $\sim x$ is logical negation, the equivalent of Python's `not x`; also $(x \& y)$ and $(x | y)$ both work
- `np.count_nonzero(x)` counts the number of entries where `x` is True or non-zero
- `x.sort()` sorts a vector in-place, and `np.sort(x)` creates a new vector which is a sorted version of `x`

²⁰<https://docs.scipy.org/doc/numpy/reference/routines.logic.html#logic-functions>

- `np.argsort(x)` gives the vector of indexes that would put `x` in order, i.e. it produces an integer vector `i` such that `x[i]` is sorted; also see other sorting functions²¹
- `np.argmax` and other search functions²²
- `np.where(cond,x,y)` is the vectorized version of Python's `x if cond else y`
- `np.vectorize(f)` is a vectorized version of an arbitrary Python function `f`

String functions:

- numpy does have some string functions—but as its name suggests the library is oriented around numbers not strings, and I recommend using Python strings and list comprehensions, and just wrapping your answer up as a vector with `np.array`.

Here's an example. Suppose we want to sort a vector of strings by length, breaking ties alphabetically.

1. Get a vector with the length of each string. I won't bother looking for numpy routines to handle strings, I'll just use Python.
2. Work out how to put lengths in order, breaking ties alphabetically by names. Digging around the documentation for sorting, we find `np.lexsort([y,x])`, which returns sorting indexes like `np.argsort(x)`, but breaks ties in `x` by another vector `y`. This is called lexicographical sorting.
3. Pick out the names in the order specified by these indexes.

```
names = np.array(['alexis', 'chloe', 'guarav', 'shay', 'adrian', 'rebecca'])
lengths = np.array([len(x) for x in names]) # or np.vectorize(len)(names)
i = np.lexsort([names, lengths])
names[i]
```

2.3. Arrays

NumPy supports matrices and higher-dimensional arrays. To enter a 2d array (i.e. a matrix) like

$$a = \begin{bmatrix} 2.2 & 3.7 & 9.1 \\ -4 & 3.1 & 1.3 \end{bmatrix}$$

we type in

```
a = np.array([[2.2, 3.7, 9.1], [-4, 3.1, 1.3]])
```

Use `a.shape` to find the dimensions of an array. In fact, vectors are nothing other than one-dimensional arrays, and their shape is a tuple of length 1.

```
a.shape # (2,3)
x = np.array([5,6,4])
x.shape # (2,)
```

NumPy doesn't have any concept of 'row vector' versus 'column vector'. It's only 2d arrays that can be described in that way.

```
np.array([[5,6,4]]).shape # (1,3), i.e. a row
np.array([[5],[6],[4]]).shape # (3,1), i.e. a column
```

To refer to a subarray, we can use an extended version of Python's slice notation.

```
a[:, :2] # all rows, first two columns
a[1, :] # second row (indexes start at 0), all columns
a[1] # another way to fetch the second row
a[:2, :2] = [[1,2],[3,4]] # assign to a submatrix
```

There are two ways to refer to an arbitrary set of elements inside the array, both called advanced indexing²³.

²¹<https://docs.scipy.org/doc/numpy/reference/routines.sort.html#sorting>

²²<https://docs.scipy.org/doc/numpy/reference/routines.sort.html#searching>

²³<https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#advanced-indexing>


```

a = np.zeros((3,3))

# Boolean advanced indexing
i = np.array([[False,True,False],[True,False,False],[False,False,True]])
a[i] = [6,7,8]

# Integer advanced indexing
ix,iy = [0,1,2], [1,0,2]
a[ix,iy] = [6,7,8]

# Both produce the same result:
array([[0., 6., 0.],
       [7., 0., 0.],
       [0., 0., 8.]])

```

For 1d vectors the only reshaping operations are slicing and concatenating, but for higher dimensional arrays there is a whole variety of reshaping functions²⁴ such as stacking, tiling, transposing, etc. The most useful operation is adding a new dimension, for example to turn a one-dimensional vector into a column vector. The second most useful is stacking vectors to form an array.

```

x = np.array([1,2,3])
x[:, np.newaxis]

array([[1],
       [2],
       [3]])

np.column_stack([[1,2], [3,4], [5,6]])

array([[1, 3, 5],
       [2, 4, 6]])

```

NumPy also has a powerful tool called broadcasting²⁵ which generalizes ‘add a scalar to a vector’, and which is used a lot in more advanced array-manipulating code. It’s more advanced than we need for this course, but it’s used a lot in machine learning and it’s worth reading about. Here’s a simple example, normalizing a matrix so the columns sum to 1.

```

a = np.array([[3,2,8],[2,6,2]])
colsums = np.sum(a, axis=0)
a / colsums

array([[0.6 , 0.25, 0.8 ],
       [0.4 , 0.75, 0.2 ]])

```

In Easter term you will study linear algebra in the *Maths for Natural Sciences* course. If you want to try out the maths, you’ll find relevant functions in `np.linalg`²⁶ and `np.dual`²⁷.

2.4. Numerical optimization and fitting

A common task in science and in machine learning is to find the minimum value of a function, which may have one or more variables. For example, we might have a collection of points that more or less follow a straight line, and we might want to use the equation $y = mx + c$. In this case, we’d like to tune the values of m and c so that the equation lies close to the data. We can achieve this by defining a function $L(m, c)$ that measures how far the points are from the straight line, and then choosing m and c to minimize $L(m, c)$.

⚠ WARNING! The methods we discuss here sometimes work brilliantly, but sometimes are unstable. This is not the fault of Python or the libraries we are using. It’s just the case that sometimes the equations in the algorithm and numerical issues in the data are not well balanced. This is something

²⁴<https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html#array-manipulation-routines>

²⁵<https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>

²⁶<https://docs.scipy.org/doc/numpy/reference/routines.linalg.html>

²⁷<https://docs.scipy.org/doc/numpy/reference/routines.dual.html>

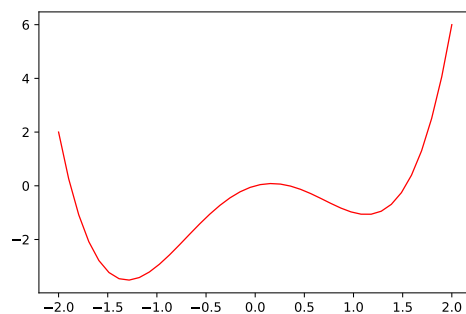
we need to bear in mind every time we use these methods, and we should check the output, for example by plotting graphs.

Let's start by minimizing a simple function of one variable. We could use calculus to find the minimum for a simple example like this, but let's do it with computer power instead.

```
def f(x, a, b, c):
    return a*x + b*(x**2) + c*(x**4)
```

We'll plot this function first, to get a rough idea of where the minimum should be. Visualisation is a crucial part of scientific computing, and we'll cover it in much more detail in section 3, but for present purposes we'll just use some very simple plotting commands. The [matplotlib tutorial](#)²⁸ explains more options.

```
x = np.linspace(-2,2,40) # 40 equally spaced points in the range [-2,2]
y = f(x, a=1, b=-3, c=1) # f is a vectorized expression ... it works on vector x
plt.plot(x, y, linestyle='-', linewidth=1, color='red')
plt.show()
```



The `scipy.optimize.fmin` function finds where the function achieves its minimum value, starting from an initial guess `x0`. The first argument is the function to optimize. In the snippet below we're giving it an anonymous function that is a version of `f` with the parameters `a`, `b`, and `c` filled in.

```
scipy.optimize.fmin(lambda x: f(x,a=1,b=-3,c=1), x0=0.5)
```

```
Optimization terminated successfully.
Current function value: -1.070230
Iterations: 16
Function evaluations: 32
array([1.13085938])
```

It found a local minimum, not the global minimum. This is often a problem with numerical optimization routines, and it's why it's helpful to look at the data first.

Here is an example of a function of two variables. We'll try to fit the straight line $y = mx + c$ through a set of points. We'll define the *loss function*

$$L(m, c) = \sum_i (mx_i + c - y_i)^2$$

and look for m and c to minimize it.

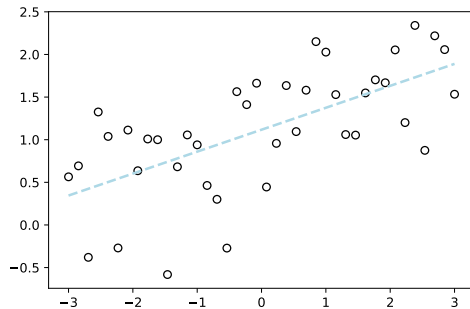
```
# synthetic set of points
x = np.linspace(-3,3,40)
y = np.sin(x) + 2 * np.random.random(x.shape)

def loss(m,c):
    return np.sum((m*x+c - y)**2)

# To optimize a function of several several variables, provide them as an array
# of the appropriate length.
optpars = scipy.optimize.fmin(lambda params: loss(params[0], params[1]),
                               x0 = [0,0])
```

²⁸https://matplotlib.org/users/pyplot_tutorial.html

```
# Plot the datapoints, together with the fitted straight line.
plt.scatter(x, y, facecolor='white', edgecolor='black')
def fit(x): return optpars[0] * x + optpars[1]
plt.plot([-3,3], [fit(-3),fit(3)],
         linestyle='--', color='lightblue', linewidth=2)
plt.show()
```



2.5. Simulation

Simulation is a mainstay of scientific computing. A common style with numpy is to predefine an vector or array to store the results, one row per timestep, and then iterate over timesteps gradually filling in the array. (This is the one case where for loops are appropriate.) Here's an example, a differential equation simulation. A model that has been proposed for TCP²⁹ is

$$\frac{dx_t}{dt} = \frac{1}{RTT^2} - p_t - RTT x_t - \frac{x_t}{2}, \quad p_t = \frac{\max(x_t - C, 0)}{x_t}$$

where x_t is the transmission rate of a sender at time t measured in packets per second, RTT is the round trip time, p_t is the packet drop probability, and C is the link capacity. We might simulate this as follows.

```
x0 = 1          # initial transmission rate, in pkt/sec
C = 10         # link capacity, in pkt/sec
T = 20        # simulated duration in seconds
RTT = 0.2     # round trip time in seconds
dt = 0.01     # timestep size
def P(x): return max(x-C,0) / x

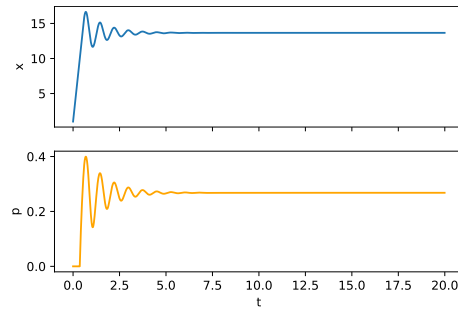
# Initialization
num_iterations = np.ceil(T/dt).astype(int)
res = np.zeros((num_iterations, 3)) # a matrix to store t,x,p
res[0,1:] = [x0, P(x0)]
steps_back = int(RTT/dt)

# Loop
for i in range(1, num_iterations):
    (t,x,p),(xold,pold) = res[i-1], res[max(i-1-steps_back,0),1:]
    dx = 1/(RTT**2) - pold*xold*x / 2
    x = x + dx * dt
    p = P(x)
    res[i] = (t + dt, x, p)

# Plot the output (see section 3 for more about plotting)
fig,(ax1,ax2) = plt.subplots(2, 1, sharex=True)
ax1.plot(res[:,0], res[:,1])
ax2.plot(res[:,0], res[:,2], color='orange')
ax1.set_ylabel('x')
ax2.set_ylabel('p')
ax2.set_xlabel('t')
```

²⁹https://en.wikipedia.org/wiki/Transmission_Control_Protocol

```
plt.show()
```



This simulation is simple and naive.

- From a mathematical point of view it's naive, because there are much more sophisticated numerical methods for solving differential equations³⁰.
- From a computer science point of view this isn't ideal, because the code tangles together the iteration logic with the logging logic. It should really be rewritten with lazy lists.
- But from a scientific computing point of view, simulations like this are so easy to put together and learn from, that they are invaluable.

lazy lists, page 8

What we have coded is called a *discrete-time simulation*, because time advances in fixed increments. In IA Algorithms you will study the 'heap' data structure, which is good simulation in which time is pegged to changes in state, called *event-driven simulation*.

³⁰<https://docs.scipy.org/doc/scipy-0.13.0/reference/generated/scipy.integrate.ode.html>

3. Working with data

3.1. Preamble

In this section we'll use Pandas³¹, a library which is ubiquitous for all Python data science. We'll look at the two standard ways of arranging data, *data frames* and *indexed arrays*. We'll also see some more advanced plotting with matplotlib³².

```
import numpy as np
import pandas
import matplotlib.pyplot as plt
```

The running example for this section is a dataset of stop-and-search records, made available by the UK home office³³. As it's a moderate-sized file (172MB) I like to download it to disk, so it's fast to reread it each time I restart the notebook. Here's how we can fetch a file from a url, using the Unix command-line tool `wget`. (The exclamation mark is called a *Jupyter magic*³⁴, and it means "Treat this line as though it were executed at the command prompt". In IB Unix Tools you'll learn more about the Unix command line.)

```
# Execute a unix command to download a file (if it's not already
# downloaded), and show download progress
import os.path
if os.path.exists('stop-and-search.csv'):
    print("file already downloaded")
else:
    !wget "https://teachingfiles.blob.core.windows.net/datasets/stop-and-search.csv"
```

When using Pandas and Matplotlib you will often hit your head against the wall and exclaim "that's crazy! who would design a library like this?" A comment from a recent discussion on Hacker News³⁵ explains it well: Pandas "is designed for scientists who know nothing about how a library should be designed or how a program should be structured. There's a lot of dynamic stuff in Pandas that while making things easier for scientists make things a lot more difficult for CS people." In my opinion, Pandas does have some ugly library design; but it also serves a need which computer scientists don't really get until they've tried working with data; and furthermore Pandas has some rather innovative thinking around indexing, which might better have been left to an experimental package rather than the mainstream.

3.2. What data looks like

We almost always work with data in the form of a spreadsheet-like table, referred to as a *dataframe*. A dataframe is a collection of named columns. Each column has the same length, and all entries in a column have the same type, though different columns may have different types. Pandas uses numpy to store columns, so it's reasonably fast.

Here's how to load a dataframe from a file using `pandas.read_csv`, and how to inspect it. (This dataframe will be used as a running example in the rest of section 3.) The Pandas library is full of handy utilities like `read_csv`.

```
# Import a dataframe using the pandas library
stopsearch = pandas.read_csv('stop-and-search.csv')

# How many rows are there?
print(f"This dataset has {len(stopsearch)} rows")

# What are the columns?
stopsearch.columns
```

³¹<http://pandas.pydata.org/>

³²<https://matplotlib.org/>

³³<https://data.police.uk/data/>

³⁴<http://ipython.readthedocs.io/en/stable/interactive/magics.html>

³⁵<https://news.ycombinator.com/item?id=21550516>

This dataset has 1013915 rows

```
Index(['force', 'month', 'age_range', 'datetime', 'gender', 'involved_person',
      'legislation', 'location', 'location_latitude', 'location_longitude',
      'location_street_id', 'location_street_name', 'object_of_search',
      'officer_defined_ethnicity', 'operation', 'operation_name', 'outcome',
      'outcome_linked_to_object_of_search', 'outcome_object_id',
      'outcome_object_name', 'removal_of_more_than_outer_clothing',
      'self_defined_ethnicity', 'type'],
      dtype='object')
```

```
# Display the first 5 rows. iloc[:5] means "select the first five rows"
# (not all columns fit on this page)
stopsearch.iloc[:5]
```

	force	month	age_range	datetime	gender	involved_person	legislation
0	bedfordshire	2019-08	18-24	2019-08-01T00:30:00+00:00	Male	True	Misuse of Drugs Act 1971 (secti
1	bedfordshire	2019-08	18-24	2019-08-03T16:28:00+00:00	Male	True	Misuse of Drugs Act 1971 (secti
2	bedfordshire	2019-08	NaN	2019-08-08T16:36:00+00:00	Male	True	Misuse of Drugs Act 1971 (secti
3	bedfordshire	2019-08	10-17	2019-08-08T18:20:00+00:00	Male	True	Misuse of Drugs Act 1971 (secti
4	bedfordshire	2019-08	18-24	2019-08-08T20:50:26+00:00	Male	True	Misuse of Drugs Act 1971 (secti

3.2.1. MISSING VALUES

Missing values (as in the third entry in the `age_range` column above) are a fact of life in data science. They should really be supported by Python itself, but they aren't, so Pandas adopts its own conventions: it uses either `np.nan` (the IEEE floating point for 'Not A Number'), or `None` (the built-in Python value commonly used to denote 'no return value'). It's best to use `np.nan` when the underlying column is a numpy vector of floating point values, but for other column types it doesn't matter which is used. To determine whether values are missing, use `pandas.isna`.

```
pandas.isna(stopsearch.age_range[:5]) # returns [False,False,True,False,False]
sum(pandas.isna(stopsearch.age_range)) # count number of missing values
```

3.3. Importing, exporting, and creating dataframes

It's very easy to import data from a simple comma-separated value (CSV) file. A CSV file looks like this:

```
"Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width", "Species"
5.1, 3.5, 1.4, 0.2, "setosa"
4.9, 3, 1.4, 0.2, "setosa"
4.7, 3.2, 1.3, 0.2, "setosa"
4.6, 3.1, 1.5, 0.2, "setosa"
5, 3.6, 1.4, 0.2, "setosa"
```

i.e. a header line, then one line per row of the data frame, with values separated by commas. We've already seen how to import a CSV, using `pandas.read_csv`³⁶. If your file is nearly a CSV but has some quirks such as comments or a missing header row, experiment with that function's 55 options. We can use the same function to read CSV files from remote urls³⁷

```
url = 'https://teachingfiles.blob.core.windows.net/datasets/iris.csv'
iris = pandas.read_csv(url)
```

In my experience, around 70% of the time you spend working with data will be fighting to import it and clean it up. See the online notebooks for a collection of recipes for web scraping, reading from a database, and parsing log files.

To write a CSV file³⁸,

```
iris.to_csv('iris.csv', index=False)
```

To create a dataframe from scratch, pass in a dictionary of columns. Python dictionaries are unordered, so you can optionally specify the column order you want with the `columns` argument.

```
iris = pandas.DataFrame({
    'species': ['setosa', 'virginica', 'virginica', 'setosa', 'versicolor'],
    'Petal.length': [1.0, 5.0, 5.8, 1.7, 4.2],
    'Petal.width': [0.2, 1.9, 1.6, 0.5, 1.2]
},
    columns = ['species', 'Petal.length', 'Petal.width'])
```

Or you can create a dataframe from a list of tuples. Now the `columns` argument is needed to say what the column names are.

```
iris = pandas.DataFrame([
    ('setosa', 1.0, 0.2), ('virginica', 5.0, 1.9), ('virginica', 5.8, 1.6),
    ('setosa', 1.7, 0.5), ('versicolor', 4.2, 1.2)
],
    columns = ['species', 'Petal.length', 'Petal.width'])
```

³⁶https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

³⁷though if you're using Azure Notebooks, be aware that Azure only permits you to connect to Azure web servers.

³⁸If you're running this notebook with Azure Notebooks, you would then use the Data | Download menu to download the file from Azure Notebooks to your local machine.

3.4. Selecting and modifying data

Data frames have a triple identity:

- they're like a dictionary, where the keys are column names and the values are numpy vectors, and we can add, modify, or remove entire columns
- they're like a database table, and we can select subtables by row and/or column
- they're like an array, and we can select parts of the dataframe based on row indexes.

Like a dictionary. We can access entire columns as though the dataframe is a dictionary.

```
stopsearch.columns          # get a list of column names
stopsearch.keys()          # ... and another way to do the same

x = stopsearch['outcome']  # get a single column
x = stopsearch.outcome     # ... and another way to do the same

del stopsearch['location'] # delete a column

# add or modify a column
stopsearch['outcome_N'] = np.where(stopsearch.outcome == 'False', 0, 1)
```

Like a database table. We can obtain a new dataframe by selecting a subset of rows and/or columns, using `.loc`.

```
stopsearch.loc[:, ['force', 'datetime', 'outcome']] # all rows, some cols
stopsearch[['force', 'datetime', 'outcome']]        # ... the same thing
stopsearch.loc[stopsearch.force=='cambridgeshire'] # some rows, all cols
stopsearch.loc[stopsearch.force=='cambridgeshire', # some rows, some cols
               ['force', 'datetime', 'outcome']]
```

If we want to select rows by row number, rather than by a boolean condition as above, we need `.iloc`.

```
stopsearch.iloc[:3]        # the first 3 rows
stopsearch[:3]             # ... and another way to do the same
stopsearch.iloc[[0,3,5]]  # select several rows
stopsearch.iloc[[5]]      # returns a one-row dataframe
stopsearch.sample(4)      # select 4 rows at random
```

Row and column selectors can be combined.

```
wantcols = ['force', 'datetime', 'outcome']
stopsearch.loc[stopsearch.force=='cambridgeshire', wantcols]
stopsearch[wantcols].iloc[:3]
stopsearch.loc[stopsearch.force=='cambridgeshire', wantcols].iloc[:3]
```

To pull out a single row as a tuple, or to pull out a single value as a scalar, there is different syntax.

```
stopsearch['force'].iat[5] # a scalar for the specified column and row
stopsearch.iloc[5]        # a tuple containing the values for row 5
```

We can use these indexing operations to update a specific element in the dataframe—but (depending how exactly we do it) Pandas will tell us off, warning us that the operation may be inefficient. I think it's cleaner to modify data using dictionary indexing, replacing an entire column, rather than hacking at individual elements.

```
stopsearch['outcome_N'][0] = 2
```

SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation:

http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-copy

To suppress this warning, take a copy of the dataframe with `df=stopsearch.copy()`, and then modify the copy.

Like an array. There is a third way to select rows from a dataframe, which in my experience is the source of endless confusion: selecting by row index. When you see a Pandas dataframe printed out, there is a column at the left without a column name. These aren't row numbers, they are *row indexes*, which behave like the keys in a dictionary. In all the examples we've seen so far the indexes happen to be numbers, but they could be any other Python object. Row indexes are there for the same reason database tables have indexes: they're vital for efficient lookup. But we won't be using them in this course.

Pandas remembers row indexes, even when you pull out a single column, and it always tries to match indexes. This is usually not what we want. I recommend that you **generally use `.values` when you are working with subsets of rows**. This gives you the actual numpy vector behind the column, not the confusing Pandas vector-plus-index object. We won't be taking advantage of row indexes in this course, but it's worth knowing they exist so you can understand the cryptic errors and error messages you will undoubtedly come across. 

```
df = pandas.DataFrame({'x': [3,3,4,8,2]}, index=['a', 'b', 'c', 'd', 'e'])
```

	x
a	3
b	3
c	4
d	8
e	2

```
# This looks like it should add [3,3,4] and [4,8,2] ... but it doesn't!  
df['x'][:3] + df['x'][2:]
```

	x
a	NaN
b	NaN
c	8.0
d	NaN
e	NaN

```
# To get the answer we were probably expecting,  
df['x'][:3].values + df['x'][-3:].values
```

```
array([ 7, 11,  6])
```

3.5. Tabulations and indexed arrays

The pattern behind much data processing is split-apply-combine-join: split your data into pieces, apply a transformation to each piece, combine the pieces, and join results from different datasets together. We could code this explicitly with a ‘for’ loop, but it would involve lots of boilerplate code — and I hope you have been persuaded by section 2.2.3 that ‘for’ loops are considered harmful. Instead, let’s see how to do it with Pandas.

3.5.1. DATAFRAME → INDEXED ARRAY

The following line of code performs a cross-tabulation: it splits the data into a separate dataframe for each combination of officer-defined ethnicity and gender, applies the len function to each sub-dataframe to get the number of rows it contains, and combines the results into a single indexed object.

```
# Select cambridgeshire records, then tabulate by ethnicity and gender
df = stopsearch.loc[stopsearch.force=='cambridgeshire'].copy()
x = df.groupby(['officer_defined_ethnicity', 'gender']).apply(len)
```

```
officer_defined_ethnicity  gender
Asian                    Female      7
                        Male       179
                        Other        1
Black                   Female     10
                        Male       257
Other                   Female      6
                        Male        28
White                   Female    253
                        Male     1465
                        Other         5

dtype: int64
```

We can also apply more elaborate functions. Here are two equivalent ways to apply np.mean to a year column in each sub-dataframe.

```
# First define the year column
df['year'] = [int(yyyymm[:4]) for yyyymm in df['month']]

groupcols = ['officer_defined_ethnicity', 'gender']
df.groupby(groupcols).apply(lambda sub_df: np.mean(sub_df['year']))
df.groupby(groupcols)['year'].apply(np.mean)
```

```
officer_defined_ethnicity  gender
Asian                    Female    2017.714286
                        Male     2017.324022
                        Other     2017.000000
Black                   Female    2017.300000
                        Male     2017.408560
Other                   Female    2017.833333
                        Male     2017.285714
White                   Female    2017.454545
                        Male     2017.326962
                        Other     2017.200000

Name: year, dtype: float64
```

For this course, we will only apply functions that return simple Python values. It’s possible but more complicated³⁹ to apply functions that return dataframes or Pandas columns or indexed arrays.

The groupby/apply commands have produced an *indexed array*. An indexed array is a cross between a normal numpy array and a dataframe. We access elements and sub-arrays by dimension, like a numpy array — but the indexes aren’t integer positions, they’re values from the underlying column. Also, the array might be ‘incomplete’, as in the example above which has no entry for [‘Black’, ‘Other’].

```
x.loc['Asian']           # select the sub-array of ethnicity Asian
x.loc[:, 'Other']       # select the sub-array of gender Other
x.loc[['Black', 'White']] # select two ethnicities, all genders
```

³⁹<http://pandas.pydata.org/pandas-docs/stable/groupby.html>

The index labels can be accessed with `x.index.levels[0].values` and `x.index.levels[1].values`.

To pretty-print an indexed array, use `unstack`⁴⁰. It will by default fill in any missing values with NaN (not a number), and you can override this with `fill_value`.

```
x.unstack(fill_value=0)
```

gender	Female	Male	Other
officer_defined_ethnicity			
Asian	7	179	1
Black	10	257	0
Other	6	28	0
White	253	1465	5

3.5.2. INDEXED ARRAY → DATAFRAME

There are two ways to convert an indexed array to a dataframe, depending on the shape of the dataframe you want to end up with.

```
# Convert an indexed array into a long-form dataframe
x[['Black', 'White']].reset_index(name='count')
```

	officer_defined_ethnicity	gender	count
0	Black	Female	10
1	Black	Male	257
2	White	Female	253
3	White	Male	1465
4	White	Other	5

```
# Convert an indexed array into a wide-form dataframe.
x[['Black', 'White']].unstack(fill_value=0).reset_index() \
    .rename_axis(None, axis=1)
```

	officer_defined_ethnicity	Female	Male	Other
0	Black	10	257	0
1	White	253	1465	5

When you first start working with data, I recommend you do all your calculations on dataframes rather than indexed arrays. If you want to do calculations on an indexed array, first turn it into a dataframe. As you get deeper into working with data, you'll discover that the skill in working with data is knowing which representation works best for your task, dataframe or indexed array. Also,

- Pandas blurs the boundary between dataframes and indexed arrays
- Both rows and columns can have hierarchical indexes, called multi-indexes⁴¹
- For clever tricks with higher-dimensional indexed arrays, read the documentation for `unstack`⁴² and `rename_axis`⁴³.
- When you read the documentation or look for help, please note that what I'm calling an indexed array, Pandas calls a `Series`⁴⁴.

⁴⁰<https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.unstack.html>

⁴¹<https://pandas.pydata.org/pandas-docs/stable/advanced.html>

⁴²<https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.unstack.html>

⁴³https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.rename_axis.html

⁴⁴<https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html>

3.6. Database-style joins

When processing data we often want to combine data at different levels of aggregation. For example, we might like to compare the frequency of false stops (i.e. where the police stopped someone and found nothing suspicious) across different ethnicities. Here's how we prepare the first three columns ...but what about the n_{tot} column and n/n_{tot} ?

```
df = stopsearch.loc[stopsearch.force=='cambridgeshire'].copy()
df['outcome'] = np.where(df.outcome == 'False', 'nothing', 'find')
x = df.groupby(['officer_defined_ethnicity', 'outcome']).apply(len)
x.reset_index(name='n')
```

	officer_defined_ethnicity	outcome	n	n_{tot}	n / n_{tot}
0	Asian	find	116	192	60%
1	Asian	nothing	76		40%
2	Black	find	170	270	63%
3	Black	nothing	100		37%
4	Other	find	28	37	76%
5	Other	nothing	9		24%
6	White	find	1060	1740	61%
7	White	nothing	680		39%

The database answer is to create a smaller table with two columns, `officer_defined_ethnicity` and n_{tot} , and then to join this to `x` using the key `officer_defined_ethnicity`.

```
y = x.groupby('officer_defined_ethnicity')['n'].apply(sum).reset_index(name='ntot')
```

	officer_defined_ethnicity	ntot
0	Asian	192
1	Black	270
2	Other	37
3	White	1740

```
z = x.merge(y, on='officer_defined_ethnicity')
p = z.n / z.ntot
z['percent_find'] = np.round(p * 100, 1)
# Also compute a margin for error; see IB Data Science for the theory
z['err'] = np.round(1.96 * np.sqrt(p*(1-p)/z.ntot) * 100, 1)
```

	officer_defined_ethnicity	outcome	n	ntot	percent_find	err
0	Asian	find	116	192	60.4	6.9
1	Asian	nothing	76	192	39.6	6.9
2	Black	find	170	270	63.0	5.8
3	Black	nothing	100	270	37.0	5.8
4	Other	find	28	37	75.7	13.8
5	Other	nothing	9	37	24.3	13.8
6	White	find	1060	1740	60.9	2.3
7	White	nothing	680	1740	39.1	2.3

```
# Show only the interesting bit of the summary table
z.loc[z.outcome=='nothing', ['officer_defined_ethnicity', 'percent_find', 'err']]
```

	officer_defined_ethnicity	percent_find	err
1	Asian	39.6	6.9
3	Black	37.0	5.8
5	Other	24.3	13.8
7	White	39.1	2.3

Pandas also lets us join indexed arrays on their common indices, and that would be a more natural way to write this calculation; but that counts as more advanced Pandas usage than we will cover here.

3.7. Plotting

Matplotlib is a huge and not very coherent⁴⁵ plotting library for Python, inspired by decades-old plotting from MATLAB. There are some alternatives, but they are mostly too restrictive, or just thin wrappers over Matplotlib, or immature, so Matplotlib is the one to learn at least for the time being.

3.7.1. BASIC PRINCIPLES

Here is the general structure of plot code. I find it helpful to build up my plot step by step, adding pieces in the order listed here, and checking at each step what the plot looks like. If you add everything all in one go, chances are it won't work and you won't know which bit went wrong.

```
# First, prepare the data and put it into a dataframe

# Set figure size and other style parameters
with plt.rc_context({'figure.figsize': (4,3)}):
    # Get the overall Figure object (used for some overall customization)
    # and Axes objects, one for each subplot (used for the actual plotting)
    fig,ax = plt.subplots(...)

# 1. Draw data points / bars / curves etc. onto ax
# 2. Configure limits and colour scales
# 3. Add annotations, text, arrows, etc.
# 4. Configure the format of the ticks
# 5. Legend, axis labels, titles

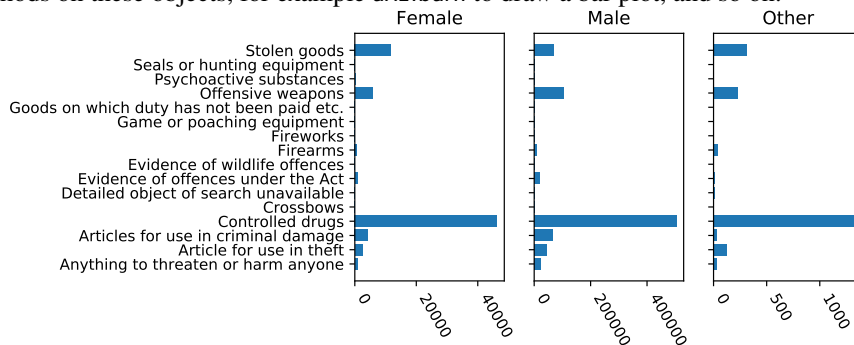
# Save as pdf or svg or png, depending on the destination
# (On Azure, use the Data | Download menu to download the saved plot.)
plt.savefig('myplot.pdf', transparent=True, bbox_inches='tight', pad_inches=0)

plt.show()
```

A plot consists of one or more subplots, as in the example below. To create this plot, we start by specifying the subplot layout we want,

```
fig,(ax1,ax2,ax3) = plt.subplots(nrows=1,ncols=3, sharey=True)
```

This gives us three objects of class Axes⁴⁶, one for each subplot. The rest of the plot is made by calling various methods on these objects, for example `ax1.barh` to draw a bar plot, and so on.



You'll also see plenty of code samples which use commands like `plt.barh` or `plt.yticks`. That's old-style 'stateful' code, where matplotlib tries to work out which subplot you're currently drawing on—it works fine if you only have one subplot, but it's confusing when you have multiple subplots. Matplotlib documentation advises that for more complex plots you should get the Axes object first and then use `ax.barh` or `ax.set_yticks`. Even if you only have one plot, I advise starting with

```
# get an Axes object for a plot with a single subplot, default is nrows=ncols=1
fig,ax = plt.subplots()
```

⁴⁵Another Hacker News comment: "Matplotlib belongs to the worst category of software: very powerful and very awful. Nothing makes any sense and it's so profoundly unintuitive it almost feels like I'm being pranked. But, of course, use it I must. Pandas also comes off as an unintuitive joke, but my displeasure with it has mostly worn off. Matplotlib however makes me feel angry pretty much everyday." <https://news.ycombinator.com/item?id=21550516>

⁴⁶https://matplotlib.org/api/axes_api.html#the-axes-class

and then using the Axes interface. You need to know how to use it anyway, and there's no point learning two interfaces.

Most of the customization methods are duplicated between the two styles, but with niggling differences: for example `plt.yticks` is equivalent to `ax.set_yticks` combined with `ax.set_yticklabels`. All of the `plt` commands have documentation that explains what the Axes equivalent is, so if you find a code sample online that uses `plt` then it's easy enough to translate it to Axes.

3.7.2. PLOT GALLERY

The best way to learn Matplotlib is to browse through galleries until you find something you like, and copy it, remembering the basic principles above. To customize your plots you'll need to make frequent use of Google, Stack Overflow⁴⁷, the matplotlib gallery⁴⁸, and maybe if things get desperate look at the documentation for `plt.*`⁴⁹ commands, and for the Axes⁵⁰ class. Here's a starting gallery. Just look at the pictures, and use this section for reference if you find yourself wondering how to produce a similar plot.

Basic plot skeleton. Here's the code behind our first plot, shown above. Note the line

```
fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, sharey=True)
```

which asks for three subplots in a row, and says that their *y* scales are to be shared. Matplotlib picks scales automatically to fit the objects drawn onto a subplot, and `sharey=True` means that all three subplots get their scales adjusted. It also means that the tick marks are only shown on one of the three subplots.

```
# Prepare the data, and put it into a dataframe
x = stopsearch.groupby(['object_of_search', 'gender']).apply(len)
df = x.unstack(fill_value=0).reset_index().rename_axis(None, axis=1)

# Set figure size, no other style parameters
with plt.rc_context({'figure.figsize':(6,3)}):
    # we want three subplots in a row,
    # so request three Axes objects on which to draw
    fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, sharey=True)

# 1. draw a horizontal barplot, one for each subplot / ethnicity
for (ax, eth) in zip([ax1, ax2, ax3], ['Female', 'Male', 'Other']):
    ax.barh(np.arange(len(df)), df[eth])

# 2. we've already specified, via sharey=True, that the three plots
# share a y-axis. No other axis limits to set.

# 4. configure the ticks
ax1.set_yticks(np.arange(len(df)))
ax1.set_yticklabels(df.object_of_search)

# 5. final tweaks
for ax, eth in zip([ax1, ax2, ax3], ['Female', 'Male', 'Other']):
    ax.set_title(eth)

plt.show()
```

Simple bar chart. Here is an elementary bar chart.

```
# Prepare the data, and put the rows in the order we want for plotting
x = stopsearch.groupby('age_range').apply(len)
x = x[['under 10', '10-17', '18-24', '25-34', 'over 34']]
```

⁴⁷<https://stackoverflow.com/questions/tagged/matplotlib>

⁴⁸<https://matplotlib.org/2.1.0/gallery/index.html>

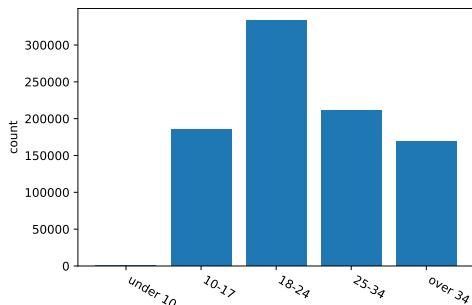
⁴⁹https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html#module-matplotlib.pyplot

⁵⁰https://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes

```
df = x.reset_index(name='n')

fig,ax = plt.subplots()
ax.bar(np.arange(len(df)), df.n)
ax.set_xticks(np.arange(len(df)))
ax.set_xticklabels(df.age_range, rotation=-30, ha='left')
ax.set_ylabel('count')

plt.show()
```



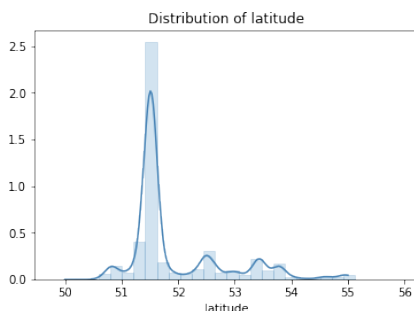
Histogram and density plot. This plot shows two graphics superimposed, a histogram (i.e. a bar chart based on binned counts), and a smooth curve for the density. To produce the smooth curve we can use a generic smoother such as `scipy.stats.gaussian_kde`, which takes the underlying data and returns a function, and then apply this function to evenly-spaced values along the x -axis to generate the points to be plotted.

```
x = stopsearch.location_latitude
x = x[~pandas.isna(x)] # remove missing values
import scipy.stats
# Smoothing is slow, and it produces just as good results on a subset
density = scipy.stats.gaussian_kde(np.random.choice(x,50000))

fig,ax = plt.subplots()
ax.hist(x, bins=30, density=True, alpha=0.2, edgecolor='steelblue')
xsample = np.linspace(50,55,200)
ax.plot(xsample, density(xsample), color='steelblue')

ax.set_xlabel('latitude')
ax.set_title('Distribution of latitude')

plt.show()
```



Scatter plot. A scatter plot, with explicit control of the colour scale and the coordinate scales.

```
# There's no point plotting more data than there are pixels on the output
df = stopsearch.iloc[np.random.choice(len(stopsearch), size=100000)]

fig,ax = plt.subplots()

cols = plt.get_cmap('Set2', len(np.unique(df.force)))

for i,police_force in enumerate(np.unique(df.force)):
```

```

want_rows = (df.force == police_force)
x,y = df.location_longitude[want_rows], df.location_latitude[want_rows]
# Set the size, alpha, and colour of the points
ax.scatter(x, y, s=1, alpha=.1, color=cols(i))

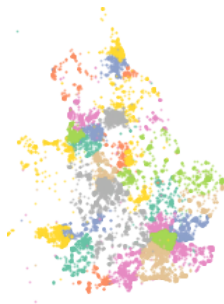
# Set the aspect ratio, based on the UK's average latitude
ax.set_aspect(1/np.cos(54/360*2*np.pi))

# Pick coordinates to show (based on viewing the plot)
ax.set_xlim([-5,2])
ax.set_ylim([50.2, 55.8])

# Get rid of the tick marks and the outer frame
ax.set_xticks([])
ax.set_yticks([])
ax.axis('off')

plt.show()

```



Time series. There are several techniques being used in this example.

- The dataset as loaded stores the datetime as a string, which isn't very useful. Here I convert it to Python datetime⁵¹, and Matplotlib knows how to display it sensibly.
- The two plot commands both have a label. Matplotlib remembers the styling that was applied for each label, and can generate an appropriate legend.
- I set figure.figsize to be (5, 1.5). Technically the units are in inches, but the output gets stretched anyway when it's included in these printed notes—so why does it help? Matplotlib measures text size in inches too, so when we tell it to generate a small plot, the text will be larger with respect to the plot size. That's why this plot has legible text, compared to the plots on earlier pages where the text was tiny.

```

df = stopsearch.loc[stopsearch.force=='cambridgeshire', ['datetime', 'outcome']].copy()
df['outcome'] = np.where(df.outcome=='False', 'nothing', 'find')

import datetime, pytz
def as_datetime(s):
    return datetime.datetime.strptime(s[:10], '%Y-%m-%d').replace(tzinfo=pytz.UTC)
df['t'] = np.vectorize(as_datetime)(df.datetime)

df = df.groupby(['t', 'outcome']).apply(len).unstack(fill_value=0).reset_index()
df = df.iloc[np.argsort(df.t)]

with plt.rc_context({'figure.figsize':(5,1.5)}):
    fig,ax = plt.subplots()

ax.plot(df.t, df.find + df.nothing, label='stops', linewidth=3)
ax.plot(df.t, df.find, label='find', linewidth=1)

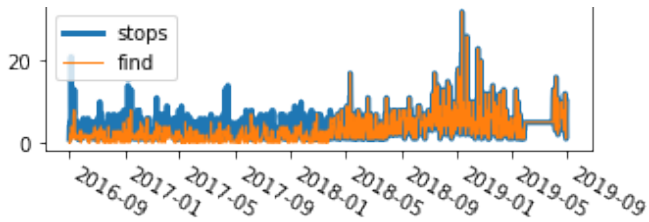
ax.legend()

```

⁵¹<https://docs.python.org/3/library/datetime.html#datetime.datetime>


```
# Some magic to improve tick labels for an entire figure
fig.autofmt_xdate(bottom=0.2, rotation=-30, ha='left')

plt.show()
```



Panel plot. Our final plot is called a *facet plot* or a *small multiples* plot. According to the plotting guru Edward Tufte⁵²,

At the heart of quantitative reasoning is a single question: Compared to what? Small multiple designs, multivariate and data bountiful, answer directly by visually enforcing comparisons of changes, of the differences among objects, of the scope of alternatives. For a wide range of problems in data presentation, small multiples are the best design solution.

Our very first plot on page 27 was also a facet plot. There are actually two types of facet plot:

- We might want a grid of plots, either 1d or 2d. To get a grid of Axes objects, use `fig, axes = plt.subplots(nrows, ncols)`.

This will return either a vector of axes or an array of axes, according to `nrows` and `ncols`.

- We might want a sequence of plots which is allowed to wrap over several lines. For this, decide how many rows and columns we'll want in total, then call `add_subplot` to add each facet one by one.

```
fig = plt.figure()
ax = fig.add_subplot(nrows, ncols, i) # i starts at 1
```

One other thing worth mentioning in this code: I first convert all the datetimes into Unix timestamps (integers, counting the number of seconds since Thursday 1970-01-01 00:00:00), and then I do simple integer arithmetic to get dates and weekdays. I find this easier than wading through library documentation about datetime utility functions, and it's also much faster because it's simple vectorized numpy expressions.

```
import datetime, pytz
def as_timestamp(s):
    t = datetime.datetime.strptime(s[:10], '%Y-%m-%d').replace(tzinfo=pytz.UTC)
    return int(t.timestamp())

df = stopsearch.loc[stopsearch.force=='cambridgeshire'].copy()
df['t'] = np.vectorize(as_timestamp)(df.datetime)
df['date'] = df.t // (24*3600)
df['weekday'] = (df.t // (24*3600) - 4) % 7
df2 = df.groupby(['date', 'weekday']).apply(len).reset_index(name='n')

with plt.rc_context({'figure.figsize': (8,5), 'figure.subplot.hspace': 0.35}):
    fig = plt.figure()

    for i, weekday in enumerate(range(7)):
        ax = fig.add_subplot(3, 3, i+1)

        # 1. Draw the data
        ax.hist(df2.loc[df2.weekday==weekday, 'n'].values, bins=range(15), alpha=.3)
        # 2. Configure limits
        ax.set_ylim([0,30])
```

⁵²https://en.wikipedia.org/wiki/Small_multiple

```
# 3. Add annotations
ax.axvline(x=np.median(df2.n), linestyle='dotted', color='black')
# 4. Configure ticks
if i < 4: ax.set_xticklabels([])
if (i % 3) != 0: ax.set_yticklabels([])
# 5. Legend, axis, titles
weekday_names = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
ax.set_title(weekday_names[weekday])

fig.suptitle('Number of stops')
plt.show()
```

