Programming concepts

Chris Paciorek

2025-10-01

Table of contents

Overview

This unit covers a variety of programming concepts, illustrated in the context of Python and with comments about and connections to other languages. It also serves as a way to teach some advanced features of Python. In general the concepts are relevant in other languages, though other languages may implement things differently. One of my goals for the unit is for us to think about why things are the way they are in Python. I.e., what principles were used in creating the language and what choices were made? While other languages use different principles and made different choices, understanding what one language does in detail will be helpful when you are learning another language or choosing a language for a project.

Quarto configuration details for this document

This document uses the `knitr` engine for rendering to be able to run chunks in multiple languages (Python and bash, as well as a few R chunks. It has the Quarto configuration `ipynb-shell-interactivity: all`, so that output from all Python code in a chunk will print in the rendered document; when done with the `knitr` engine, the output is interspersed with the code in the chunk rather than printed all at the end.

1. Text manipulation, string processing and regular expressions (regex)

Text manipulations in Python have a number of things in common with UNIX, R, and Perl, as many of the ideas/software evolved from UNIX. When I use the term *string* here, I'll be referring to any sequence of characters that may include numbers, white space (including newlines), and special characters, usually stored as an object of the `str` class.

String processing and regular expressions in Python

Here we'll see functionality for working with strings in Python, focusing on regular expressions with the `re` package. This will augment our consideration of regular expressions in the shell, in particular by seeing how we can replace patterns in addition to finding them.

The `re` package provides Perl-style regular expressions, but it doesn't seem to support named character classes such as `[:digit:]`. Instead use classes such as `\d` and `[0-9]`.

Finding patterns

In Python, you can apply a matching function and then query the result to get information about what was matched and where in the string.

```
text = "Here's my number: 919-543-3300."
m = re.search("\\d+", text)
m
```

```
<re.Match object; span=(18, 21), match='919'>
```

```
m.group()
```

```
'919'
```

```
m.start()
```

```
18
```

```
m.end()
```

```
21
```

```
m.span()
```

```
(18, 21)
```

Notice that that showed us only the first match.

The discussion of special characters explains why we need to provide `\\d` rather than `\d`.

We can instead use `findall` to get all the matches.

```
re.findall("\\d+", text)
```

```
['919', '543', '3300']
```

This is equivalent to:

```
pattern = re.compile("\\d+")
re.findall(pattern, text)
```

```
['919', '543', '3300']
```

The compile can be omitted and will be done implicitly, but is a good idea to do explicitly if you have a complex regex pattern that you will use repeatedly (e.g., on every line in a file). It is also a reminder that regular expressions is a separate language, which can be compiled into a program. The compilation results in an object that relies on finite state machines to match the pattern.

To ignore case, do the following:

```
text = "That cat in the Hat"
re.findall("hat", text, re.IGNORECASE)
```

```
['hat', 'Hat']
```

There are several other regex flags (also called compilation flags) that can control the behavior of the matching engine in interesting ways (check out `re.VERBOSE` and `re.MULTILINE` for instance).

We can of course use list comprehension to work with multiple strings. But we need to be careful to check whether a match was found.

```
def return_group(pattern, txt):
    m = re.search(pattern, txt)
    if m:
       return m.group()
    else:
       return None

text = ["Here's my number: 919-543-3300.", "hi John, good to meet you",
        "They bought 731 bananas", "Please call 1.919.554.3800"]
[return_group("\\d+", str) for str in text]
```

```
['919', None, '731', '1']
```

Recall that we can search for location-specific matches in relation to the start and end of a string.

```
text = "hats are all that are important to a hatter."
re.findall("^hat\\w+", text)
```

```
['hats']
```

Recall that we can search based on repetitions (as already demonstrated with the `\w+` just above).

```
text = "Here's my number: 919-543-3300. They bought 731 hats. Please call 1.919.554.3800."
re.findall("\\d{3}[-.]\\d{3}[-.]\\d{4}", text)
```

```
['919-543-3300', '919.554.3800']
```

As another example, the phone number detection problem could have been done a bit more compactly (as well as more generally to allow for an initial "1-" or "1.") as:

```
text = "Here's my number: 919-543-3300. They bought 731 bananas. Please call 1.919.554.3800."
re.findall("((1[-.])?(\\d{3}[-.]){1,2}\\d{4})", text)
```

```
[('919-543-3300', '', '543-'), ('1.919.554.3800', '1.', '554.')]
```

Question: the above regex would actually match something that is not a valid phone number. What can go wrong?

When you are searching for all occurrences of a pattern in a large text object, it may be beneficial to use `finditer`:

```
it = re.finditer("(http|ftp):\\/\\/", text)  # http or ftp followed by ://

for match in it:
    match.span()
```

This method behaves lazily and returns an iterator that gives us one match at a time, and only scans for the next match when we ask for it. This is similar to the behavior we saw with `pandas.read_csv(chunksize = n)`

Manipulating and replacing patterns

We can replace matching substrings with `re.sub`.

```
text = "Here's my number: 919-543-3300."
re.sub("\\d", "Z", text   )
```

```
"Here's my number: ZZZ-ZZZ-ZZZZ."
```

Next let's consider grouping using `()`. We'll see that the grouping operator also controls what is returned as the matched patterns.

Here's a basic example of using grouping via parentheses with the OR operator.

```
text = "At the site http://www.ibm.com. Some other text. ftp://ibm.com"
re.search("(http|ftp):\\/\\/", text).group()
```

```
'http://'
```

However, if we want to find all the matches and try to use `findall`, we see that, when grouping operators are present, it returns only the "captured" groups, as discussed a bit in `help(re.findall)`, so we'd need to add an additional grouping operator to capture the full pattern when using `findall`:

```
re.findall("(http|ftp):\\/\\/", text)
```

```
['http', 'ftp']
```

```
re.findall("((http|ftp):\\/\\/)", text)
```

```
[('http://', 'http'), ('ftp://', 'ftp')]
```

If we only wanted to full pattern without capturing the inner group, we can use some additional syntax, `?:`, for "non-capturing" groups:

```
re.findall("((?:http|ftp):\\/\\/)", text)
```

```
['http://', 'ftp://']
```

Groups are also used when we need to reference back to a detected pattern when doing a replacement. This is why they are sometimes referred to as "capturing groups". For example, here we'll find any numbers and add underscores before and after them:

```
text = "Here's my number: 919-543-3300. They bought 731 bananas. Please call 919.554.3800."
re.sub("([0-9]+)", "_\\1_", text)
```

```
"Here's my number: _919_-_543_-_3300_. They bought _731_ bananas. Please call _919_._554_._3800_."
```

Here we'll remove commas not used as field separators.

```
text = '"H4NY07011","ACKERMAN, GARY L.","H","$13,242",,,'
re.sub("([^\",]),", "\\1", text)
```

```
'"H4NY07011","ACKERMAN GARY L.","H","$13242",,,'
```

How does that work? Consider that `[^\",]` matches a character that is not a quote and not a comma. The regex is such a character followed by a comma, with the matched character saved in `\\1` because of the grouping operator.

> **Challenge**
>
> Instead of removing the commas to remove the ambiguity, how would you convert the comma delimiters to pipe (`|`) delimiters (since the pipe is rarely used in text)?

Extending the use of `\\1`, we can refer to multiple captured groups:

```
text = "Here's my number: 919-543-3300. They bought 731 bananas. Please call 919.554.3800."
re.sub("([0-9]{3})[-\.]([0-9]{3})[-\.]([0-9]{4})", "area code \\1, number \\2-\\3", text)
```

```
"Here's my number: area code 919, number 543-3300. They bought 731 bananas. Please call area code 919
```

Additional regex functionality

Regex extensions that we won't discuss further here include:

- Groups can also be given names, instead of having to refer to them by their numbers.
- We can have `sub` call a "callback" function to do the replacement. The function will be invoked on each match with the argument being equivalent to the result of `re.search`.
- We can reference previously captured groups within a pattern using the same `\\1`-style syntax (as opposed to when doing replacement as seen above).

Challenge

Suppose a text string has dates in the form "Aug-3", "May-9", etc. and I want them in the form "3 Aug", "9 May", etc. How would I do this regex?

Greedy matching

Finally, let's consider where a match ends when there is ambiguity.

As a simple example consider that if we try this search, we match as many digits as possible, rather than returning the first "9" as satisfying the request for "one or more" digits.

```
text = "See the 998 balloons."
re.findall("\\d+", text)
```

```
['998']
```

That behavior is called *greedy* matching, and it's the default. That example also shows why it is the default. What would happen if it were not the default?

However, sometimes greedy matching doesn't get us what we want.

Consider this attempt to remove multiple html tags from a string.

```
text = "Do an internship <b> in place </b> of <b> one </b> course."
re.sub("<.*>", "", text)
```

```
'Do an internship  course.'
```

Notice what happens because of greedy matching.

One way to avoid greedy matching is to use a `?` after the repetition specifier.

```
re.sub("<.*?>", "", text)
```

```
'Do an internship  in place  of  one  course.'
```

However, that syntax is a bit frustrating because `?` is also used to indicate 0 or 1 repetitions, making the regex a bit hard to read/understand.

Challenge

Suppose I want to strip out HTML tags but without using the `?` to avoid greedy matching. How can I be more careful in constructing my regex?

Special characters in Python

Recall that when characters are used for special purposes, we need to 'escape' them if we want them interpreted as the actual (*literal*) character. In what follows, I show this in Python, but similar manipulations are sometimes needed in the shell and in R.

This can get particularly confusing in Python as the backslash is also used to input special characters such as newline (`\n`) or tab (`\t`). Apparently there was some change in handling *escape sequences* as of Python 3.12. We now need to do this for the regex `\d`:

```
re.search("\\d+", "a93b")
```

```
<re.Match object; span=(1, 3), match='93'>
```

In Python 3.11, it was fine to use `\d`, but now we need `\\d`, because Python now tries to interpret `\d` as a special character (like `\n`, but `\d` doesn't exist) and doesn't pass it directly along as regex syntax.

Here are some examples of using special characters.

```
tmp = "Harry said, \"Hi\""
print(tmp)   # This prints out without a newline -- this is hard to show in rendered doc.
```

```
Harry said, "Hi"
```

```
tmp = "Harry said, \"Hi\".\n"
print(tmp)   # This prints out with the newline -- hard to show in rendered doc.
```

```
Harry said, "Hi".
```

```
tmp = ["azar", "foo", "hello\tthere\n"]
print(tmp[2])
```

```
hello   there
```

```
re.search("[\tZ]", tmp[2])   # Search for a tab or a 'Z'.
```

```
<re.Match object; span=(5, 6), match='\t'>
```

Here are some examples of using various special characters in regex syntax. To use a special character as a regular character, we need to escape it (which in Python 3.12 involves two backslashes, as discussed above):

```
## Search for characters that are not 'z'
## (using ^ as regular expression syntax)
re.search("[^z]", "zero")
```

```
<re.Match object; span=(1, 2), match='e'>
```

```
## Show results for various input strings:
for st in ["a^2", "93", "zzz", "zit", "azar"]:
    print(st + ":\t", re.search("[^z]", st))
```

```
a^2:     <re.Match object; span=(0, 1), match='a'>
93:  <re.Match object; span=(0, 1), match='9'>
zzz:     None
zit:     <re.Match object; span=(1, 2), match='i'>
azar:    <re.Match object; span=(0, 1), match='a'>
```

```
## Search for either a '^' (as a regular character) or a 'z':
for st in ["a^2", "93", "zzz", "zit", "azar"]:
    print(st + ":\t", re.search("[\\^z]", st))
```

```
a^2:     <re.Match object; span=(1, 2), match='^'>
93:  None
zzz:     <re.Match object; span=(0, 1), match='z'>
zit:     <re.Match object; span=(0, 1), match='z'>
azar:    <re.Match object; span=(1, 2), match='z'>
```

```
## Search for exactly three characters
## (using . as regular expression syntax)
for st in ["abc", "1234", "def"]:
    print(st + ":\t", re.search("^.{3}$", st))
```

```
abc:     <re.Match object; span=(0, 3), match='abc'>
1234:    None
def:     <re.Match object; span=(0, 3), match='def'>
```

```
## Search for a period (as a regular character)
for st in ["3.9", "27", "4.2"]:
    print(st + ":\t", re.search("\\.", st))
```

```
3.9:     <re.Match object; span=(1, 2), match='.'>
27:  None
4.2:     <re.Match object; span=(1, 2), match='.'>
```

> **Challenge**
>
> Explain why we use a single backslash to get a newline and double backslash to write out a Windows path in the examples here:

```
## Suppose we want to use a \ in our string:
print("hello\nagain")
```

```
hello
again
```

```
print("hello\\nagain")
```

```
hello\nagain
```

```
print("My Windows path is: C:\\Users\\nadal.")
```

```
My Windows path is: C:\Users\nadal.
```

Another way to achieve this effect if your string does not contain any special characters is to prefix your string literal with an r for "raw":

```
print(r"My Windows path is: C:\Users\nadal.")
```

```
My Windows path is: C:\Users\nadal.
```

On a more involved note, searching for an actual backslash gets even more complicated (you can search online for "backslash plague" or "backslash hell"), because we need to pass two backslashes as the regular expression, so that a literal backslash is searched for. However, to pass two backslashes, we need to escape each of them with a backslash so Python doesn't treat each backslash as part of a special character. So that's four backslashes to search for a single backslash! Yikes. One rule of thumb is just to keep entering backslashes until things work!

```
## Use and search for an actual backslash
tmp = "something \\ other\n"
print(tmp)
```

```
something \ other
```

```
re.search("\\\\", tmp)
```

```
<re.Match object; span=(10, 11), match='\\'>
```

```
try:
    re.search("\\", tmp)
except Exception as error:
    print(error)
```

```
bad escape (end of pattern) at position 0
```

Again here you can use "raw" strings, at the price of losing the ability to use any special characters:

```
## Search for an actual backslash
re.search(r"\\", tmp)
```

```
<re.Match object; span=(10, 11), match='\\'>
```

The use of the raw string `r"\\"` tells Python to treat this string literal without any escaping, but that does not apply to the regex engine (or else we would have used a single backslash), so we do need the second backslash. So yes. This can be quite confusing.

> ⚠ Pasting quotation marks
>
> Be careful when cutting and pasting from documents that are not text files as you may paste in something that looks like a single or double quote, but which Python cannot interpret as a quote because it's some other ASCII (or Unicode) quote character. If you paste in a ” from PDF, it will not be interpreted as a standard Python double quote mark.

Similar things come up in the shell and in R, but in the shell you often don't need as many backslashes. E.g. you could do this to look for a literal backslash character.

```
echo "hello" > file.txt
echo "with a \ there" >> file.txt
grep '\\' file.txt
```

```
with a \ there
```

2. Interacting with the operating system and external code and configuring Python

Interacting with the operating system

Scripting languages allow one to interact with the operating system in various ways. Most allow you to call out to the shell to run arbitrary shell code and save results within your session.

I'll assume everyone knows about the following functions/functionality for interacting with the filesystem and file in Python: `os.getcwd`, `os.chdir`, `import`, `pickle.dump`, `pickle.load`

Also in IPython there is additional functionality/syntax.

Here are a variety of tools for interacting with the operating system:

- To run UNIX commands from within Python, use `subprocess.run()`, as follows, noting that we can save the result of a system call to an Python object:

```
import subprocess, io
subprocess.run(["ls", "-al"])   ## results apparently not shown when compiled...
```

```
CompletedProcess(args=['ls', '-al'], returncode=0)
```

```
files = subprocess.run(["ls", "-al"], capture_output = True)
files.stdout
```

```
b'total 4081\ndrwxr-sr-x 14 paciorek scfstaff      82 Oct  8 08:18 .\ndrwxr-sr-x 16 paciorek scf
```

```
with io.BytesIO(files.stdout) as stream:  # create a file-like object
    content = stream.readlines()
content[2:4]
```

```
[b'drwxr-sr-x 16 paciorek scfstaff      52 Oct  8 08:16 ..\n', b'-rw-r--r--  1 paciorek scfstaff
```

- There are also a bunch of functions that will do specific queries of the filesystem, including

```
os.path.exists("unit2-dataTech.qmd")
```

```
True
```

```
os.listdir("../data")
```

```
['hivSequ.csv', 'cpds.csv', 'precipData.txt', 'stackoverflow-2021.db', 'coop.txt.gz', 'airline.c
```

- There are some tools for dealing with differences between operating systems. `os.path.join` is a nice example:

```
os.listdir(os.path.join("..", "data"))
```

```
['hivSequ.csv', 'cpds.csv', 'precipData.txt', 'stackoverflow-2021.db', 'coop.txt.gz', 'airline.c
```

  It's best if you can to write your code, as shown here with `os.path.join`, in a way that is *agnostic* to the underlying operating system (i.e., that works regardless of the operating system).

- To get some info on the system you're running on:

```
import platform
platform.system()
```

```
'Linux'
```

```
os.uname()
```

```
posix.uname_result(sysname='Linux', nodename='smeagol', release='6.8.0-85-generic', version='#85
```

```
platform.python_version()
```

```
'3.13.2'
```

```
sys.version
```

```
'3.13.2 | packaged by conda-forge | (main, Feb 17 2025, 14:40:20) [GCC 13.3.0]'
```

- To retrieve environment variables:

```
os.environ['PATH']
```

```
'/system/linux/miniforge-3.13/bin:/system/linux/miniforge-3.13/condabin:/system/linux/miniforge-
```

- You can have an Python script act as a shell script (like running a bash shell script) as follows.

  1. Write your Python code in a text file, say `example.py`
  2. As the first line of the file, include `#!/usr/bin/python` (like `#!/bin/bash` in a bash shell file, as seen in Unit 2) or for more portability across machines, include `#!/usr/bin/env python`.
  3. Make the Python code file executable with `chmod`: `chmod ugo+x example.py`.
  4. Run the script from the command line: `./example.py`

  If you want to pass arguments into your script, you can do so with the `argparse` package.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-y', '--year', default=2002,
                    help='year to download')
parser.add_argument('-m', '--month', default=None,
                    help='month to download')
args = parse.parse_args()
args.year
year = int(args.year)
```

Now we can run it as follows in the shell:

```
./example.py 2004 January
```

- Use `Ctrl-C` to interrupt execution. This will generally back out gracefully, returning you to a state as if the command had not been started. Note that if Python is exceeding the amount of memory available, there can be a long delay. This can be frustrating, particularly since a primary reason you would want to interrupt is when Python runs out of memory.

Interacting with external code

Scripting languages such as R, Python, and Julia allow you to call out to "external code", which often means C or C++ (but also Fortran, Java and other languages).

Calling out to external code is particularly important in languages like R and Python that are often much slower than compiled code and less important in a fast language like Julia (which uses Just-In-Time compilation – more on that later).

In fact, the predecessor language to R, which was called 'S' was developed specifically (at AT&T's Bell Labs in the 1970s and 1980s) as an interactive wrapper around Fortran, the numerical programming language most commonly used at the time (and still widely relied on today in various legacy codes).

In Python, one can directly call out to C or C++ code or one can use *Cython* to interact with C. With Cython, one can:

- Have Cython automatically translate Python code to C, if you provide type definitions for your variables.
- Define C functions that can be called from your Python code.

In R, one can call directly out to C or C++ code using *.Call* or one can use the Rcpp package. *Rcpp* is specifically designed to be able to write C++ code that feels somewhat like writing R code and where it is very easy to pass data between R and C++.

3. Modules and packages

Scripting languages that become popular generally have an extensive collection of add-on packages available online (the causal relationship of the popularity and the extensive add-on packages goes in both directions).

A big part of Python's popularity is indeed the extensive collection of add-on packages on PyPI (and GitHub and elsewhere) and via `Conda` that provide much of Python's functionality (including core numerical capabilities via `numpy` and `scipy`).

To make use of a package it needs to be installed on your system (using `pip install` or `conda install`) *once* and loaded into Python (using the `import` statement) *every time you start a new session.*

Some modules are *installed* by default with Python (e.g., `os` and `re`), but all need to be loaded by the user in a given Python session.

Modules

A *module* is a collection of related code in a file with the extension `.py`. The code can include functions, classes, and variables, as well as runnable code. To access the objects in the module, you need to import the module.

Here we'll create `mymod.py` from the shell, but of course usually one would create it in an editor.

```
cat << EOF > mymod.py
x = 7
range = 3
def myfun(x):
    print("The arg is: ", str(x), ".", sep = '')
EOF
```

```
import mymod
print(mymod.x)
```

```
7
```

```
mymod.myfun(99)
```

```
The arg is: 99.
```

The import statement

The import statement allows one to get access to code in a module. Importantly it associates the names of the objects in the module with a name accessible in the scope in which it was imported (i.e., the current context). The mapping of names (references) to objects is called a *namespace.* We discuss scopes and namespaces in more detail later.

```
del mymod
try:           # Check if `mymod` is in scope.
    mymod.x
except Exception as error:
    print(error)
```

```
name 'mymod' is not defined
```

```
y = 3

import mymod
mymod         # This is essentially a dictionary in the current (global) scope.
```

```
<module 'mymod' from '/accounts/vis/paciorek/teaching/243fall25/fall-2025/units/mymod.py'>
```

```
x             # This is not a name in the current (global) scope.
```

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

```
range         # This is a builtin, not from the module.
```

```
<class 'range'>
```

```
mymod.x
```

```
7
```

```
dir(mymod)
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spe
```

```
mymod.x
```

```
7
```

```
mymod.range
```

```
3
```

So `y` and `mymod` are in the global namespace and `range` and `x` are in the module namespace of `mymod`. You can access the built-in `range` function from the global namespace but it turns out it's actually in the built-ins scope (more later).

Note the usefulness of distinguishing the objects in a module from those in the global namespace. We'll discuss this more in a bit.

That said, we can make an object defined in a module directly accessible in the current scope (adding it to the global namespace in this example) at which point it is distinct from the object in the module:

```
from mymod import x
x   # now part of global namespace
```

```
7
```

```
dir()
```

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', '
```

```
mymod.x = 5
x = 3
mymod.x
```

```
5
```

```
x
```

```
3
```

But in general we wouldn't want to use `from` to import objects in that fashion because we could introduce name *conflicts* and we reduce modularity.

That said, it can be tedious to always have to type the module name (and in many cases there are multiple submodule names you'd also need to type).

```
import mymod as m
m.x
```

```
5
```

Packages

A package is a directory containing one or more modules and with a file named `__init__.py` that is called when a package is imported and serves to initialize the package.

Let's create a basic package.

```
mkdir mypkg

cat << EOF > mypkg/__init__.py
## Make objects from mymod.py available as mypkg.foo rather than mypkg.mymod.foo.
## The "." is a "relative" import that means find "mymod" here in this directory.
from .mymod import *

print("Welcome to my package.")
EOF

cat << EOF > mypkg/mymod.py
x = 7

def myfun(val):
    print(f"Converting {val} to integer: {int(val)}.")
EOF
```

Note that if there were other modules, we could have imported from those as well.

Now we can use the objects from the module without having to know that it was in a particular module (because of how `__init__.py` was set up).

```
import mypkg
```

```
Welcome to my package.
```

```
mypkg.x
```

```
7
```

```
mypkg.myfun(7.3)
```

```
Converting 7.3 to integer: 7.
```

Note, one can set `__all__` in an `__init__.py` to define what is imported, which makes clear what is publicly available and hides what is considered internal.

Internal/private objects

We could add another module that the main module uses but that is not intended for direct use by the user. Here we name the function to start with `_` following the convention that this indicates a private/internal function.

```
cat << EOF > mypkg/auxil.py

def _helper(val):
    return val + 10
EOF


cat << EOF >> mypkg/mymod.py

from .auxil import _helper

def myfun10(val):
    print(f"Converting {val} to integer plus 10: {int(_helper(val))}.")
EOF
```

```
del mypkg
import mypkg
mypkg.myfun10(7.3)
```

```
Converting 7.3 to integer plus 10: 17.
```

```
del mypkg
```

Subpackages

Packages can also have modules in nested directories, achieving additional modularity via *subpackages*. A package can automatically import the subpackages via the main `__init__.py` or require the user to import them manually, e.g., `import mypkg.mysubpkg`.

```
mkdir mypkg/mysubpkg

cat << EOF > mypkg/mysubpkg/__init__.py
from .values import *
print("Welcome to my package's subpackage.")
EOF
```

```
cat << EOF > mypkg/mysubpkg/values.py
x = 999
b = 7
_my_internal_var = 9
EOF
```

```
import mypkg.mysubpkg     ## Note that __init__.py is invoked
```

```
Welcome to my package's subpackage.
```

```
mypkg.mysubpkg.b
```

```
7
```

```
mypkg.x
```

```
7
```

Note that a given `__init__.py` is invoked when importing anything nested within the directory containing the `__init__.py`; in the case above the `__init__.py` from `mypkg` is invoked, though for some reason the "Welcome to my package." output is not showing up in this rendered document.

If we wanted to automatically import the subpackage we would add `from . import mysubpkg` to `mypkg/__init__.py`, which uses relative imports. The alternative "absolute" import would be `import mypkg.mysubpkg`, which finds `mypkg` using `sys.path` (which specifies a set of paths of where to look).

One would generally not import the items from `mysubpkg` directly into the `mypkg` namespace but there may be cases one would do something like this. For example `numpy.linspace` is actually found in `numpy/core/function_base.py`, but we don't need to refer to `numpy.core.linspace` because of how `numpy` structures the `import` statements in `__init__.py`. In contrast, the linear algebra functions are available via the subpackage namespace as `numpy.linalg.<function_name>`.

Take a look at `dir(numpy)`, `dir(numpy.linalg)`, and `dir(numpy.core)` to get a better sense for this in a real package.

Installing packages

If a package is on PyPI or available through Conda but not on your system, you can install it easily (usually). You don't need root permission on a machine to install a package, though you may need to use `pip install --user` or set up a new Conda environment.

Packages often depend on other packages. In general, if one package depends on another, pip or conda will generally install the dependency automatically.

One advantage of Conda is that it can also install non-Python packages on which a Python package depends, whereas with pip you sometimes need to install a system package to satisfy a dependency.

> **Using Mamba/libmamba and the Conda-forge channel**
>
> It's not uncommon to run into a case where conda has trouble installing a package because of version inconsistencies amongst the dependencies. `mamba` is a drop-in replacement for `conda` and often does a better job of this "dependency resolution". We use `mamba` by default on the SCF. In recent versions of Conda, you can also use the Mamba's dependency resolver when running `conda` commands by running `conda config --set solver libmamba`, which puts `solver: libmamba` in your `.condarc` file. It's also generally recommended to use the `conda-forge` *channel* (i.e., location) when installing packages with Conda (this is done automatically when using `mamba`). `conda-forge` provides a wide variety of up-to-date packages, maintained by the community.

Making your package installable (optional)

It's pretty easy to configure your package so that it can be built and installed via `pip`. See the structure of this example repository. In fact, one can install the package with only either `setup.py` or `pyproj.toml`, but the other files listed here are recommended:

- `pyproj.toml` (or `pyproject.toml`): this is a configuration file used by packaging tools. In the `mytoy` example it specifies to use `setuptools` to build and install the package.
- `setup.py`: this is run when the package is built and installed when using `setuptools`. In the example, it simply runs `setuptools.setup()`. With recent versions of `setuptools`, you don't actually need this so long as you have the `pyproj.toml` file.
- `setup.cfg`: provides metadata about the package when using `setuptools`.
- `environment.yml`: provides information about the full environment in which your package should be used (including examples, documentation, etc.). For projects using `setuptools`, a minimal list of dependencies needed for installation and use of the package can instead be included in the `install_requires` option of `setup.cfg`.
- `LICENSE`: specifies the license for your package giving the terms under which others can use it.

The `postBuild` file is a completely optional file only needed if you want to use the package with a MyBinder environment.

At the numpy GitHub repository, by looking in `pyproject.toml`, you can see that `numpy` is build and installed using a system called *Meson*, while at the Jupyter GitHub repository you can see that the `jupyter` package is built and installed using `setuptools`.

Building a package usually refers to compiling source code but for a Python package that just has Python code, nothing needs to be compiled. *Installing* a package means putting the built package into a location on your computer where packages are installed.

You can also make your package public on PyPI or through Conda, but that is not something we'll cover here.

Reproducibility and package management

For reproducibility, it's important to know the versions of the packages you use (and the version of Python). `pip` and `conda` make it easy to do this. You can create a *requirements* file that captures the

packages you are currently using (and, critically, their versions) and then install exactly that set of packages (and versions) based on that requirements file.

```
pip freeze > requirements.txt
pip install -r requirements.txt

conda env export --no-builds > environment.yml
conda env create -f environment.yml
```

The `--no-builds` flag make sure not to include information about the dependencies that are system-specific (e.g., that would only work on MacOS and not on Windows).

Conda is a general package manager. You can use it to manage Python packages but lots of other software as well, including R and Julia.

Conda environments provide an additional layer of modularity/reproducibility, allowing you to set up a fully reproducible environment for your computation. Here (by explicitly giving `python=3.13`) the Python 3.13 executable and all packages you install in the environment are fully independent of whatever Python executables are installed on the system.

```
conda create -n myenv python=3.13
source activate myenv
conda install numpy
```

> **Warning**
>
> If you use `conda activate` rather than `source activate`, Conda will prompt you to run `conda init`, which will make changes to your `~/.bashrc` that, for one, activate the Conda base environment automatically when a shell is started. This may be fine, but it's helpful to be aware.

Package locations

Packages in Python (and in R, Julia, etc.) may be installed in various places on the filesystem, and it sometimes it is helpful (e.g., if you end up with multiple versions of a package installed on your system) to be able to figure out where on the filesystem the package is being loaded from.

We can use the `__file__` and `__version__` objects in a package to see where on the filesystem a package is installed and what version it is:

```
import numpy as np
np.__file__
```

```
'/system/linux/miniforge-3.13/lib/python3.13/site-packages/numpy/__init__.py'
```

```
np.__version__
```

```
'2.1.3'
```

(`pip list` or `conda list` will also show version numbers, for all packages.)

`sys.path` shows where Python looks for packages on your system.

Source vs. binary packages

The difference between a *source* package and a *binary* package is that the source package has the raw Python (and C/C++ and Fortran, in some cases) code as text files, while the binary package has all the non-Python code in a binary/non-text format, with the C/C++ and Fortran code already having been compiled.

If you install a package from source, C/C++/Fortran code will be compiled on your system (if the package has such code). That should mean the compiled code will work on your system, but requires you to have a compiler available and things properly configured. A binary package doesn't need to be compiled on your system, but in some cases the code may not run on your system because it was compiled in such a way that is not compatible with your system.

Python *wheels* are a binary package format for Python packages. Wheels for some packages will vary by platform (i.e., operating system) so that the package will install correctly on the system where it is being installed.

4. Types and data structures

Data structures

Please see the data structures section of Unit 2 for some general discussion of data structures.

We'll also see more complicated data structures when we consider objects in the section on object-oriented programming.

Types and classes

Overview and static vs. dynamic typing

The term 'type' refers to how a given piece of information is stored and what operations can be done with the information.

'Primitive' types are the most basic types that often relate directly to how data are stored in memory or on disk (e.g., boolean, integer, numeric (real-valued, aka *double* or *floating point*), character, pointer (aka *address*, *reference*).

In compiled languages like C and C++, one has to define the type of each variable. Such languages are *statically* typed. Interpreted (or scripting) languages such as Python and R have *dynamic* types. One can associate different types of information with a given variable name at different times and without declaring the type of the variable:

```
x = 'hello'
print(x)
```

```
hello
```

```
x*3
```

```
'hellohellohello'
```

```
x = 7
x*3
```

```
21
```

```
x = {'mykey': 7}
x*3
```

```
TypeError: unsupported operand type(s) for *: 'dict' and 'int'
```

In contrast in a language like C, one has to declare a variable based on its type before using it:

```
double y;
double x = 3.1;
y = x * 7.1;
```

Dynamic typing can be quite helpful from the perspective of quick implementation and avoiding tedious type definitions and problems from minor inconsistencies between types (e.g., multiplying an integer by a real-valued number). But static typing has some critical advantages from the perspective of software development, including:

- protecting against errors from mismatched values and unexpected user inputs, and
- generally much faster execution because the type of a variable does not need to be checked (and the location of the value found) when the code is run.

More complex types in Python (and in R) often use references (*pointers*, aka *addresses*) to the actual locations of the data. We'll see this in detail when we discuss Memory.

Types in Python

You should be familiar with the important built-in data types in Python, most importantly lists, tuples, and dictionaries, as well as basic scalar types such as integers, floats, and strings.

Let's look at the type of various built-in data structures in Python and in numpy, which provides important types for numerical computing.

```
x = 3
type(x)
```

```
<class 'int'>
```

```
x = 3.0
type(x)
```

```
<class 'float'>
```

```
x = 'abc'
type(x)
```

```
<class 'str'>
```

```
x = False
type(x)
```

```
<class 'bool'>
```

```
x = [3, 3.0, 'abc']
type(x)
```

```
<class 'list'>
```

```
import numpy as np

x = np.array([3, 5, 7])  ## array of integers
type(x)
```

```
<class 'numpy.ndarray'>
```

```
type(x[0])
```

```
<class 'numpy.int64'>
```

```
x = np.random.normal(size = 3) # array of floats (aka 'doubles')
type(x[0])
```

```
<class 'numpy.float64'>
```

```
x = np.random.normal(size = (3,4)) # multi-dimensional array
type(x)
```

```
<class 'numpy.ndarray'>
```

Sometimes numpy may modify a type to make things easier for you, which often works well, but you may want to control it yourself to be sure:

```
x = np.array([3, 5, 7.3])
x
```

```
array([3. , 5. , 7.3])
```

```
type(x[0])
```

```
<class 'numpy.float64'>
```

```
x = np.array([3.0, 5.0, 7.0]) # Force use of floats (either `3.0` or `3.`).
type(x[0])
```

```
<class 'numpy.float64'>
```

```
x = np.array([3, 5, 7], dtype = 'float64')
type(x[0])
```

```
<class 'numpy.float64'>
```

This can come up when working on a GPU, where the default is usually 32-bit (4-byte) numbers instead of 64-bit (8-byte) numbers.

Composite objects

Many objects can be *composite* (e.g., a list of dictionaries or a dictionary of lists, tuples, and strings).

```
mydict = {'a': 3, 'b': 7}
mylist = [3, 5, 7]

mylist[1] = mydict
mylist
```

```
[3, {'a': 3, 'b': 7}, 7]
```

```
mydict['a'] = mylist
```

Mutable objects

Most objects in Python can be modified *in place* (i.e., modifying only some of the object), but tuples, strings, and sets are *immutable*:

```
x = (3,5,7)
try:
    x[1] = 4
except Exception as error:
    print(error)
```

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

```
s = 'abc'
s[1]
```

```
'b'
```

```
try:
    s[1] = 'y'
except Exception as error:
    print(error)
```

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

Converting between types

This also goes by the term *coercion* and *casting*. Casting often needs to be done explicitly in compiled languages and somewhat less so in interpreted languages like Python.

We can *cast* (coerce) between different basic types:

```
y = str(x[0])
y
```

```
'3'
```

```
y = int(x[0])
type(y)
```

```
<class 'int'>
```

Some common conversions are converting numbers that are being interpreted as strings into actual numbers and converting between booleans and numeric values.

In some cases Python will automatically do conversions behind the scenes in a smart way (or occasionally not so smart way). Consider these attempts/examples of implicit coercion:

```
x = np.array([False, True, True])
x.sum()         # What do you think is going to happen?
```

```
np.int64(2)
```

```
x = np.random.normal(size = 5)
try:
    x[3] = 'hat'    # What do you think is going to happen?
except Exception as error:
    print(error)
```

```
could not convert string to float: 'hat'
```

```
myList = [1, 3, 5, 9, 4, 7]
# myList[2.0]    # What do you think is going to happen?
# myList[2.73]   # What do you think is going to happen?
```

R is less strict and will do conversions in some cases that Python won't:

```
x <- rnorm(5)
x[2.0]
```

```
[1] 0.5304117
```

```
x[2.73]
```

```
[1] 0.5304117
```

Question: What are the advantages and disadvantages of the different behaviors of Python and R?

Dataframes

Hopefully you're also familiar with the Pandas dataframe type.

Pandas picked up the idea of dataframes from R and functionality is similar in many ways to what you can do with R's `dplyr` package.

`dplyr` and `pandas` provide a lot of functionality for the "split-apply-combine" framework of working with "rectangular" data. Unfortunately, using Pandas can be a bit hard to learn/remember. I suggest looking into learning `polars` as an alternative.

Often analyses are done in a stratified fashion - the same operation or analysis is done on subsets of the data set. The subsets might be different time points, different locations, different hospitals, different people, etc.

The split-apply-combine framework is intended to operate in this kind of context: - first one splits the dataset by one or more variables, - then one does something to each subset, and - then one combines the results.

split-apply-combine is also closely related to the famous Map-Reduce framework underlying big data tools such as Hadoop and Spark.

It's also very similar to standard SQL queries involving filtering, grouping, and aggregation.

Python object protocols

There are a number of broad categories of kinds of objects: `mapping`, `number`, `sequence`, `iterator`. These are called object protocols.

All objects that fall in a given category share key characteristics. For example `sequence` objects have notions of length and accessing (ordered) elements by index, while `iterator` objects have notions of "next" and of "stopping".

Object protocols are implemented based on the Python object model using **dunder** methods. For a class (such as one you write) to implement an object protocol, it must define a particular set of class methods, discussed at the link above.

5. Programming paradigms: object-oriented and functional programming

Object-oriented and functional programming are two important approaches to programming.

Functional programming (FP) focuses on writing functions that take inputs and produce outputs. Ideally those functions don't change the state (i.e., the values) of any variables and can be treated as black boxes. Functions can be treated like other variables, such as passing functions as arguments to another function (as one does with `map` in Python).

Object-oriented programming (OOP) revolves around objects that belong to classes. The class of an object defines the fields (the data objects) holding information and methods that can be applied to those fields. When one calls a method, it may modify the value of the fields. A statistical analogy is that an object of a class is like the realization (the object) of a random variable (the class).

One can think of functional programming as being focused on actions (or *verbs* to make an analogy with human language). One carries out a computation as a sequence of function calls. One can think of OOP as being focused on the objects (or *nouns*). One carries out a computation as a sequence of operations with the objects, using the class methods.

Many languages are multi-paradigm, containing aspects of both approaches and allowing programmers to use either approach. Both R and Python are like this, though one would generally consider R to be more functional and Python to be more object-oriented.

Let's illustrate the ideas with some numpy and list functionality.

```
import numpy as np
x = np.array([1.2, 3.5, 4.2, 9.7])
x.shape     # field (or attribute) of the numpy array class
x.sum()     # method of the class
np.sum(x)   # equivalent numpy function
len(x)      # built-in function

# functional approach: apply functions sequentially
x2 = np.reshape(x, (2,2))
x2t = np.transpose(x2)

# functional, but using class methods
x2 = x.reshape(2,2)
x2t = x2.transpose()

# OOP: modify objects using class methods
y = list([1.2, 3.5, 4.2])
y.append(7.9)  # y modified in place using class method
```

Different people have different preferences, but which is better sometimes depends on what you are trying to do. If your computation is a data analysis pipeline that involves a series of transformations of some data, a functional approach might make more sense, since the focus is on a series of actions rather than the state of objects. If your computation involves various operations on fixed objects whose state needs to change, OOP might make more sense. For example, if you were writing code to keep track of student information, it would probably make sense to have each student as an object of a `Student` class with methods such as `register` and `assign_grade`.

6. Object-oriented programming (OOP)

OOP involves organizing your code around objects that contain information, and methods that operate in specific ways on those objects. Objects belong to classes. A class is made up of fields (the data) that store information and methods (functions) that operate on the fields.

By analogy, OOP focuses on the nouns, with the verbs being part of the nouns, while FP focuses on the verbs (the functions), which operate on the nouns (the arguments).

Most of the things we work with in Python are objects. Functions are also objects, as are classes.

```
type(len)
```

```
<class 'builtin_function_or_method'>
```

```
def foo(x):
    print(x)

type(foo)
```

```
<class 'function'>
```

```
import numpy as np
x = np.array([1.2, 3.4])
type(x)
```

```
<class 'numpy.ndarray'>
```

Principles

Some of the standard concepts in object-oriented programming include *encapsulation*, *inheritance*, *polymorphism*, and *abstraction*.

Encapsulation involves preventing direct access to internal data in an object from outside the object. Instead the class is designed so that access (reading or writing) happens through the interface set up by the programmer (e.g., 'getter' and 'setter' methods). However, Python actually doesn't really enforce the notion of internal or private information.

Inheritance allows one class to be based on another class, adding more specialized features. For example in the `statsmodels` package, the OLS class inherits from the WLS class.

Polymorphism allows for different behavior of an object or function depending on the context. A polymorphic function behaves differently depending on the input types. For example, think of a print function or an addition operator behaving differently depending on the type of the input argument(s). A polymorphic object is one that can belong to different classes (e.g., based on inheritance), and a given method name can be used with any of the classes. An example would be having a base or super class called 'algorithm' and various specific machine learning algorithms inheriting from that class. All of the classes might have a 'predict' method.

Abstraction involves hiding the details of how something is done (e.g., via the method of a class), giving the user an interface to provide inputs and get outputs. By making the actual computation a black box, the programmer can modify the internals without changing how a user uses the system.

Classes generally have *constructors* that initialize objects of the class and *destructors* that remove objects.

Classes in Python

Python provides a pretty standard approach to writing object-oriented code focused on classes.

Our example is to create a class for working with random time series. Each object of the class has specific parameter values that control the stochastic behavior of the time series. With a given object we can simulate one or more time series (realizations).

Here's the initial definition of the class with methods and fields (aka attributes).

```
import numpy as np

class tsSimClass:
    '''
    Class definition for time series simulator
```

```
    '''
    ## dunder methods (more later)
    def __init__(self, times, mean = 0, cor_param = 1, seed = 1):
        ## This is the constructor, called when `tsSimClass(...)` is invoked
        ## to create an instance of the class.

        ## For robustness, need checks that `cor_param` is numeric of length 1 and `times` is np arra
        ## Public attributes
        self.n = len(times)
        self.mean = mean
        self.cor_param = cor_param
        ## Private attributes (encapsulation)
        self._times = times
        self._current_U = False
        ## Some setup steps
        self._calc_mats()
        np.random.seed(seed)
    def __str__(self):    # 'print' method
        return f"An object of class `tsSimClass` with {self.n} time points."
    def __len__(self):
        return self.n

    ## Public methods: getter and setter (encapsulation)
    def set_times(self, new_times):
        self._times = new_times
        self._current_U = False
        self._calc_mats()
    def get_times(self):
        return self._times

    ## Main public method
    def simulate(self):
        if not self._current_U:
            self._calc_mats()
        ## analogous to mu+sigma*z for generating N(mu, sigma^2)
        return self.mean + np.dot(self.U.T, np.random.normal(size = self.n))

    ## Private method.
    def _calc_mats(self):
        ## Calculates correlation matrix and Cholesky factor (caching).
        lag_mat = np.abs(self._times[:, np.newaxis] - self._times)
        cor_mat = np.exp(-lag_mat ** 2 / self.cor_param ** 2)
        self.U = np.linalg.cholesky(cor_mat)
        print("Done updating correlation matrix and Cholesky factor.")
        self._current_U = True
```

Now let's see how we would use the class.

```
myts = tsSimClass(np.arange(1, 101), 2, 1)
```

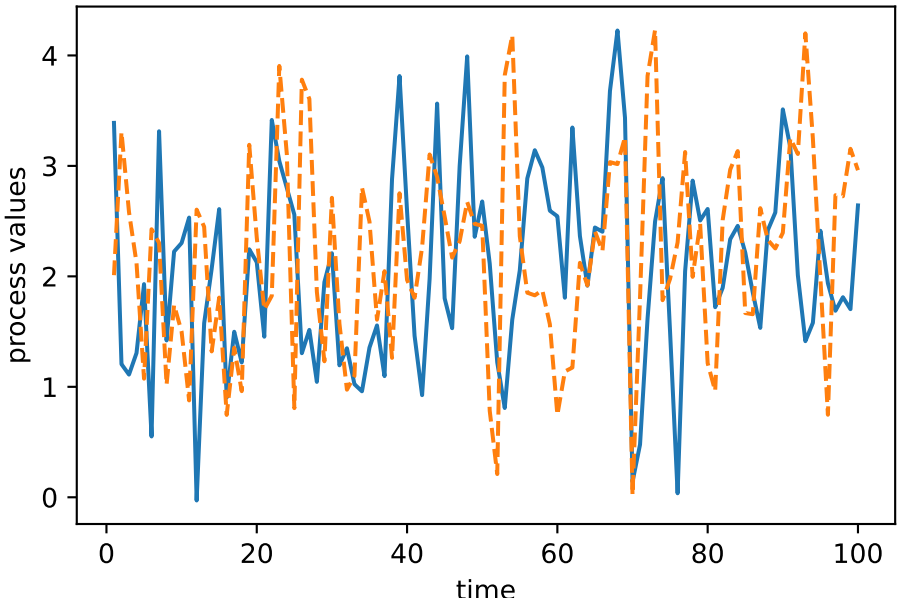
```
Done updating correlation matrix and Cholesky factor.
```

```
print(myts)
```

```
An object of class `tsSimClass` with 100 time points.
```

```
np.random.seed(1)
## Here's a simulated time series.
y1 = myts.simulate()

import matplotlib.pyplot as plt
plt.plot(myts.get_times(), y1, '-')
plt.xlabel('time')
plt.ylabel('process values')
## Simulate a second series.
y2 = myts.simulate()
plt.plot(myts.get_times(), y2, '--')
plt.show()
```



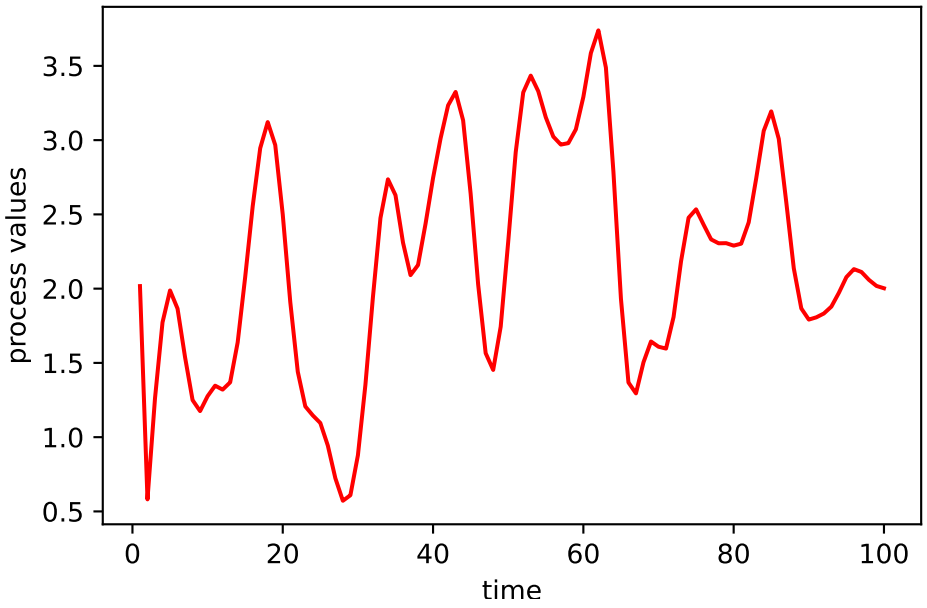
We could set up a different object that has different parameter values. That new simulated time series is less wiggly because the `cor_param` value is larger than before.

```
myts2 = tsSimClass(np.arange(1, 101), 2, 4)
```

```
Done updating correlation matrix and Cholesky factor.
```

```
np.random.seed(1)
## Here's a simulated time series with a different value of
## the correlation parameter (cor_param).
y3 = myts2.simulate()

plt.plot(myts2.get_times(), y3, '-', color = 'red')
plt.xlabel('time')
plt.ylabel('process values')
plt.show()
```



Copies and references

Next let's think about when copies are made. In the next example `myts_ref` is a copy of `myts` in the sense that both names point to the same underlying object. But no data were copied when the assignment to `myts_ref` was done.

```
myts_ref = myts
## 'myts_ref' and 'myts' are names for the same underlying object.
import copy
myts_full_copy = copy.deepcopy(myts)
```

```
## Now let's change the values of a field.
myts.set_times(np.arange(1,1001,10))
```

```
Done updating correlation matrix and Cholesky factor.
```

```
myts.get_times()[0:4]
```

```
array([ 1, 11, 21, 31])
```

```
myts_ref.get_times()[0:4] # the same as `myts`
```

```
array([ 1, 11, 21, 31])
```

```
myts_full_copy.get_times()[0:4] # different from `myts`
```

```
array([1, 2, 3, 4])
```

In contrast `myts_full_copy` is a reference to a different object, and all the data from `myts` had to be copied over to `myts_full_copy`. This takes additional memory (and time), but is also safer, as it avoids the possibility that the user might modify `myts` and not realize that they were also affecting `myts_ref`. We'll discuss this more when we discuss copying in the section on memory use.

Encapsulation

Those of you familiar with OOP will probably be familiar with the idea of public and private fields and methods.

Why have private fields (i.e., *encapsulation*)? The use of private fields shields them from modification by users. Python doesn't really provide this functionality but by convention, attributes whose name starts with `_` are considered private. In this case, we don't want users to modify the `times` field. Why is this important? In this example, the correlation matrix and the Cholesky factor U are both functions of the array of times. So we don't want to allow a user to directly modify `times`. If they did, it would leave the fields of the object in inconsistent states. Instead we want them to use `set_times`, which correctly keeps all the fields in the object internally consistent (by calling `_calc_mats`). It also allows us to improve efficiency by controlling when computationally expensive operations are carried out.

In a module, objects that start with `_` are a weak form of private attributes. Users can access them, but `from foo import *` does not import them.

Inheritance

Inheritance can be a powerful way to reduce code duplication and keep your code organized in a logical (nested) fashion. Special cases can be simple extensions of more general classes. A good example of inheritance in Python and R is how regression models are handled. E.g., in Python's `statsmodels` package, the `OLS` class inherits from the `WLS` class. Or if we think back to the random time series example and generalize it, one might have a `StochasticProcess` class, with a `GaussianProcess` class that inherits from it, and a `ExpGaussianProcess` class (implementing a Gaussian process with an exponential covariance`inheriting from the`GaussianProcess‘ class.

```
class Bear:
      def __init__(self, name, age):
```

```
            self.name = name
            self.age = age
        def __str__(self):
            return f"A bear named '{self.name}' of age {self.age}."
        def color(self):
            return "unknown"

class GrizzlyBear(Bear):
        def __init__(self, name, age, num_people_killed = 0):
            super().__init__(name, age)
            self.num_people_killed = num_people_killed
        def color(self):
            return "brown"

yog = Bear("Yogi the Bear", 23)
print(yog)
```

```
A bear named 'Yogi the Bear' of age 23.
```

```
yog.color()
```

```
'unknown'
```

```
num399 = GrizzlyBear("Jackson Hole Grizzly 399", 35)
print(num399)
```

```
A bear named 'Jackson Hole Grizzly 399' of age 35.
```

```
num399.color()
```

```
'brown'
```

```
num399.num_people_killed
```

```
0
```

Here the `GrizzlyBear` class has additional fields/methods beyond those inherited from the base class (the `Bear` class), i.e., `num_people_killed` (since grizzly bears are much more dangerous than some other kinds of bears), and perhaps additional or modified methods. Python uses the methods specific to the `GrizzlyBear` class if present before falling back to methods of the `Bear` class if not present in the `GrizzlyBear` class.

The above is an example of polymorphism. Instances of the `GrizzlyBear` class are polymorphic because they can have behavior from both the `GrizzlyBear` and `Bear` classes. The `color` method is polymorphic in that it can be used for both classes but is defined to behave differently depending on the class.

Attributes

Both fields and methods are *attributes*.

We saw the notion of attributes when looking at HTML and XML, where the information was stored as key-value pairs that in many cases had additional information in the form of attributes.

Class attributes vs. instance attributes

Here `count` is a class attribute while `name` and `age` are instance attributes.

```
class Bear:
      count = 0
      def __init__(self, name, age):
          self.name = name
          self.age = age
          Bear.count += 1

yog = Bear("Yogi the Bear", 23)
yog.count
```

```
1
```

```
smokey = Bear("Smokey the Bear", 77)
smokey.count
```

```
2
```

The class attribute allows us to manipulate information relating to all instances of the class, as seen here where we keep track of the number of bears that have been created.

Adding attributes

What do you think will happen if we do the following?

```
yog.bizarre = 7
yog.bizarre

def foo(x):
    print(x)

foo.bizarre = 3
foo.bizarre
```

It turns out we can add instance attributes on the fly in some cases, which is a bit disconcerting in some ways.

Generic function OOP

Let's consider the `len` function in Python. It seems to work magically on various kinds of objects.

```
x = [3, 5, 7]
len(x)
```

```
3
```

```
x = np.random.normal(size = 5)
len(x)
```

```
5
```

```
x = {'a': 2, 'b': 3}
len(x)
```

```
2
```

Suppose you were writing the `len` function. What would you have to do to make it work as it did above? What would happen if a user wants to use `len` with a class that they define?

Instead, Python implements the `len` function by calling the `__len__` method of the class that the argument belongs to.

```
x = {'a': 2, 'b': 3}
len(x)
```

```
2
```

```
x.__len__()
```

```
2
```

`__len__` is a *dunder* method (a "Double-UNDERscore" method), which we'll discuss more in a bit.

Something similar occurs with operators:

```
x = 3
x + 5
```

```
8
```

```
x = 'abc'
x + 'xyz'
```

```
'abcxyz'
```

```
x.__add__('xyz')
```

```
'abcxyz'
```

This use of generic functions is convenient in that it allows us to work with a variety of kinds of objects using familiar functions.

The use of such generic functions and operators is similar in spirit to function or method *overloading* in C++ and Java. It is also how the (very) old S3 system in R works. And it's a key part of the (fairly) new Julia language.

Why use generic functions?

The Python developers could have written `len` as a regular function with a bunch of `if` statements so that it can handle different kinds of input objects.

This has some disadvantages:

1. We need to write the code that does the checking.
2. Furthermore, all the code for the different cases all lives inside one potentially very long function, unless we create class-specific helper functions.
3. Most importantly, `len` will only work for existing classes. And users can't easily extend it for new classes that they create because they don't control the `len` (built-in) function. So a user could not add the additional conditions/classes in a big if-else statement. The generic function approach makes the system *extensible* – we can build our own new functionality on top of what is already in Python.

Multiple dispatch OOP

The dispatch system involved in `len` and `+` involves only the first argument to the function (or operator). In contrast, Julia emphasizes the importance of multiple dispatch as particularly important for mathematical computation. With multiple dispatch, the specific method can be chosen based on more than one argument.

In R, the old (but still used in some contexts) S4 system in R and the new R7 system both provide for multiple dispatch.

As a very simple example unrelated to any specific language, multiple dispatch would allow one to do the following with the addition operator:

```
3 + 7    # 10
3 + 'a'  # '3a'
'hi' +  ' there'  # 'hi there'
```

The idea of having the behavior of an operator or function adapt to the type of the input(s) is one aspect of *polymorphism*.

The Python object model and *dunder* methods

Now that we've seen the basics of classes, as well as generic function OOP, we're in a good position to understand the Python object model.

Objects are dictionaries that provide a mapping from attribute names to their values, either fields or methods.

dunder methods are special methods that Python will invoke when various functions are called on instances of the class or other standard operations are invoked. They allow classes to interact with Python's built-ins.

Here are some important dunder methods:

- `__init__` is the constructor (initialization) function that is called when the class name is invoked (e.g., `Bear(...)`)
- `__len__` is called by `len()`
- `__str__` is called by `print()`
- `__repr__` is called when an object's name is invoked
- `__call__` is called if the instance is invoked as a function call (e.g., `yog()` in the `Bear` case)

- `__add__` is called by the `+` operator.
- `__getitem__` is called by the `[` slicing operator.

The print function

Like `len`, `print` is a generic function, with various class-specific methods.

We can write a print method for our own class by defining the `__str__` method. One generally also defines a `__repr__` method giving what to display when the name of an object is typed.

Dunder methods: example

```
## Basic class definition
class Bear:
      def __init__(self, name, age):
          self.name = name
          self.age = age

yog = Bear("Yogi the Bear", 23)
print(yog)
```

```
<__main__.Bear object at 0x792f4b4d2cf0>
```

```
## Class definition with various dunder methods
class Bear:
      def __init__(self, name, age):
          self.name = name
          self.age = age
      def __str__(self):
          return f"A bear named {self.name} of age {self.age}."
      def __repr__(self):
          return f"Bear(name={self.name}, age={self.age})"
      def __add__(self, value):
          self.age += value
          return None

yogi = Bear("Yogi the Bear", 23)
print(yogi)   # Invokes __str__
```

```
A bear named Yogi the Bear of age 23.
```

```
yogi          # Invokes __repr__
```

```
Bear(name=Yogi the Bear, age=23)
```

```
yogi + 12
print(yogi)
```

```
A bear named Yogi the Bear of age 35.
```

Challenge

Let's check our understanding of the object model.
How would you get Python to quit immediately, without asking for any more information, when you simply type `q` (no parentheses!) instead of `quit()`? (Hint: you can do this by understanding what happens when you type `q` and how to exploit the characteristics of Python classes.)

Python object protocols: Example 1 – iterators

A container class that supports iteration should provide the `__iter__` and `__next__` methods to implement the iterator protocol.

Here we see that `tuple`s are iterable containers (although they are not iterators themselves):

```
mytuple = ("apple", "banana", "cherry")

for item in mytuple:
    print(item)
```

```
apple
banana
cherry
```

```
## We can manually create the iterator and iterate through it.
myit = iter(mytuple)
## myit = mytuple.__iter__()  ## This is equivalent to using `iter(mytuple)`.

type(myit)
```

```
<class 'tuple_iterator'>
```

```
print(next(myit))
```

```
apple
```

```
print(next(myit))
```

```
banana
```

```
myit.__next__()   ## This is equivalent to using `next(myit)`.
```

```
'cherry'
```

I think it makes sense that tuples are iterable containers but they are not iterators themselves, because we wouldn't want to "consume" our tuple (leaving it empty) by iterating over it.

Here's another iterator example. `zip` objects are iterators themselves (and we can see them being consumed).

```
x = zip(['clinton', 'bush', 'obama', 'trump'], ['Dem', 'Rep', 'Dem', 'Rep'])
next(x)
```

```
('clinton', 'Dem')
```

```
next(x)
```

```
('bush', 'Rep')
```

```
next(x)
```

```
('obama', 'Dem')
```

```
next(x)
```

```
('trump', 'Rep')
```

```
next(x)
```

```
StopIteration
```

We can also go from an iterable object to a standard list:

```
x = zip(['clinton', 'bush', 'obama', 'trump'], ['Dem', 'Rep', 'Dem', 'Rep'])
list(x)
```

```
[('clinton', 'Dem'), ('bush', 'Rep'), ('obama', 'Dem'), ('trump', 'Rep')]
```

Python object protocols: Example 2 – context managers using `with`

We've seen that the standard way to read/write to a file in Python uses `with`, like this:

```
with open('myfile.txt', 'r') as file:
     lines = file.readlines()
```

This creates a "context manager" that is equivalent to:

```
file = open('myfile.txt', 'r')
try:
    lines = file.readlines()
finally:
    file.close()
```

With either approach, we get (1) exception handling in case something goes wrong with the read/write (in this case `readlines`) and (2) automatic closing of the file, regardless of what happens in the execution of the code, based on the `finally` block that does any needed cleanup.

What does this have to do with *dunder* methods and object protocols?

Well, one can use `with` with any class for which one wants to implements the context manager protocol by providing `__enter__` and `__exit__` methods. These are invoked before and after the code in the scope of the `with` block is run.

Here's a basic example:

```
import time

class MyTimer(object):
```

```
    def __enter__(self):
        print(f"Starting at {time.ctime()}.")

    def __exit__(self, exception_type, exception_value, traceback):
        print(f"Ending at {time.ctime()}.")

with MyTimer():
    x = np.random.normal(size=50000000)
    del x
```

```
Starting at Wed Oct  8 08:18:31 2025.
Ending at Wed Oct  8 08:18:32 2025.
```

A more useful example would be setting up a context manager to handle database queries by connecting to the database via `__enter__` and disconnecting via `__exit__`.

7. Functional programming

This section covers an approach to programming called *functional programming* as well as various concepts related to writing and using functions.

Functional programming (in Python)

Overview of functional programming

Functional programming is an approach to programming that emphasizes the use of modular, self-contained functions. Such functions should operate only on arguments provided to them (avoiding global variables), and **produce no side effects**, although in some cases there are good reasons for making an exception. Another aspect of functional programming is that functions are considered 'first-class' citizens in that they can be passed as arguments to another function, returned as the result of a function, and assigned to variables. In other words, a function can be treated as any other variable.

In many cases (including Python and R), anonymous functions (also called 'lambda functions') can be created on-the-fly for use in various circumstances.

One can do functional programming in Python by focusing on writing modular, self-contained functions rather than classes. And functions are first-class citizens. However, there are aspects of Python that do not align with the principles mentioned above.

- Python's pass-by-reference behavior causes functions to potentially have the important side effects of modifying arguments that are mutable (e.g., lists and numpy arrays but not tuples) if the programmer is not careful about not modifying arguments within functions.
- Some operations are carried out by statements (e.g., `import`, `def`) rather than functions.

In contrast, R functions have pass-by-value behavior, which is more consistent with a pure functional programming approach.

The principle of no side effects

Before we discuss Python further, let's consider how R behaves in more detail as R conforms more strictly to a functional programming perspective.

Most functions available in R (and ideally functions that you write as well) operate by taking in arguments and producing output that is then (presumably) used subsequently. The functions generally don't have any effect on the state of your R environment/session other than the output they produce.

An important reason for this (plus for not using global variables) is that it means that it is easy for people using the language to understand what code does. Every function can be treated a black box – you don't need to understand what happens in the function or worry that the function might do something unexpected (such as changing the value of one of your variables). The result of running code is simply the result of a composition of functions, as in mathematical function composition.

One aspect of this is that R uses a *pass-by-value* approach to function arguments. In R (but not Python), when you pass an object in as an argument and then modify it in the function, you are modifying a local copy of the variable that exists in the context (the *frame*) of the function and is deleted when the function call finishes:

```
x <- 1:3
myfun <- function(x) {
      x[2] <- 7
      print(x)
      return(x)
}

new_x <- myfun(x)
```

```
[1] 1 7 3
```

```
x   # unmodified
```

```
[1] 1 2 3
```

In contrast, Python uses a *pass-by-reference* approach, seen here:

```
x = np.array([1,2,3])
def myfun(x):
  x[1] = 7
  return x

new_x = myfun(x)
x   # modified!
```

```
array([1, 7, 3])
```

And actually, given the pass-by-reference behavior, we would probably use a version of `myfun` that looks like this:

```
x = np.array([1,2,3])
def myfun(x):
```

```
  x[1] = 7
  return None

myfun(x)
x   # modified!
```

```
array([1, 7, 3])
```

Note how easy it would be for a Python programmer to violate the 'no side effects' principle. In fact to avoid it, we need to do some additional work in terms of making a copy of `x` to a new location in memory before modifying it in the function.

```
x = np.array([1,2,3])
def myfun(x):
  y = x.copy()
  y[1] = 7
  return y

new_x = myfun(x)
x   # no side effects!
```

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

More on pass-by-value vs. pass-by-reference later.

Even in R, there are some (necessary) exceptions to the idea of no side effects, such as `par()`, `library()`, and `plot()`.

Functions are first-class objects

Everything in Python is an object, including functions and classes. We can assign functions to variables in the same way we assign numeric and other values.

When we make an assignment we associate a name (a 'reference') with an object in memory. Python can find the object by using the name to look up the object in the namespace.

```
x = 3
type(x)
```

```
<class 'int'>
```

```
try:
    x([1,3,5])  # x is not a function (yet)
except Exception as error:
    print(error)
```

```
'int' object is not callable
```

```
x = sum

x([1,3,5])
```

```
9
```

```
type(x)
```

```
<class 'builtin_function_or_method'>
```

We can call a function based on the text name of the function.

```
function = getattr(np, "mean")
function(np.array([1,2,3]))
```

```
np.float64(2.0)
```

We can also pass a function into another function as the actual function object. This is an important aspect of functional programming. We can do it with our own function or (as we'll see shortly) with various built-in functions, such as `map`.

```
def apply_fun(fun, a):
    return fun(a)

apply_fun(round, 3.5)
```

```
4
```

A function that takes a function as an argument, returns a function as a result, or both is known as a *higher-order function*.

Which operations are function calls?

Python provides various statements that are not formal function calls but allow one to modify the current Python session:

- `import`: import modules or packages
- `def`: define functions or classes
- `return`: return results from a function
- `del`: remove an object

As we saw earlier with `+` (`__add__`), operators are examples of generic function OOP, where the appropriate method of the class of the first operand is called.

```
x = np.array([0,1,2])
x - 1
```

```
array([-1,  0,  1])
```

```
x.__sub__(1)
```

```
array([-1,  0,  1])
```

```
x
```

```
array([0, 1, 2])
```

Note that the use of the operator does not modify the object.

(Note that you can use `return(x)` and `del(x)` but behind the scenes the Python interpreter is intepreting those as `return x` and `del x`.)

Map operations

A *map* operation takes a function and runs the function on each element of some collection of items, analogous to a mathematical map. This kind of operation is very commonly used in programming, particularly functional programming, and often makes for clean, concise, and readable code.

Python provides a variety of map-type functions: `map` (a built-in) and `pandas.apply`. These are examples of higher-order functions – functions that take a function as an argument. Another map-type operation is *list comprehension*, shown here:

```
x = [1,2,3]
y = [pow(val, 2) for val in x]
y
```

```
[1, 4, 9]
```

In Python, `map` is run on the elements of an *iterable* object. Such objects include lists as well as the result of `range()` and other functions that produce iterables.

```
x = [1.0, -2.7, 3.5, -5.1]
list(map(abs, x))
```

```
[1.0, 2.7, 3.5, 5.1]
```

```
list(map(pow, x, [2,2,2,2]))
```

```
[1.0, 7.290000000000001, 12.25, 26.009999999999998]
```

Or we can use *lambda* functions to define a function on the fly:

```
x = [1.0, -2.7, 3.5, -5.1]
result = list(map(lambda vals: vals * 2, x))
```

A lambda function is a temporary function that is defined "on-the-fly" rather than in advance and is never given a name. (These are also sometimes called *anonymous* functions.)

If you need to pass another argument to the function you can use a lambda function as above or `functools.partial`:

```
from functools import partial

# Create a new round function with 'ndigits' argument pre-set
round3 = partial(round, ndigits = 3)

# Apply the function to a list of numbers
list(map(round3, [32.134234, 7.1, 343.7775]))
```

```
[32.134, 7.1, 343.777]
```

Let's compare using a map-style operation (with Pandas) to using a for loop to run a stratified analysis for a generic example (this code won't run because the variables don't exist):

```
# stratification
subsets = df.groupby('grouping_variable')

# map using pandas.apply: one line, easy to understand
results = subsets.apply(analysis_function)

# for loop: needs storage set up and multiple lines
results <- []
for _,subset in subsets:   # iterate over the key-value pairs (the subsets)
  results.append(analysis_function(subset))
```

Map operations are also at the heart of the famous *MapReduce* framework, used in Hadoop and Spark for big data processing.

Function evaluation, frames, and the call stack

Overview

When we run code, we end up calling functions inside of other function calls. This leads to a nested series of function calls. The series of calls is the *call stack*. The stack operates like a stack of cafeteria trays - when a function is called, it is added to the stack (pushed) and when it finishes, it is removed (popped).

Understanding the series of calls is important when reading error messages and debugging. In Python, when an error occurs, the call stack is shown, which has the advantage of giving the complete history of what led to the error and the disadvantage of producing often very verbose output that can be hard to understand. (In contrast, in R, only the function in which the error occurs is shown, but you can see the full call stack by invoking `traceback()`.)

What happens when an Python function is evaluated?

- The user-provided function arguments are evaluated in the calling scope and the results are matched to the argument names in the function definition.
- A new frame containing a new namespace is created to store information related to the function call and placed on the stack. Assignment to the argument names is done in the namespace, including any default arguments.
- The function is evaluated in the (new) local scope. Any look-up of variables not found in the local scope (using the namespace that was created) is done using the lexical scoping rules to look in the series of enclosing scopes (if any exist), then in the global/module scope, and then in the built-ins scope.
- When the function finishes, the return value is passed back to the calling scope and the frame is taken off the stack. The namespace is removed, unless the namespace is the enclosing scope for an existing namespace.

I'm not expecting you to fully understand that previous paragraph and all the terms in it yet. We'll see all the details as we proceed through this Unit.

Frames and the call stack

Python keeps track of the call stack. Each function call is associated with a frame that has a *namespace* that contains the local variables for that function call.

There are a bunch of functions that let us query what frames are on the stack and access objects in particular frames of interest. This gives us the ability to work with objects in the frame from which a function was called.

We can use functions from the `traceback` package to query the call stack.

```
import traceback

def function_a():
    function_b()  # line 2 within function, line 4 overall

def function_b():
    # some comment
    # another comment
    function_c()  # line 4 within function, line 9 overall

def function_c():
    traceback.print_stack() # line 2 within, line 12 overall
    # raise RuntimeError("A fake error")

function_a()    # line 1 relative to the call, line 15 overall
```

If we run that in Python (not via rendering the qmd file directly, because the line numbering shown is strange when things go through the quarto rendering process), we see this:

```
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in function_a
  File "<stdin>", line 4, in function_b
  File "<stdin>", line 2, in function_c
```

If we run the code by putting it in a module, say `trace.py` and running `python trace.py`, we see this, with the line numbering relative to the overall file, but showing the same stack of function calls:

```
  File "file.py", line 15, in <module>
    function_a()    # line 1 relative to the call, line 15 overall
  File "file.py", line 4, in function_a
    function_b()  # line 2 within function, line 4 overall
  File "file.py", line 9, in function_b
    function_c()  # line 4 within function, line 9 overall
  File "file.py", line 12, in function_c
    traceback.print_stack() # line 2 within, line 12 overall
```

If we comment out the `print_stack()` call and uncomment the error, we see the same traceback information. That's exactly what is happening when you get a long series of information when Python stops with an error.

Function inputs and outputs

Arguments

You can see the arguments (and any default values) for a function using the help system.

Let's create an example function:

```
def add(x, y, z=1, absol=False):
    if absol:
        return abs(x+y+z)
    else:
        return x+y+z
```

When defining a function, arguments without defaults must come first.

When using a function, there are some rules that must be followed.

First, users must provide values for arguments without defaults.

```
add(3, 5)
```

```
9
```

```
add(3, 5, 7)
```

```
15
```

```
add(3, 5, absol=True, z=-5)
```

```
3
```

```
add(z=-5, x=3, y=5)
```

```
3
```

```
try:
    add(3)
except Exception as error:
    print(error)
```

```
add() missing 1 required positional argument: 'y'
```

Second, arguments can be specified by position (based on the order of the inputs) or by name (keyword), using `name=value`. The user needs to provide positional arguments first.

```
add(z=-5, 3, 5)  ## Can't trap `SyntaxError` with `try`
# SyntaxError: positional argument follows keyword argument
```

Functions may have unspecified arguments, which are designated using `*args`. ('args' is a convention - you can call it something else). Unspecified arguments occurring at the beginning of the argument list are generally a collection of like objects that will be manipulated (consider `print`).

Here's an example where we see that we can manipulate *args*, which is a tuple, as desired.

```
def sum_args(*args):
    print(args[2])
    total = sum(args)
    return total

result = sum_args(1, 2, 3, 4, 5)
```

```
3
```

```
print(result)  # Output: 15
```

```
15
```

This syntax also comes in handy for some existing functions, such as `os.path.join`, which can take either an arbitrary number of inputs or a list.

```
os.path.join('a','b','c')
```

```
'a/b/c'
```

```
x = ['a','b','c']
os.path.join(*x)
```

```
'a/b/c'
```

`*args` only handles positional arguments. You'd need to also have `**kwargs` as an argument to handle keyword arguments. In the function, `args` will be a tuple while `kwargs` will be a dictionary.

Function outputs

`return x` will specify `x` as the output of the function. `return` can occur anywhere in the function, and allows the function to exit as soon as it is done.

We can return multiple outputs using `return` - the return value will then be a tuple.

```
def f(x):
    if x < 0:
        return -x**2
    else:
        res = x^2
        return x, res


f(-3)
```

```
-9
```

```
f(3)
```

```
(3, 1)
```

```
out1,out2 = f(3)
```

If you want a function to be invoked for its side effects, you can omit `return` or explicitly have `return None` or simply `return`.

Pass by value vs. pass by reference

When talking about programming languages, one often distinguishes *pass-by-value* and *pass-by-reference*.

Pass-by-value means that when a function is called with one or more arguments, a copy is made of each argument and the function operates on those copies. In pass-by-value, changes to an argument made within a function do not affect the value of the argument in the calling environment.

Pass-by-reference means that the arguments are not copied, but rather that information is passed allowing the function to find and modify the original value of the objects passed into the function. In pass-by-reference changes inside a function can affect the object outside of the function.

Pass-by-value is elegant and modular in that functions do not have side effects - the effect of the function occurs only through the return value of the function. However, it can be inefficient in terms of the amount of computation and of memory used. In contrast, pass-by-reference is more efficient, but also more dangerous and less modular. It's more difficult to reason about code that uses pass-by-reference because effects of calling a function can be hidden inside the function. Thus pass-by-value is directly related to functional programming.

Arrays and other non-scalar objects in Python are pass-by-reference (but note that tuples are immutable, so one could not modify a tuple that is passed as an argument).

```
def myfun(x):
    x[1] = 99

y = [0, 1, 2]
z = myfun(y)
type(z)
```

```
<class 'NoneType'>
```

```
y
```

```
[0, 99, 2]
```

Let's see what operations cause arguments modified in a function to affect state outside of the function:

```
def myfun(f_scalar, f_x, f_x_new, f_x_newid, f_x_copy):
    print(id(f_scalar))
    print(id(f_x))
    # `scalar` in global unaffected, as `f_scalar` is redefined.
    f_scalar = 99

    # `x` in global is MODIFIED, as `x` and `f_x` have same id.
    f_x[0] = 99
```

```
    # `x_new` in global unaffected as `f_x_new` is redefined.
    f_x_new = [99,2,3]

    # `x_newid` in global is MODIFIED as `x_newid` and `y` have same id.
    y = f_x_newid
    y[0] = 99

    # `x_copy` in global is unaffected as `x_copy` has different id from `z`
    z = f_x_copy.copy()
    z[0] = 99


scalar = 1
x = [1,2,3]
x_new = [1,2,3]
x_newid = [1,2,3]
x_copy = [1,2,3]

print(id(scalar))
```

```
133244272818992
```

```
print(id(x))
```

```
133243985151744
```

```
myfun(scalar, x, x_new, x_newid, x_copy)
```

```
133244272818992
133243985151744
```

Here are the cases where state is preserved:

```
scalar
```

```
1
```

```
x_new
```

```
[1, 2, 3]
```

```
x_copy
```

```
[1, 2, 3]
```

And here are the cases where state is modified:

```
x
```

```
[99, 2, 3]
```

```
x_newid
```

```
[99, 2, 3]
```

Basically if you replace the reference (object name) then the state outside the function is preserved. That's because a new local variable in the function scope is created. If you modify part of the object, state is not preserved.

The same behavior occurs with other mutable objects such as numpy arrays.

Pointers (optional)

To put pass-by-value vs. pass-by-reference in a broader context, I want to briefly discuss the idea of a pointer, common in compiled languages such as C.

```
int x = 3;
int* ptr;
ptr = &x;
*ptr * 7; // returns 21
```

- `int x=3` declares `x` to be an int and immediately defines it to have the value 3.
- The `int*` declares `ptr` to be a pointer to (the address of) the integer `x`.
- `&x` gets the address where `x` is stored.
- `*ptr` dereferences `ptr`, returning the value in that address (which is 3 since `ptr` is the address of `x`.

Arrays in C are really pointers to a block of memory:

```
int x[10];
```

In this case `x` will be the address of the first element of the array. We can access the first element as `x[0]` or `*x`.

Why have we gone into this? In C, you can pass a pointer as an argument to a function. The result is that only the scalar address is copied and not the entire object, and inside the function, one can modify the original object, with the new value persisting on exit from the function. For example in the following example one passes in the address of an object and that object is then modified in place, affecting its value when the function call finishes.

```
int myCal(int* ptr){
    *ptr = *ptr + *ptr;
}

myCal(&x)  # x itself will be modified
```

So Python behaves similarly to the use of pointers in C.

Namespaces and scopes

As discussed here in the Python docs, a *namespace* is a mapping from names to objects that allows Python to find objects by name via clear rules that enforce modularity and avoid name conflicts.

Namespaces are created and removed through the course of executing Python code. When a function is run, a namespace for the local variables in the function is created, and then deleted when the function finishes executing. Separate function calls (including recursive calls) have separate namespaces.

Scope is closely related concept – a scope determines what namespaces are accessible from a given place in one's code. Scopes are nested and determine where and in what order Python searches the various namespaces for objects.

Note that the ideas of namespaces and scopes are relevant in most other languages, though the details of how they work can differ.

These ideas are very important for modularity, isolating the names of objects to avoid conflicts.

This allows you to use the same name in different modules or submodules, as well as different packages using the same name.

Of course to make the objects in a module or package available we need to use `import`.

Consider what happens if you have two modules that both use `x` and you import `x` using `from`.

```
from mypkg.mymod import x
from mypkg.mysubpkg import x
x  # which x is used?
```

We've added `x` twice to the namespace of the global scope. Are both available? Did one 'overwrite' the other? How do I access the other one?

This is much better:

```
import mypkg
mypkg.x
```

```
7
```

```
import mypkg.mysubpkg
mypkg.mysubpkg.x
```

```
999
```

Side note: notice that `import mypkg` causes the name `mypkg` itself to be in the current (global) scope.

We can see the objects in a given namespace/scope using `dir()`.

```
xyz = 7
dir()
```

```
['Bear', 'GrizzlyBear', 'MyTimer', '__annotations__', '__builtins__', '__doc__', '__loader__', '__nam
```

```
import mypkg
dir(mypkg)
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__pat
```

```
import mypkg.mymod
dir(mypkg.mymod)
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spe
```

```
import builtins
dir(builtins)
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BaseExceptionGroup', 'Block
```

Here are the key scopes to be aware of, in order ("LEGB") of how the namespaces are searched:

- **L**ocal scope: objects available within function (or class method).
- non-local (**E**nclosing) scope: objects available from functions enclosing a given function (we'll talk about this more later; this relates to *lexical scoping*).
- **G**lobal (aka 'module') scope: objects available in the module in which the function is defined (which may simply be the default global scope when you start the Python interpreter). This is also the local scope if the code is not executing inside a function.
- **B**uilt-ins scope: objects provided by Python through the built-ins module but available from anywhere.

Note that `import` adds the name of the imported module to the namespace of the current (i.e., local) scope.

We can see the local and global namespaces using `locals()` and `globals()`.

```
cat local.py
```

```
gx = 7

def myfun(z):
    y = z*3
    print("local: ", locals())
    print("global: ", globals())
```

Run the following code to see what is in the different namespaces:

```
import local

gx = 99
local.myfun(3)
```

Strangely (for me being more used to R, where package namespaces are 'locked' such that objects can't be added), we can add an object to a namespace created from a module or package:

```
mymod.x = 33
dir(mymod)
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spe
```

```
import numpy as np
np.x = 33
'x' in dir(np)
```

```
True
```

As more motivation, consider this example.

Suppose we have this code in a module named `test_scope.py`:

```
cat test_scope.py
```

```
magic_number = 2

def transform(x):
    return x*5

def myfun(val):
    return(transform(val) * magic_number)
```

Now suppose we also define `magic_number` and `transform()` in the scope in which `myfun` is called from.

```
import test_scope
magic_number = 900
def transform(x):
   print("haha")

test_scope.myfun(3)
```

```
30
```

We see that Python uses `magic_number` and `transform()` from the module. What would be bad about using `magic_number` or `transform()` from the scope of the Python session rather than the scope of the module?

Consider a case where instead of using the `test_scope.py` module we were using code from a package and that code has a variable called `magic_number` or function called `transform`.

Lexical scoping and enclosing scopes

In this section, we seek to understand what happens in the following circumstance. Namely, where does Python get the value for the object `x`?

```
def f(y):
  return x + y

f(3)
```

The answer depends on where `f` is defined. If it is defined in a nested fashion within another function (or functions), then the lookup occurs in the (enclosing) scope(s) of those functions.

The enclosing scope is the scope in which a function is defined, not the scope from which a function is called.

Once the enclosing scopes are searched, if the object name is not found, then Python looks in the global/module scope where the function is defined. This is not considered part of the enclosing scope, but it does still follow the rule of looking where the function is defined, not where it is called from.

This approach is called *lexical scoping*. R and many other languages also use lexical scoping.

The behavior of looking up object names based on where functions are defined rather than where they are called from extends the local-global scoping discussed in the previous section, with similar motivation.

Let's dig deeper to understand where Python looks for non-local variables, illustrating lexical scoping and the LEGB rule:

```
## Case 1
x = 3
def f2():
    print(x)

def f():
    x = 7
    f2()

f() # what will happen?

## Case 2
x = 3
def f2()
    print(x)

def f():
    x = 7
    f2()

x = 100
f() # what will happen?

## Case 3
x = 3
def f():
    def f2():
        print(x)
    x = 7
    f2()

x = 100
f() # what will happen?

## Case 4
x = 3
def f():
    def f2():
        print(x)
```

```
    f2()

x = 100
f() # what will happen?
```

Here's a tricky example:

```
y = 100
def fun_constructor():
    y = 10
    def g(x):
        return x + y
    return g

## fun_constructor() creates functions
myfun = fun_constructor()
myfun(3)
```

Let's work through this:

1. What is the enclosing scope for the function `g()`?
2. Which `y` does `g()` use?
3. Where is `myfun` defined (this is tricky – how does `myfun` relate to `g`)?
4. What is the enclosing scope for `myfun()`?
5. When `fun_constructor()` finishes, does its scope (and namespace) disappear? What would happen if it did?
6. What does `myfun` use for `y`?

We can use the `inspect` package to see information about the closure.

```
import inspect
inspect.getclosurevars(myfun)
```

```
ClosureVars(nonlocals={}, globals={}, builtins={'id': <built-in function id>, 'print': <built-in func
```

(Note that I haven't fully investigated the use of `inspect`, but it looks like it has a lot of useful tools.)

Be careful when using variables from non-local scopes as the value of that variable may well not be what you expect it to be. In general one wants to think carefully before using variables that are taken from outside the local scope, but in some cases it can be useful.

Next we'll see some ways of accessing variables outside of the local scope.

Global and non-local variables

We can create and modify global variables and variables in the enclosing scope using `global` and `nonlocal` respectively. Note that *global* is in the context of the current module so this could be a variable in your current Python session if you're working with functions defined in that session or a global variable in a module or package.

```
del x

def myfun():
    global x
    x = 7

myfun()
print(x)
```

```
7
```

```
x = 9
myfun()
print(x)
```

```
7
```

```
def outer_function():
    x = 10  # Outer variable
    def inner_function():
        nonlocal x
        x = 20  # Modify the outer variable.
    print(x)  # Output: 10
    inner_function()
    print(x)  # Output: 20

outer_function()
```

```
10
20
```

In R, one can do similar things using the global assignment operator `<<-`.

Closures

One way to associate data with functions is to use a *closure*. This is a functional programming way to achieve something like an OOP class. This Wikipedia entry nicely summarizes the idea, which is a general functional programming idea and not specific to Python.

Using a closure involves creating one (or more functions) within a function call and returning the function(s) as the output. When one executes the original function (the constructor), the new function(s) is created and returned and one can then call that function(s). The function then can access objects in the enclosing scope (the scope of the constructor) and can use `nonlocal` to assign into the enclosing scope, to which the function (or the multiple functions) have access. The nice thing about this compared to using a global variable is that the data in the closure is bound up with the function(s) and is protected from being changed by the user.

```
x = np.random.normal(size = 5)
def scaler_constructor(data):
```

```
    def g(param):
        return param * data
    return g

scaler = scaler_constructor(x)
del x # to demonstrate we no longer need x
scaler(3)
```

```
array([-4.5020648 , -3.25168752, -4.05046623,  0.66985868,  3.50796174])
```

```
scaler(6)
```

```
array([-9.0041296 , -6.50337504, -8.10093246,  1.33971736,  7.01592347])
```

So calling `scaler(3)` multiplies 3 by the value of `data` stored in the closure (the namespace of the enclosing scope) of the function `scaler`.

Note that it can be hard to inspect the memory use involved in the closure.

Here's a more realistic example. There are other ways you could do this, but this is slick:

```
def make_container(n):
    x = np.zeros(n)
    i = 0
    def store(value = None):
        nonlocal x, i
        if value is None:
            return x
        else:
            x[i] = value
            i += 1
    return store


nboot = 20
bootmeans = make_container(nboot)

import pandas as pd
iris = pd.read_csv('https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/io/data/c
data = iris['SepalLength']

for i in range(nboot):
    bootmeans(np.mean(np.random.choice(data, size = len(data), replace = True)))


bootmeans()

bootmeans.__closure__
```

Closures are also used as "function factories" (where the outer (generator) function is also called a "factory function") to easily generate a set of related functions. Here's an example:

```
def number_formatter(notation='US'):
    """
    Creates a closure for formatting decimal numbers.

    Args:
        notation (str): 'US' for US notation (commas for thousands, period for decimal)
                        'EU' for European notation (periods for thousands, comma for decimal)

    Returns:
        function: A closure that formats numbers according to the specified notation
    """
    def format_number(number):
        ## GitHub Copilot suggested the `number:,` syntax and the string replace approach.
        result = f"{number:,}"
        if notation == 'US':
            # US notation: 1,234.56
            return result
        elif notation == 'EU':
            # European notation: 1.234,56
            # Swap commas and periods
            return result.replace(',', 'TEMP').replace('.', ',').replace('TEMP', '.')
        else:
            raise ValueError("Notation must be 'US' or 'EU'")

    return format_number

us_printer = number_formatter('US')
eu_printer = number_formatter('EU')

us_printer(1234.56)
```

```
'1,234.56'
```

```
eu_printer(1234.56)
```

```
'1.234,56'
```

Decorators

Now that we've seen function generators, it's straightforward to discuss *decorators*.

A decorator is a wrapper around a function that extends the functionality of the function without actually modifying the function.

We can create a simple decorator "manually" like this:

```
def verbosity_wrapper(myfun):
    def wrapper(*args, **kwargs):
        print(f"Starting {myfun.__name__}.")
        output = myfun(*args, **kwargs)
        print(f"Finishing {myfun.__name__}.")
        return output
    return wrapper

verbose_rnorm = verbosity_wrapper(np.random.normal)

x = verbose_rnorm(size = 5)
```

```
Starting normal.
Finishing normal.
```

```
x
```

```
array([ 1.07413602,  0.20128607, -1.09658699, -1.93034821, -2.11644448])
```

Python provides syntax that helps you create decorators with less work (this is an example of the general idea of *syntactic sugar*).

We can easily apply our decorator defined above to a function as follows. Now the function name refers to the wrapped version of the function.

```
@verbosity_wrapper
def myfun(x):
    return x

y = myfun(7)
```

```
Starting myfun.
Finishing myfun.
```

```
y
```

```
7
```

Our decorator doesn't do anything useful, but hopefully you can imagine that the idea of being able to have more control over the operation of functions could be useful. For example we could set up a timing wrapper so that when we run a function, we get a report on how long it took to run the function. Or using the idea of a closure, we could keep a running count of the number of times a function has been called.

One real-world example of using decorators is in setting up functions to run in parallel in `dask`, which we'll discuss in Unit 7. Another is the use of Numba to do just-in-time (JIT) compilation of Python code.

8. Memory and copies

Overview

The main things to remember when thinking about memory use are: (1) numeric arrays take 8 bytes per element and (2) we need to keep track of when large objects are created, including local variables in the frames of functions. To do (2) we need to know when a copy of an actual object is created versus when we simply copy the name/label/reference to an existing object.

```
x = np.random.normal(size = 5)
x.itemsize # 8 bytes
```

```
8
```

```
x.nbytes
```

```
40
```

Allocating and freeing memory

Unlike compiled languages like C, in Python we do not need to explicitly allocate storage for objects. (However, we will see that there are times that we do want to allocate storage in advance, rather than successively concatenating onto a larger object.)

Python automatically manages memory, releasing memory back to the operating system when it's not needed via a process called *garbage collection*. Very occasionally you may want to remove large objects as soon as they are not needed. `del` does not actually free up memory, it just disassociates the name from the memory used to store the object. In general Python will quickly clean up such objects without a reference (i.e., a name), so there is generally no need to call `gc.collect()` to force the garbage collection.

In a language like C in which the user allocates and frees up memory, memory leaks are a major cause of bugs. Basically if you are looping and you allocate memory at each iteration and forget to free it, the memory use builds up inexorably and eventually the machine runs out of memory. In Python, with automatic garbage collection, this is generally not an issue, but occasionally memory leaks could occur.

The heap and the stack

The *heap* is the memory that is available for dynamically creating new objects while a program is executing, e.g., if you create a new object in Python or call *new* in C++. When more memory is needed the program can request more from the operating system. When objects are removed in Python, Python will handle the garbage collection of releasing that memory.

The *stack* is the memory used for local variables when a function is called. I.e., the stack is the memory associated with function frames. Hence the term "stack trace" to refer to the active function frames during execution of code.

There's a nice discussion of this on this Stack Overflow thread.

Monitoring memory use

Monitoring overall memory use on a UNIX-style computer

To understand how much memory is available on your computer, one needs to have a clear understanding of disk caching. The operating system will generally cache files/data in memory when it reads from disk. Then if that information is still in memory the next time it is needed, it will be much faster to access it the second time around than if it had to read the information from disk. While the cached information is using memory, that same memory is immediately available to other processes, so the memory is available even though it is "in use".

We can see this via `free -h --si`.

The `-h` is for 'human-readable' for nicer numerical printout and the `--si` uses gigabytes (base 10) instead of gibibytes (base 2).

```
paciorek@gandalf:~> free -h --si
               total        used        free      shared  buff/cache   available
Mem:            135G         24G         30G        7.8M         80G        110G
Swap:           274G         44G        230G
```

You'll generally be interested in the `Mem` row. (See below for some comments on `Swap`.) The `shared` column is complicated and probably won't be of use to you. The `buff/cache` column shows how much space is used for disk caching and related purposes but is actually available. Hence the `available` column is the sum of the `free` and `buff/cache` columns (more or less). In this case 24 GB is in use (indicated in the `used` column), leaving 110 GB available for use.

`top` (Linux or Mac) shows both total memory use and statistics by process.

Here are some example lines from the first few lines of output from `top`:

```
MiB Mem : 128877.0 total,  28825.9 free,  23856.4 used,  77249.1 buff/cache
MiB Swap: 262144.0 total, 220103.3 free,  42040.7 used. 105020.6 avail Mem
```

We see that this machine has 129 GiB RAM (the 'total' column in the `Mem` row), with 106 GiB available (29 GiB free plus 77 GiB buff/cache as seen in the `Mem` row). 24 GiB is in use.

Swap is essentially the reverse of disk caching. It is disk space that is used for memory when the machine runs out of physical memory. You generally don't want your machine to be using swap for memory because your jobs can slow to a crawl. As seen above, the `swap` line in `top` shows 262 GiB swap space (274 GB in `free`), with 42 GiB (44 GB) in use. One would often hope that little swap space is in use, but this can get complicated. In this case I'm not sure why 42 GiB of swap is being used given there is plenty of physical memory available.

Note that some of the numbers differ a bit between `top` and `free`, more so that can be explained based on gigabytes vs. gibibytes. I'm not sure why that is, but it doesn't affect our overall assessment of memory used and available here.

Monitoring memory use in Python

There are a number of ways to see how much memory is being used. When Python is actively executing statements, you can use `top` from the UNIX shell.

In Python, we can call out to the system to get the info we want:

```
import psutil

# Get memory information
memory_info = psutil.Process().memory_info()

# Print the memory usage
print("Memory usage:", memory_info.rss/10**6, " MB.")
```

```
Memory usage: 329.97376  MB.
```

```
# Let's turn that into a function for later use:
def mem_used():
    print("Memory usage:", psutil.Process().memory_info().rss/10**6, " MB.")
```

We can see the size of an object (in bytes) with `sys.getsizeof()`.

```
my_list = [1, 2, 3, 4, 5]
sys.getsizeof(my_list)
```

```
104
```

```
x = np.random.normal(size = 10**7) # should use about 80 MB
sys.getsizeof(x)
```

```
80000112
```

However, we need to be careful about objects that refer to other objects:

```
y = [3, x]
sys.getsizeof(y)  # Whoops!
```

```
72
```

Here's a trick where we serialize the object, as if to export it, and then see how long the binary representation is.

```
import pickle
ser_object = pickle.dumps(y)
sys.getsizeof(ser_object)
```

```
80000202
```

There are also some flags that one can start `python` with that allow one to see information about memory use and allocation. See `man python`. You could also look into the `memory_profiler` or `pympler` packages.

How memory is used in Python

Two key tools: `id` and `is`

We can use the `id` function to see where in memory an object is stored and `is` to see if two object are actually the same objects in memory. It's particularly useful for understanding storage and memory use for complicated data structures. We'll also see that they can be handy tools for seeing where copies are made and where they are not.

```
x = np.random.normal(size = 10**7)
id(x)
```

```
133243985423856
```

```
sys.getsizeof(x)
```

```
80000112
```

```
y = x
id(y)
```

```
133243985423856
```

```
x is y
```

```
True
```

```
sys.getsizeof(y)
```

```
80000112
```

```
z = x.copy()
id(z)
```

```
133244033566640
```

```
sys.getsizeof(z)
```

```
80000112
```

Memory use in specific circumstances

How lists are stored

Here we can use `id` to determine how the overall list is stored as well as the elements of the list.

```
nums = np.random.normal(size = 5)
obj = [nums, nums, np.random.normal(size = 5), ['adfs']]

id(nums)
```

```
133244200598864
```

```
id(obj)
```

```
133243985602304
```

```
id(obj[0])
```

```
133244200598864
```

```
id(obj[1])
```

```
133244200598864
```

```
id(obj[2])
```

```
133244200586864
```

```
id(obj[3])
```

```
133243987960832
```

```
id(obj[3][0])
```

```
133244027806672
```

```
obj[0] is obj[1]
```

```
True
```

```
obj[0] is obj[2]
```

```
False
```

What do we notice?

- The list itself appears to be a array of references (pointers) to the component elements.
- Each element has its own address.
- Two elements of a list can use the same memory (see the first two elements here, whose contents are at the same memory address).
- A list element can use the same memory as another object (or part of another object).

A note about calling `id` on list elements, e.g., `id(obj[0])`. Running that code causes execution of `obj[0]`, and the result is passed into `id`. That creates a new temporary object that is passed to `id`. The temporary object references the same object that `obj[0]` does, so we find out the id of `obj[0]`.

How character strings are stored.

Similar tricks are used for storing strings (and also integers). We may explore this in a problem set problem.

How numpy arrays are stored

To do vectorized calculations and linear algebra efficiently, one needs the data in arrays stored contiguously in memory, and this is how numpy arrays are stored. We've seen that a numpy array involves the actual numbers plus 112 bytes of metadata/overhead associated with the object.

We can't usefully call `id` on elements of the array. Consider this:

```
x = np.random.normal(size=5)
type(x[1])
```

```
<class 'numpy.float64'>
```

```
id(x[1])
```

```
133244027652912
```

```
tmp1 = x[1]
tmp2 = x[1]
id(tmp1)
```

```
133244027660336
```

```
id(tmp2)
```

```
133244027661328
```

Each time we get the `x[1]` element we are creating a new numpy float64 scalar object. This is because each element of the array is not its own object (unlike a list element) so extracting the element involves copying the value and making a new object.

Another indication of the array being stored contiguously is that if we pickle the array, its size is just a bit more than the size needed just to store the 8-byte numbers. If we instead pickle a list containing those same numbers, the result is more than twice as big, related to the pointers involved in referring to the individual numbers.

Modifying elements in place

What do this simple experiment tell us?

```
x = np.random.normal(size = 5)
id(x)
```

```
133244032959856
```

```
x[2] = 3.5
id(x)
```

```
133244032959856
```

It makes some sense that modifying elements of an object here doesn't cause a copy – if it did, working with large objects would be very difficult.

Shallow copying

A *shallow* copy only makes a new top-level object. The elements within the object are still shared with the original object.

```
x = [3, [1,2,3]]
y = x.copy()
id(x)
```

```
133243987962112
```

```
id(y)
```

```
133243986850368
```

```
x[0] = 9
y[0]
```

```
3
```

```
id(x[1])
```

```
133243985626816
```

```
id(y[1])
```

```
133243985626816
```

```
y[1][2] = 99
x
```

```
[9, [1, 2, 99]]
```

We can force a *deep* copy such that nothing is shared between the objects.

```
import copy
x = [3, [1,2,3]]
y = copy.deepcopy(x)

id(x[1])
```

```
133243985667200
```

```
id(y[1])
```

```
133243985630336
```

```
y[1][2] = 99
x
```

```
[3, [1, 2, 3]]
```

When are copies made?

Let's try to understand when Python uses additional memory for objects, and how it knows when it can delete memory. We'll use large objects so that we can use `free` or `top` to see how memory use by the Python process changes.

```
x = np.random.normal(size = 10**8)
id(x)
```

```
133244210342448
```

```
y = x
id(y)
```

```
133244210342448
```

```
x = np.random.normal(size = 10**8)
id(x)
```

```
133243985416368
```

Only if we re-assign `x` to reference a different object does additional memory get used.

How does Python know when it can free up memory?

Python keeps track of how many names refer to an object and only removes memory when there are no remaining references to an object.

```
import sys

x = np.random.normal(size = 10**8)
y = x
sys.getrefcount(y)
```

```
3
```

```
del x
sys.getrefcount(y)
```

```
2
```

```
del y
```

We can see the number of references using `sys.getrefcount`. Confusingly, the number is one higher than we'd expect, because it includes the temporary reference from passing the object as the argument to `getrefcount`.

```
x = np.random.normal(size = 5)
sys.getrefcount(x)  # In reality, only 1.
```

```
2
```

```
y = x
sys.getrefcount(x)  # In reality, 2.
```

```
3
```

```
sys.getrefcount(y)  # In reality, 2.
```

```
3
```

```
del y
sys.getrefcount(x)  # In reality, only 1.
```

```
2
```

```
y = x
x = np.random.normal(size = 5)
sys.getrefcount(y)  # In reality, only 1.
```

```
2
```

```
sys.getrefcount(x)  # In reality, only 1.
```

```
2
```

This notion of reference counting occurs in other contexts, such as shared pointers in C++ and in how R handles copying and garbage collection.

Strategies for saving memory

A few basic strategies for saving memory include:

- Avoiding unnecessary copies.
- Removing objects that are not being used, at which point the Python garbage collector should free up the memory.
- Use iterators (e.g., `range()`) and generators to avoid building long lists that are iterated over.

If you're really trying to optimize memory use, you may also consider:

- Using types that take up less memory (e.g., `Bool`, `Int16`, `Float32`) when possible.

  ```
  x = np.array(np.random.normal(size = 5), dtype = "float32")
  x.itemsize
  ```

  ```
  4
  ```

  ```
  x = np.array([3,4,2,-2], dtype = "int16")
  x.itemsize
  ```

  ```
  2
  ```

- Reading data in from files in chunks rather than reading the entire dataset (more in Unit 7).
- Exploring packages such as `arrow` for efficiently using memory, as discussed in Unit 2.

Example

Let's work through a real example where we keep a running tally of current memory in use and maximum memory used in a function call. We'll want to consider hidden uses of memory and when copies of the object are made rather than simply creating a new reference to an existing object. This code (translated from the original R code) comes from a PhD student's research. For our purposes here, let's assume that the input `x` and `y` are very long numpy arrays using a lot of memory.

`np.isnan` returns an boolean array of True/False values indicating whether each input element is a NaN (not a number).

```
def fastcount(xvar, yvar):
    naline = np.isnan(xvar)
    naline[np.isnan(yvar)] = True
    localx = xvar.copy()
    localy = yvar.copy()
    localx[naline] = 0
    localy[naline] = 0
    useline = ~naline
    ## We'll ignore the rest of the code.
    ## ....

fastcount(x, y) # Assume x, y are existing large numpy arrays.
```

> Challenge
>
> In the code above, note all the places where new memory must be allocated.

9. Efficiency

Interpreters and compilation

Why are interpreted languages slow?

Compiled code runs quickly because the original code has been translated into instructions (machine language) that the processor can understand (i.e., zeros and ones). In the process of doing so, various checking and lookup steps are done once and don't need to be redone when running the compiled code.

In contrast, when one runs code in an interpreted language such as Python or R, the interpreter needs to do all the checking and lookup each time the code is run. This is required because the types and locations in memory of the variables could have changed.

We'll focus on Python in the following discussion, but most of the concepts apply to other interpreted languages.

For example, consider this code:

```
x = 3
abs(x)
x*7

x = 'hi'
abs(x)
x*3
```

Because of dynamic typing, when the interpreter sees `abs(x)` it needs to check if `x` is something to which the absolute value function can be applied, including dealing with the fact that `x` could be a list or array with many numbers in it. In addition it needs to (using scoping rules) look up the value of `x`.

(Consider that `x` might not even exist at the point that `abs(x)` is called.) Only then can the absolute value calculation happen. For the multiplication, Python needs to lookup the version of `*` that can be used, depending on the type of `x`.

Let's consider writing a loop with some ridiculous code:

```
x = np.random.normal(size = 10)
for i in range(10):
    if np.random.normal(size = 1) > 0:
        x = 'hi'
    if np.random.normal(size = 1) > 0.5:
        del x
    x[i]= np.exp(x[i])
```

There is no way around the fact that because of how dynamic this is, the interpreter needs to check if `x` exists, if it is an array of sufficient length, if it contains numeric values, and it needs to go retrieve the required value, EVERY TIME the `np.exp()` is executed. Now the code above is unusual, and in most cases, we wouldn't have the `if` statements that modify `x`. So you could imagine a process by which the checking were done on the first iteration and then not needed after that – that gets into the idea of just-in-time compilation, discussed later.

The standard Python interpreter (CPython) is a C function so in some sense everything that happens is running as compiled code, but there are lots more things being done to accomplish a given task using interpreted code than if the task had been written directly in code that is compiled. By analogy, consider talking directly to a person in a language you both know compared to talking to a person via an interpreter who has to translate between two languages. Ultimately, the same information gets communicated (hopefully!) but the number of words spoken and time involved is much greater.

When running more complicated functions, there is often a lot of checking that is part of the function itself. For example scipy's `solve_triangular` function ultimately calls out to the `trtrs` Lapack function, but before doing so, there is a lot of checking that can take time. To that point, the documentation suggests you might set `check_finite=False` to improve performance at the expense of potential problems if the input matrices contain troublesome elements.

We can flip the question on its head and ask what operations in an interpreted language will execute quickly. In Python, these include:

- operations that call out to compiled C code,
- linear algebra operations (these call out to compiled C or Fortran code provided by the BLAS and LAPACK software packages), and
- vectorized calls rather than loops:
  - vectorized calls generally run loops in compiled C code rather than having the loop run in Python, and
  - that means that the interpreter doesn't have to do all the checking discussed above for every iteration of the loop.

Compilation

Overview

Compilation is the process of turning code in a given language (such a C++) into machine code. Machine code is the code that the processor actually executes. The machine code is stored in the executable file, which is a binary file. The history of programming has seen ever great levels of abstraction, so that humans can write code using syntax that is easier for us to understand, re-use, and develop building blocks that can be put together to do complicated tasks. For example assembly language is a step above machine code. Languages like C and Fortran provide additional abstraction beyond that. The Statistics 750 class at CMU has a nice overview if you want to see more details.

Note that interpreters such as Python are themselves programs – the standard Python interpreter (CPython) is a C program that has been compiled. It happens to be a program that processes Python code. The interpreter doesn't turn Python code into machine code, but the interpreter itself is machine code.

Just-in-time (JIT) compilation

Standard compilation (ahead-of-time or AOT compilation) happens before any code is executed and can involve a lot of optimization to produce the most efficient machine code possible.

In contrast, just-in-time (JIT) compilation happens at the time that the code is executing. JIT compilation is heavily used in Julia, which is very fast (in some cases as fast as C). JIT compilation involves translating to machine code as the code is running. One nice aspect is that the results are cached so that if code is rerun, the compilation process doesn't have to be redone. So if you use a language like Julia, you'll see that the speed can vary drastically between the first time and later times you run a given function during a given session.

One thing that needs to be dealt with is type checking. As discussed above, part of why an interpreter is slow is because the type of the variable(s) involved in execution of a piece of code is not known in advance, so the interpreter needs to check the type. In JIT systems, there are often type inference systems that determine variable types.

JIT compilation can involve translation from the original code to machine code or translation of bytecode (see next section) to machine code.

At the end of this unit, we'll see the use of JIT compilation with the JAX package in Python.

`numba` is a standard JIT compiler for Python and numpy that uses the LLVM compiler library. To use it one applies the `numba.njit` decorator to a Python function.

Byte compiling (optional)

Functions in Python and Python packages may byte compiled. What does that mean? Byte-compiled code is a special representation that can be executed more efficiently because it is in the form of compact codes that encode the results of parsing and semantic analysis of scoping and other complexities of the Python source code. This byte code can be executed faster than the original Python code because it skips the stage of having to be interpreted by the Python interpreter.

If you look at the file names in the directory of an installed Python package you may see files with the `.pyc` extension. These files have been byte-compiled.

We can byte compile our own functions using either the `py_compile` or `compileall` modules. Here's an example (silly since as experienced Python programmers, we would use vectorized calculation here

rather than this unvectorized code.)

```
import time

def f(vals):
    x = np.zeros(len(vals))
    for i in range(len(vals)):
        x[i] = np.exp(vals[i])
    return x

x = np.random.normal(size = 10**6)
t0 = time.time()
out = f(x)
time.time() - t0
```

```
0.7561802864074707
```

```
t0 = time.time()
out = np.exp(x)
time.time() - t0
```

```
0.013027191162109375
```

```
import py_compile
py_compile.compile('vec.py')
```

```
'__pycache__/vec.cpython-312.pyc'
```

```
cp __pycache__/vec.cpython-312.pyc vec.pyc
rm vec.py    # Make sure non-compiled module not loaded.
```

```
import vec
vec.__file__
```

```
'/accounts/vis/paciorek/teaching/243fall24/fall-2024/units/vec.pyc'
```

```
t0 = time.time()
out = vec.f(x)
time.time() - t0
```

```
0.7301280498504639
```

Unfortunately, as seen above byte compiling may not speed things up much. I'm not sure why.

Benchmarking and profiling

Recall that it's a waste of time to optimize code before you determine (1) that the code is too slow for how it will be used and (2) which are the slow steps on which to focus your attempts to speed the code up. A 100x speedup in a step that takes 1% of the time will speed up the overall code by essentially nothing.

Timing your code

There are a few ways to time code:

```
import time
x = np.random.normal(size = 10**7)
t0 = time.time()
y = np.exp(x)
t1 = time.time()

print(f"Execution time: {t1-t0} seconds.")
```

```
Execution time: 0.07209277153015137 seconds.
```

In general, it's a good idea to repeat (replicate) your timing, as there is some stochasticity in how fast your computer will run a piece of code at any given moment.

```
import time
t0 = time.time()
y = np.exp(3.)
t1 = time.time()

print(f"Execution time: {t1-t0} seconds.")
```

```
Execution time: 0.0057201385498046875 seconds.
```

Using `time` is fine for code that takes a little while to run, but for code that is really fast (such as the code above), it may not be very accurate. Measuring fast bits of code is tricky to do well. This next approach is better for benchmarking code (particularly faster bits of code).

```
import timeit

timeit.timeit('x = np.exp(3.)', setup = 'import numpy as np', number = 100)
```

```
9.436404798179865e-05
```

```
code = '''
x = np.exp(3.)
'''

timeit.timeit(code, setup = 'import numpy as np', number = 100)
```

```
8.632097160443664e-05
```

That reports the **total** time for the 100 replications.

We can run it from the command line.

```
python -m timeit -n 1000 'x = 3.73 * 5.12'
python -m timeit -s 'import numpy' -n 1000 'x = numpy.exp(3.)'
python -m timeit -s 'from scipy.special import gammaln' -n 1000 'x = gammaln(3.)'
```

```
1000 loops, best of 5: 15.9 nsec per loop
```

```
1000 loops, best of 5: 628 nsec per loop
1000 loops, best of 5: 692 nsec per loop
```

`timeit` ran the code 1000 times for 5 different repetitions, giving the **average** time for the 1000 samples for the best of the 5 repetitions.

We can see that there can be large relative differences for different fundamental mathematical operations, although each one is individually fast on an absolute basis.

Profiling

The `Cprofile` module will show you how much time is spent in different functions, which can help you pinpoint bottlenecks in your code.

I haven't run this code when producing this document as the output of the profiling can be lengthy.

```
def lr_slow(y, x):
    xtx = x.T @ x
    xty = x.T @ y
    inv = np.linalg.inv(xtx)
    return inv @ xty

## generate random observations and random matrix of predictors
n = 7000
y = np.random.normal(size = 7000)
x = np.random.normal(size = (7000,1000))

t0 = time.time()
regr = lr_slow(y, x)
t1 = time.time()
print(f"Execution time: {t1-t0} seconds.")

import cProfile
cProfile.run('lr_slow(y,x)')
```

The `cumtime` column includes the time spent in nested calls to functions while the `tottime` column excludes it.

As we'll discuss in detail in Unit 10, we almost never want to explicitly invert a matrix. Instead we factorize the matrix and use the factorized result to do the computation of interest. In this case using the Cholesky decomposition is a standard approach, followed by solving triangular systems of equations.

```
import scipy as sp

def lr_fast(y, x):
    xtx = x.T @ x
    xty = x.T @ y
    L = sp.linalg.cholesky(xtx)
    out = sp.linalg.solve_triangular(L.T,
```

```
          sp.linalg.solve_triangular(L, xty, lower=True),
          lower=False)
    return out

t0 = time.time()
regr = lr_fast(y, x)
t1 = time.time()
print(f"Execution time: {t1-t0} seconds.")

cProfile.run('lr_fast(y,x)')
```

In principle, the Cholesky now dominates the computational time (but is much faster than `inv`), so there's not much more we can do in this case. That said, it's not obvious from the profiling that the fact that most of the time is in the Cholesky is the case. Interpreting the output of a profiler can be hard. In this case to investigate further I would probably time individual steps with `timeit`.

You might wonder if it's better to use `x.T` or `np.transpose(x)`. Try using `timeit` to decide.

The Python profilers (`cProfile` and `profile` (not shown)) use deterministic profiling – calculating the interval between events (i.e., function calls and returns). However, there is some limit to accuracy – the underlying 'clock' measures in units of about 0.001 seconds.

(In contrast, R's profiler works by sampling (statistical profiling) - every little while during a calculation it finds out what function R is in and saves that information to a file. So if you try to profile code that finishes really quickly, there's not enough opportunity for the sampling to represent the calculation accurately and you may get spurious results.)

Writing efficient (Python) code

We'll discuss a variety of these strategies, including:

- Pre-allocating memory rather than growing objects iteratively
- Vectorization and use of fast matrix algebra
- Consideration of loops vs. map operations
- Speed of lookup operations, including hashing

While illustrated in Python, many of the underlying ideas pertain in other contexts.

Pre-allocating memory

Let's consider whether we should pre-allocate space for the output of an operation or if it's ok to keep extending the length of an array or list.

```
n = 100000
z = np.random.normal(size = n)

## Pre-allocation

def fun_prealloc(vals):
```

```
    n = len(vals)
    x = [0] * n
    for i in range(n):
        x[i] = np.exp(vals[i])
    return x

## Appending to a list

def fun_append(vals):
    x = []
    for i in range(n):
        x.append(np.exp(vals[i]))
    return x

## Appending to a numpy array

def fun_append_np(vals):
    x = np.array([])
    for i in range(n):
        x = np.append(x, np.exp(vals[i]))
    return x


t0 = time.time()
out1 = fun_prealloc(z)
time.time() - t0
```

```
0.09515643119812012
```

```
t0 = time.time()
out2 = fun_append(z)
time.time() - t0
```

```
0.11667084693908691
```

```
t0 = time.time()
out3 = fun_append_np(z)
time.time() - t0
```

```
2.442230224609375
```

So what's going on? First let's consider what is happening with the use of `np.append`. Note that it is a function, rather than a method, and we need to reassign to `x`. What must be happening in terms of memory use and copying when we append an element?

```
x = np.random.normal(size = 5)
id(x)
```

```
133244210342448
```

```
id(np.append(x, 3.34))
```

```
133243985424624
```

We can avoid that large cost of copying and memory allocation by pre-allocating space for the entire output array. (This is equivalent to variable initialization in compiled languages.)

Ok, but how is it that we can append to the **list** at apparently no cost?

It's not magic, just that Python is clever. Let's get an idea of what is going on:

```
def fun_append2(vals):
   n = len(vals)
   x = []
   print(f"Initial id: {id(x)}")
   sz = sys.getsizeof(x)
   print(f"iteration 0: size {sz}")
   for i in range(n):
       x.append(np.exp(vals[i]))
       if sys.getsizeof(x) != sz:
           sz = sys.getsizeof(x)
           print(f"iteration {i}: size {sz}")
   print(f"Final id: {id(x)}")
   return x

z = np.random.normal(size = 1000)
out = fun_append2(z)
```

```
Initial id: 133243985991616
iteration 0: size 56
iteration 0: size 88
iteration 4: size 120
iteration 8: size 184
iteration 16: size 248
iteration 24: size 312
iteration 32: size 376
iteration 40: size 472
iteration 52: size 568
iteration 64: size 664
iteration 76: size 792
iteration 92: size 920
iteration 108: size 1080
iteration 128: size 1240
iteration 148: size 1432
iteration 172: size 1656
iteration 200: size 1912
iteration 232: size 2200
iteration 268: size 2520
```

```
iteration 308: size 2872
iteration 352: size 3256
iteration 400: size 3704
iteration 456: size 4216
iteration 520: size 4792
iteration 592: size 5432
iteration 672: size 6136
iteration 760: size 6936
iteration 860: size 7832
iteration 972: size 8856
Final id: 133243985991616
```

Surprisingly, the id of `x` doesn't seem to change, even though we are allocating new memory at many of the iterations. What is happening is that `x` is a wrapper object that contains within it a reference to an array of references (pointers) to the list elements. The location of the wrapper object doesn't change, but the underlying array of references/pointers is being reallocated.

Side note: I don't know of any way to find out the location of the underlying array of pointers. I believe that under the hood, Python uses the `realloc` system call to request memory from the operating system when it needs to use additional pointers, and that the operating system can then try to allocate the additional memory at the same location (i.e., extending the block of memory).

Note that as we discussed in the previous section on memory use, our assessment of size above does not include the actual size of the list elements, as illustrated by having one element of the list be a big object:

```
print(sys.getsizeof(out))
```

```
8856
```

```
out[2] = np.random.normal(size = 100000)
print(sys.getsizeof(out))
```

```
8856
```

> Grow objects using lists, then convert
>
> One upshot of how Python efficiently grows lists is that if you need to grow an object, use a Python list. Then once it is complete, you can always convert it to another type, such as a numpy array.

Vectorization and use of fast matrix algebra

One key way to write efficient Python code is to take advantage of numpy's vectorized operations.

```
n = 10**6
z = np.random.normal(size = n)
t0 = time.time()
```

```
x = np.exp(z)
print(time.time() - t0)
```

```
0.012758016586303711
```

```
x = np.zeros(n)  # Leave out pre-allocation timing to focus on computation.
t0 = time.time()
for i in range(n):
    x[i] = np.exp(z[i])


print(time.time() - t0)
```

```
0.7464439868927002
```

So what is different in how Python handles the calculations above that explains the huge disparity in efficiency? The vectorized calculation is being done natively in C in a for loop. The explicit Python for loop involves executing the for loop in Python with repeated calls to C code at each iteration. This involves a lot of overhead because of the repeated processing of the Python code inside the loop. For example, in each iteration of the loop, Python is checking the types of the variables because it's possible that the types might change, as discussed earlier.

You can usually get a sense for how quickly a Python call will pass things along to C or Fortran by looking at the body of the relevant function(s) being called.

Unfortunately seeing the source code in Python often involves going and finding it in a file on disk, whereas in R, printing a function will show its source code. However you can use `??` in IPython to get the code for non-builtin functions. Consider `numpy.linspace??`.

Here I found the source code for the scipy `triangular_solve` function, which calls out to a Fortran function `trtrs`, found in the LAPACK library.

```
## On an SCF machine:
/usr/local/linux/miniforge-3.12/lib/python3.12/site-packages/scipy/linalg/_basic.py
```

With a bit more digging around we could verify that `trtrs` is a LAPACK funcion by doing some grepping:

```
./linalg/_basic.py:    trtrs, = get_lapack_funcs(('trtrs',), (a1, b1))
```

Many numpy and scipy functions allow you to pass in arrays, and operate on those arrays in vectorized fashion. So before writing a for loop, look at the help information on the relevant function(s) to see if they operate in a vectorized fashion. Functions might take arrays for one or more of their arguments.

Outside of the numerical packages, we often have to manually do the looping:

```
x = [3.5, 2.7, 4.6]
try:
    math.cos(x)
except Exception as error:
    print(error)
```

```
must be real number, not list
```

```
[math.cos(val) for val in x]
```

```
[-0.9364566872907963, -0.9040721420170612, -0.11215252693505487]
```

```
list(map(math.cos, x))
```

```
[-0.9364566872907963, -0.9040721420170612, -0.11215252693505487]
```

> **Challenge**
>
> Consider the chi-squared statistic involved in a test of independence in a contingency table:
>
> $$\chi^2 = \sum_i \sum_j \frac{(y_{ij} - e_{ij})^2}{e_{ij}}, \quad e_{ij} = \frac{y_{i\cdot} y_{\cdot j}}{y_{\cdot\cdot}}$$
>
> where $y_{i\cdot} = \sum_j y_{ij}$ and $y_{\cdot j} = \sum_i y_{ij}$ and $y_{\cdot\cdot} = \sum_i \sum_j y_{ij}$. Write this in a vectorized way without any loops. Note that 'vectorized' calculations also work with matrices and arrays.

Sometimes we can exploit vectorized mathematical operations in surprising ways, though sometimes the code is uglier.

```
x = np.random.normal(size = n)

## List comprehension
timeit.timeit('truncx = [max(0,val) for val in x]', number = 10, globals = {'x':x})
```

```
1.7778889809269458
```

```
## Vectorized slice replacement
timeit.timeit('truncx = x.copy(); truncx[x < 0] = 0', number = 10, globals = {'x':x})
```

```
0.06688038306310773
```

```
## Vectorized math trick
timeit.timeit('truncx = x * x>0', number = 10, globals = {'x':x})
```

```
0.016767743974924088
```

We'll discuss what has to happen (in terms of calculations, memory allocation, and copying) in the two vectorized approaches to try to understand which is more efficient.

> **Additional efficiency tips**
>
> - If you do need to loop over dimensions of a matrix or array, if possible loop over the smallest dimension and use the vectorized calculation on the larger dimension(s). For example if you have a 10000 by 10 matrix, try to set up your problem so you can loop over the 10 columns rather than the 10000 rows.
> - In general, in Python looping over rows is likely to be faster than looping over columns

because of numpy's row-major ordering (by default, matrices are stored in memory as a long array in which values in a row are adjacent to each other). However how numpy handles this is more complicated (see more in the Section on cache-aware programming), such that it may not matter for numpy calculations.
- You can use direct arithmetic operations to add/subtract/multiply/divide a 1-d array by each column of a matrix, e.g. `A*b` does element-wise multiplication of each row of A by a 1-d array b. If you need to operate by column, you can do it by transposing the matrix.

Caution: relying on Python's broadcasting rule in the context of vectorized operations, such as is done when direct-multiplying a matrix by a 1-d array to scale the columns relative to each other, can be dangerous as the code may not be easy for someone to read and poses greater dangers of bugs. In some cases you may want to first write the code more directly and then compare the more efficient code to make sure the results are the same. It's also a good idea to comment your code in such cases.

Vectorization, mapping, and loops

Next let's consider when loops and mapping would be particularly slow and how mapping and loops might compare to each other.

First, the potential for inefficiency of looping and map operations in interpreted languages will depend in part on whether a substantial part of the work is in the overhead involved in the looping or in the time required by the function evaluation on each of the elements.

Here's an example, where the core computation is very fast, so we might expect the overhead of looping (in its various forms seen here) to be important.

```
import time
n = 10**6
x = np.random.normal(size = n)

t0 = time.time()
out = np.exp(x)
time.time() - t0
```

```
0.014786243438720703
```

```
t0 = time.time()
vals = np.zeros(n)
for i in range(n):
    vals[i] = np.exp(x[i])


time.time() - t0
```

```
1.0164167881011963
```

```
t0 = time.time()
vals = [np.exp(v) for v in x]
time.time() - t0
```

```
0.8443398475646973
```

```
t0 = time.time()
vals = list(map(np.exp, x))
time.time() - t0
```

```
0.8089804649353027
```

Regardless of how we do the looping (an explicit loop, list comprehension, or `map`), it looks like we can't avoid the overhead unless we use the vectorized call, which is of course the recommended approach in this case, both for speed and readability (and conciseness).

Second, is it faster to use `map` than to use a loop? In the example above it is somewhat (but not substantially) faster to use `map` (and still much slower than vectorization). In the loop case, the interpreter needs to do the checking we discussed earlier in this section at each iteration of the loop. What about in the `map` case? For mapping over a numpy array, perhaps not, but what if mapping over a list? Without digging into how `map` works, it's hard to say, but it does seem that based on this example, `map` is not doing anything special that saves much time when mapping over the elements of a numpy array.

Here's an example where the bulk of time is in the actual computation and not in the looping itself. We'll run a bunch of regressions on a matrix `X` (i.e., each column of `X` is a predictor) using each column of the matrix `mat` to do a separate regression.

```
import time
import statsmodels.api as sm

n = 500000;
nr = 10000
nCalcs = int(n/nr)

mat = np.random.normal(size = (nr, nCalcs))

X = list(range(nr))
X = sm.add_constant(X)

def regrFun(i):
    model = sm.OLS(mat[:,i], X)
    return model.fit().params[1]

t0 = time.time()
out1 = list(map(regrFun, range(nCalcs)))
time.time() - t0
```

```
0.13381361961364746
```

```
t0 = time.time()
out2 = np.zeros(nCalcs)
for i in range(nCalcs):
    out2[i] = regrFun(i)
```

```
time.time() - t0
```

```
0.06110072135925293
```

Here they're about the same time (depending on when I render the document, I am getting some stochasticity in the timing). This is not too surprising – the overhead of the iteration should be small relative to the fundamental cost of the model fitting, and we wouldn't be concerned with the overhead of using a loop.

Matrix algebra efficiency

Often calculations that are not explicitly linear algebra calculations can be done as matrix algebra. If our Python installation has a fast (and possibly parallelized) BLAS, this allows our calculation to take advantage of it.

For example, we can sum the rows of a matrix by multiplying by a 1-d array of ones.

```
mat = np.random.normal(size=(500,500))

timeit.timeit('mat.dot(np.ones(500))', setup = 'import numpy as np',
              number = 1000, globals = {'mat': mat})
```

```
0.023054955003317446
```

```
timeit.timeit('np.sum(mat, axis = 1)', setup = 'import numpy as np',
              number = 1000, globals = {'mat': mat})
```

```
0.1186856430140324
```

Given the extra computation involved in actually multiplying each number by one, it's surprising that this is faster than numpy sum function. One thing we'd want to know is whether the BLAS matrix multiplication call is being done in parallel.

On the other hand, big matrix operations can be slow.

> Challenge
>
> Suppose you want a new matrix that computes the differences between successive columns of a matrix of arbitrary size. How would you do this as matrix algebra operations? It's possible to write it as multiplying the matrix by another matrix that contains 0s, 1s, and -1s in appropriate places. Here it turns out that the *for* loop is much faster than matrix multiplication. However, there is a way to do it faster as matrix direct subtraction.

Order of operations and efficiency

When doing matrix algebra, the order in which you do operations can be critical for efficiency. How should I order the following calculation?

```
n = 5000
A = np.random.normal(size=(n, n))
B = np.random.normal(size=(n, n))
x = np.random.normal(size=n)

t0 = time.time()
res1 = (A @ B) @ x
print(time.time() - t0)
```

```
1.9474890232086182
```

```
t0 = time.time()
res1 = A @ (B @ x)
print(time.time() - t0)
```

```
0.04534173011779785
```

> Challenge
>
> Why is the second order much faster?
> Count the number of multiplications involved in each of the two approaches.

Avoiding unnecessary operations

We can use the matrix direct product (i.e., `A*B`) to do some manipulations much more quickly than using matrix multiplication.

> Challenge
>
> How can I use the direct product to find the trace of a matrix, XY?

When working with diagonal matrices, you can generally get much faster results by being smart. The following operations: $X + D$, DX, XD are mathematically the sum of two matrices and products of two matrices. But we can do the computation without using two full matrices.

```
n = 1000
X = np.random.normal(size=(n, n))
diagvals = np.random.normal(size=n)
D = np.diag(diagvals)

# The following lines are very inefficient
summedMat = X + D
prodMat1 = D @ X
prodMat2 = X @ D
```

Challenge

Consider either $X + D$, DX, or XD. How can I use vectorization to do this much more quickly than the direct, naive translation of the math into code?

Exploit sparsity/structure!

More generally, sparse matrices and structured matrices (such as block diagonal matrices) can generally be worked with MUCH more efficiently than treating them as arbitrary matrices. The `scipy.sparse` package (for both structured and arbitrary sparse matrices) can help, as can specialized code available in other languages, such as C and Fortran packages.

Speed of lookup operations

There are lots of situations in which we need to retrieve values for subsequent computations. In some cases we might be retrieving elements of an array or looking up values in a dictionary.

Let's compare the speed of some different approaches to lookup.

```
n = 1000
x = list(np.random.normal(size = n))
labels = [str(v) for v in range(n)]
xD = dict(zip(labels, x))

timeit.timeit("x[500]", number = 10**6, globals = {'x':x})
```

```
0.02318129496416077
```

```
timeit.timeit("xD['500']", number=10**6, globals = {'xD':xD})
```

```
0.034863428969401866
```

How is it that Python can look up by key in the dictionary at essentially the same speed as jumping to an index position? It uses hashing, which allows O(1) lookup. In contrast, if one has to look through each label (key) in turn, that is O(n), which is much slower:

```
timeit.timeit("x[labels.index('500')]", number = 10**6, globals = {'x':x, 'labels': labels})
```

```
6.440078149025794
```

As a further point of contrast, if we look up elements by name in R in named vectors or lists, that is much slower than looking up by index, because R doesn't use hashing in that context and has to scan through the objects one by one until it finds the one with the name it is looking for. This stands in contrast to R and Python being able to directly go to the position of interest based on the index of an array, or to the hash-based lookup in a Python dictionary or an R environment.

Hashing (including name lookup)

Above I mentioned that Python uses hashing to store and lookup values by key in a dictionary. I'll briefly describe what hashing is here, because it is a commonly-used strategy in programming in general.

A hash function is a function that takes as input some data (some input of arbitrary length) and maps it to a fixed-length output that can be used as a shortened reference to the data. (The function should be deterministic, always returing the same output for a given input.) We've seen this in the context of git commits where each commit was labeled with a long base-16 number. This also comes up when verifying files on the Internet. You can compute the hash value on the file you get and check that it is the same as the hash value associated with the legitimate copy of the file. Even small changes in the file will result in a different hash value.

While there are various uses of hashing, for our purposes here, hashing can allow one to look up values by their name via a hash table. The idea is that you have a set of key-value pairs (sometimes called a dictionary) where the key is the name associated with the value and the value is some arbitrary object. You want to be able to quickly find the value/object quickly.

Hashing allows one to quickly determine an index associated with the key and therefore quickly find the relevant value based on the index. For example, one approach is to compute the hash as a function of the key and then take the remainder when dividing by the number of possible results (here the fact that the result is a fixed-length output is important) to get the index. Here's the procedure in pseudocode:

```
hash = hashfunc(key)
index = hash %% array_size
## %% is modulo operator - it gives the remainder
```

In general, there will be collisions – multiple keys will be assigned to the same index (this is unavoidable because of the fact that the hash function returns a fixed length output). However with a good hash function, usually there will be a small number of keys associated with a given bucket. So each bucket will contain a list of a small number of values and the associated keys. (The buckets might contain the actual values or they might contain the addresses of where the values are actually stored if the values are complicated objects.) Then determining the correct value (or the required address) within a given bucket is fast even with simple linear search through the items one by one. Put another way, the hash function distributes the keys amongst an array of buckets and allows one to look up the appropriate bucket quickly based on the computed index value. When the hash table is properly set up, the cost of looking up a value does not depend on the number of key-value pairs stored.

Python uses hashing to look up the value based on the key in a given dictionary, and similarly when looking up variables in namespaces. This allows Python to retrieve objects very quickly.

Additional general strategies for efficiency

It's also useful to be aware of some other strategies for improving efficiency.

Cache-aware programming

In addition to main memory (what we usually mean when we talk about RAM), computers also have memory caches, which are small amounts of fast memory that can be accessed very quickly by the processor. For example your computer might have L1, L2, and L3 caches, with L1 the smallest and fastest and L3 the largest and slowest. The idea is to try to have the data that is most used by the processor in the cache.

If the next piece of data needed for computation is available in the cache, this is a *cache hit* and the data can be accessed very quickly. However, if the data is not available in the cache, this is a *cache miss* and the speed of access will be a lot slower. *Cache-aware programming* involves writing your code to minimize cache misses. Generally when data is read from memory it will be read in chunks, so values that are contiguous will be read together.

How does this inform one's programming? For example, if you have a matrix of values stored in row-major order, computing on a row will be a lot faster than computing on a column, because the row can be read into the cache from main memory and then accessed in the cache. In contrast, if the matrix is large and therefore won't fit in the cache, when you access the values of a column, you'll have to go to main memory repeatedly to get the values for the column because the values in the column are not stored contiguously.

There's a nice example of the importance of the cache at the bottom of this blog post.

If you know the size of the cache, you can try to design your code so that in a given part of your code you access data structures that will fit in the cache. This sort of thing is generally more relevant if you're coding in a language like C. But it can matter sometimes in interpreted languages too.

Let's see what happens in Python. By default, matrices in numpy are row-major, also called "C order". I'll create a long matrix with a small number of very long columns and a wide matrix with a small number of very long rows.

```
nr = 800000
nc = 100

A = np.random.normal(size=(nr, nc))   # long matrix
tA = np.random.normal(size=(nc, nr))  # wide matrix

## Verify that A is row-major using `.flags` (notice the `C_CONTIGUOUS: True`).
A.flags
```

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

Note that I didn't use `A.T` or `np.transpose` as that doesn't make a copy in memory and so the transposed matrix doesn't end up being row-major. You can use `A.flags` and A.T.flags‘ to see this.

Now let's time calculating the sum by column in the long matrix vs. the sum by row in the wide matrix.

Exactly the same number of arithmetic operations needs to be done in an equivalent manner for the two cases. We want to use a large enough matrix so the entire matrix doesn't fit in the cache, but not so large that the example takes a long time or a huge amount of memory. We'll use a rectangular matrix, such that the summation for a single column of the long matrix or a single row of the wide matrix involves many numbers, but there are a limited number of such summations. This focuses the example on the efficiency of the column-wise vs. row-wise summation rather than any issues that might be involved in managing large numbers of such summations (e.g., doing many, many summations that involve just a few numbers).

```
# Define the sum calculations as functions
def sum_by_row():
    return np.sum(tA, axis=1)

def sum_by_column():
    return np.sum(A, axis=0)

timeit.timeit(sum_by_row, number=10)
```

```
0.40739037399180233
```

```
timeit.timeit(sum_by_column, number=10)  # potentially slow
```

```
0.47676839400082827
```

Suppose we instead do the looping manually.

```
timeit.timeit('[np.sum(A[:,col]) for col in range(A.shape[1])]',
    setup = 'import numpy as np', number=10, globals = {'A': A})
```

```
5.483329343027435
```

```
timeit.timeit('[np.sum(tA[row,:]) for row in range(tA.shape[0])]',
    setup = 'import numpy as np', number=10, globals = {'tA': tA})
```

```
0.42822560301283374
```

Indeed, the row-wise calculations are much faster when done manually. However, when done with the `axis` argument in `np.sum` there is little difference. So that suggests numpy might be doing something clever in its implementation of `sum` with the `axis` argument.

> Challenge
>
> Suppose you were writing code for this kind of use case. How could you set up your calculations to do either row-wise or column-wise operations in a way that processes each number sequentially based on the order in which the numbers are stored? For example suppose the values are stored row-major but you want the column sums.

When we define a numpy array, we can choose to use column-major order (i.e., "Fortran" order) with the `order` argument.

Loop fusion

Let's consider this (vectorized) code:

```
x = np.exp(x) + 3*np.sin(x)
```

This code has some downsides.

- Think about whether any additional memory has to be allocated.
- Think about how many for loops will have to get executed.

Contrast that to running directly as a for loop (e.g., here in Julia or in C/C++):

```
for i in 1:length(x)
    x[i] = exp(x[i]) + 3*sin(x[i])
end
```

How does that affect the downsides mentioned above?

Combining loops is called 'fusing' and is an important optimization that Julia can do, as shown in this demo. It's also a key optimization done by XLA, a compiler used with JAX and Tensorflow, so one approach to getting loop fusion in Python is to use JAX (or Tensorflow) for such calculations within Python rather than simply using numpy.

JIT compilation (with JAX)

You can think of JAX as a version of numpy enabled to use the GPU (or automatically parallelize on CPU threads) and provide automatic differentiation. For its core operations, JAX defines versions of the operations (as compiled code) for the CPU and for the GPU.

We can also JIT compile JAX code. Behind the scenes, the instructions are compiled to machine code for different backends (e.g., CPU and GPU) using the XLA compiler.

Let's first consider running a vectorized calculation using JAX on the CPU, which will use multiple threads (discussed in Unit 6), each thread running on a separate CPU core on our computer. For now all we need to know is that the calculation will run in parallel, so we expect it to be faster than when using numpy, which won't run the calculation in parallel.

> **Warning**
>
> This demo uses 4 GB of memory just in the process of creating `x`. Run on your own machine with caution.

```
import time
import numpy as np
import jax.numpy as jnp

def myfun_np(x):
    y = np.exp(x) + 3 * np.sin(x)
    return y
```

```
def myfun_jnp(x):
    y = jnp.exp(x) + 3 * jnp.sin(x)
    return y

n = 500000000

x = np.random.normal(size = n).astype(np.float32)  # 32-bit for consistency with JAX default
x_jax = jnp.array(x)  # 32-bit by default
print(x_jax.platform())
```

```
cpu
```

```
t0 = time.time()
z = myfun_np(x)
t1 = time.time() - t0

t0 = time.time()
z_jax = myfun_jnp(x_jax).block_until_ready()
t2 = time.time() - t0

print(f"numpy time: {round(t1,3)}\njax time: {round(t2,3)}")
```

Running on the SCF gandalf machine (not shown above), we get these times.

```
numpy time: 17.181
jax time: 5.722
```

There's a nice speedup compared to numpy.

Since JAX will often execute computations asynchronously (in particular when using the GPU), the `block_until_ready` invocation ensures that the computation finishes before we stop timing.

By default the JAX floating point type is 32-bit so we forced the use of 32-bit numbers for numpy for comparability. One could have JAX use 64-bit numbers like this:

```
import jax
jax.config.update("jax_enable_x64", True)
```

Next let's consider JIT compiling it, which should fuse the vectorized operations and avoid temporary objects. The JAX docs have a nice discussion of when JIT compilation will be beneficial.

```
import jax
myfun_jnp_jit = jax.jit(myfun_jnp)

t0 = time.time()
z_jax_jit = myfun_jnp_jit(x_jax).block_until_ready()
t3 = time.time() - t0
print(f"jitted jax time: {round(t3,3)}")
```

```
jitted jax time: 3.218
```

So that gives a nice two-fold additional speedup.

We could also have used the `jax.jit` decorator rather than directly calling `jax.jit`:

```
@jax.jit
def myfun_jnp(x):
    y = jnp.exp(x) + 3 * jnp.sin(x)
    return y
```

Lazy evaluation

What's strange about this R code?

```
f <- function(x) print("hi")
system.time(mean(rnorm(1000000)))
```

```
   user  system elapsed
  0.058   0.004   0.061
```

```
system.time(f(3))
```

```
[1] "hi"
```

```
   user  system elapsed
      0       0       0
```

```
system.time(f(mean(rnorm(1000000))))
```

```
[1] "hi"
```

```
   user  system elapsed
  0.001   0.000   0.001
```

It seems like the `rnorm(1000000)` is not actually executed when being passed to `f`. That is "lazy evaluation" in action - the code that is passed in as an argument is only evaluated when it is needed (and in this case it's not needed for the function to run).

Lazy evaluation is not just an R thing. It also occurs in Tensorflow (particularly version 1), the Python Dask package, and in Spark. The basic idea is to delay executation until it's really needed, with the goal that if one does so, the system may be able to better optimize a series of multiple steps as a joint operation relative to executing them one by one.

However, Python itself does not have lazy evaluation.