Programming concepts

Chris Paciorek

2022-09-06

Table of contents

PDF

References:

- TBD

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.

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 these 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, and special characters, rather than to the character class of R objects. The string or strings will generally be stored as an R character vector.

String processing and regular expressions in Python

For details of string processing in Python, including use of regular expressions, see the string processing tutorial. (You can ignore the sections on R.) That tutorial then refers to the bash shell tutorial for details on regular expressions.

In class we'll work through some problems in the string processing tutorial, focusing in particular on the use of 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.

Regex/string processing challenges

We'll work on these challenges (and perhaps one or two others) in class in the process of working through the string processing tutorial.

1. What regex would I use to find any number with or without a decimal place.
2. 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 search and replace operation? (Alternatively, how could I do this without using regular expressions at all?)

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 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`).

Here are some examples of using special characters.

> **Note** It is hard to compile the qmd file correctly for these code chunks, so I am just pasting in the output from running in Python 'manually' in some cases.)

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

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

```
tmp = "Harry said, \"Hi\".\n"
print(tmp)      # prints out with the newline
```

```
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.

```
## Search for characters that are not 'z'
## (using ^ as regular expression syntax)
[print(st + ":\t", re.search("[^z]", st))
    for st in ["a^2", "93", "zzz", "zit", "azar"]]

## Search for either a '^' (as a regular character) or a 'z':
```

```
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'>
[None, None, None, None, None]
```

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

## Search for exactly three characters
## (using . as regular expression syntax)
```

```
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'>
[None, None, None, None, None]
```

```
[print(st + ":\t", re.search("^.{3}$", st))
    for st in ["abc", "1234", "def"]]

## Search for a period (as a regular character)
```

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

```
[print(st + ":\t", re.search("\.", st)) for st in ["3.9", "27", "4.2"]]
```

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

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.
```

Advanced note: Searching for an actual backslash gets even more complicated, 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!

```
## Search for an actual backslash
tmp = "something \ other\n"
print(tmp)  # notice the escaping of the literal backslash
```

```
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
```

Warning 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 R cannot interpret as a quote because it's some other ASCII quote character. If you paste in a ” from PDF, it will not be interpreted as a standard R 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

## Or without regular expressions
grep -F "\" file.txt
```

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 `os.()`, as follows, noting that we can save the result of a system call to an R 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 2586\ndrwxr-sr-x 14 paciorek scfstaff      67 Aug 17 16:32 .\ndrwxr-sr-x 11 paciorek scf
```

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

```
[b'drwxr-sr-x 11 paciorek scfstaff      34 Aug 17 16:01 ..\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")
```

```
['coop.txt.gz', 'test.db', 'cpds.csv', 'IPs.RData', 'airline.csv', 'stackoverflow-2016.db', 'hiv
```

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

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

```
['coop.txt.gz', 'test.db', 'cpds.csv', 'IPs.RData', 'airline.csv', 'stackoverflow-2016.db', 'hiv
```

  It's best if you can to write your code in a way that is *agnostic* to the underlying 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='5.15.0-73-generic', version='#8
```

```
platform.python_version()
```

```
'3.11.0'
```

- To retrieve environment variables:

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

```
'/system/linux/mambaforge-3.11/bin:/usr/local/linux/mambaforge-3.11/condabin:/system/linux/mamba
```

- 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
```

Controlling the behavior of Python

- 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.

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.

```
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(7)
```

```
The arg is: 7.
```

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 local scope (i.e., the local context). The mapping of names (references) to objects is called a *namespace.* We discuss scopes and namespaces in more detail later.

```
try:           # mymod not in scope
    mymod.x
except Exception as error:
    print(error)

import mymod
mymod         # essentially a dictionary in the local scope
x             # not a name in the local scope
range         # a builtin, not from the module
mymod.x
dir(mymod)
mymod.x
mymod.range
```

So `range` and `mymod` are in the global scope and `range` and `x` are in the module scope of `mymod`.

Note the usefulness of distinguishing the objects in a module from those in the global scope. 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, at which point it is distinct from the object in the module:

```
from mymod import x
x
```

```
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 a set of 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
from mymod import *

print("Welcome to my package.")
EOF

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

def myfun(val):
    print("The arg is: ", str(val), ".", sep = '')
EOF
```

```
import mypkg
```

```
Welcome to my package.
```

```
mypkg.x
```

```
7
```

```
mypkg.myfun(7)
```

```
The arg is: 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.

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.mysubmod`.

```
mkdir mypkg/mysubmod

cat << EOF > mypkg/mysubmod/__init__.py
print("Welcome to my package's submodule.")
EOF

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

```
import mypkg.mysubmod.values  ## Note that __init__.py is invoked
```

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

```
mypkg.mysubmod.values.b
```

```
7
```

Note that a given `__init__.py` is invoked when importing anything nested within the directory containing the `__init__.py`.

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.

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](.

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 > environment.yml
conda env create -f environment.yml
```

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=3.11`) the Python 3.11 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=3.11
source activate myenv
conda install numpy
```

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. If you run `pkgname.__file__`, you'll be able to see where the imported package is installed.

`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 = 7
x*3
```

```
21
```

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 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'>
```

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?
```

```
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'
```

```
myArray = [1, 3, 5, 9, 4, 7]
# myArray[2.0]    # What do you think is going to happen?
# myArray[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] 1.016687
```

```
x[2.73]
```

```
[1] 1.016687
```

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.

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 a notion of "next", while `iterator` objects have a notion of "stopping".

If you implement your own class that falls into one of these categories, it should follow the relevant protocol by providing the required methods. For example container class that supports iteration should provide the `__iter__` method, which produces an iterator object. An iterator object should provide `__iter__` and `__next__`.

Here we see that `tuple`s are iterable containers:

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

print(next(myit))
```

```
apple
```

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

```
banana
```

```
myit.__next__()
```

```
'cherry'
```

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

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

```
next(x)
```

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

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

```
r = range(5)
r
```

```
range(0, 5)
```

```
list(r)
```

```
[0, 1, 2, 3, 4]
```

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.

```
import numpy as np
x = np.array([1.2, 3.5, 4.2])
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
```

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).

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.

```
import numpy as np

class tsSimClass:
    ## class for holding time series simulators
    def __init__(self, times, mean = 0, corParam = 1):
        ## add assertions that corParam is numeric, length 1 and times is np array
        self._times = times
        self.n = len(times)
        self.mean = mean
        self.corParam = corParam
        self._currentU = False
        self._calcMats()
    def __str__(self):    # 'print' method
        return(f"An object of class `tsSimClass` with {self.n} time points.")
    def __len__(self):
        return(self.n)
    def setTimes(self, newTimes):
        self._times = newTimes
        self._calcMats()
    def getTimes(self):
        return(self._times)
    def simulate(self):
        if not self._currentU:
            self._calcMats()
        ## 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)))
    def _calcMats(self):
        ## calculates correlation matrix and Cholesky factor
        lagMat = np.abs(self._times[:, np.newaxis] - self._times)
        corMat = np.exp(-lagMat ** 2 / self.corParam ** 2)
        self.U = np.linalg.cholesky(corMat)
        print("Done updating correlation matrix and Cholesky factor.")
        self._currentU = True
```

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

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

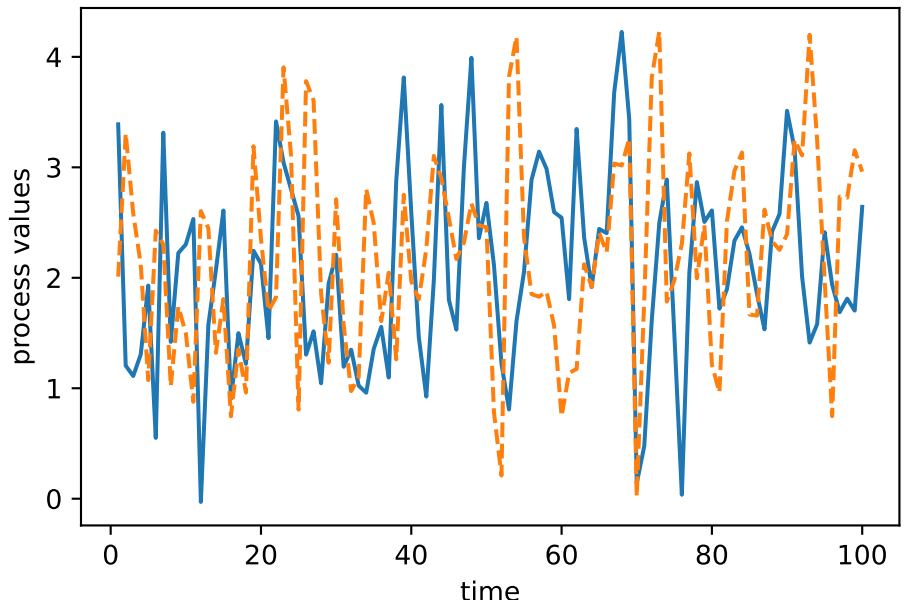
```
Done updating correlation matrix and Cholesky factor.
```

```
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.getTimes(), y1, '-')
plt.xlabel('time')
plt.ylabel('process values')
## simulate a second series
y2 = myts.simulate()
plt.plot(myts.getTimes(), y2, '--')
plt.show()
```



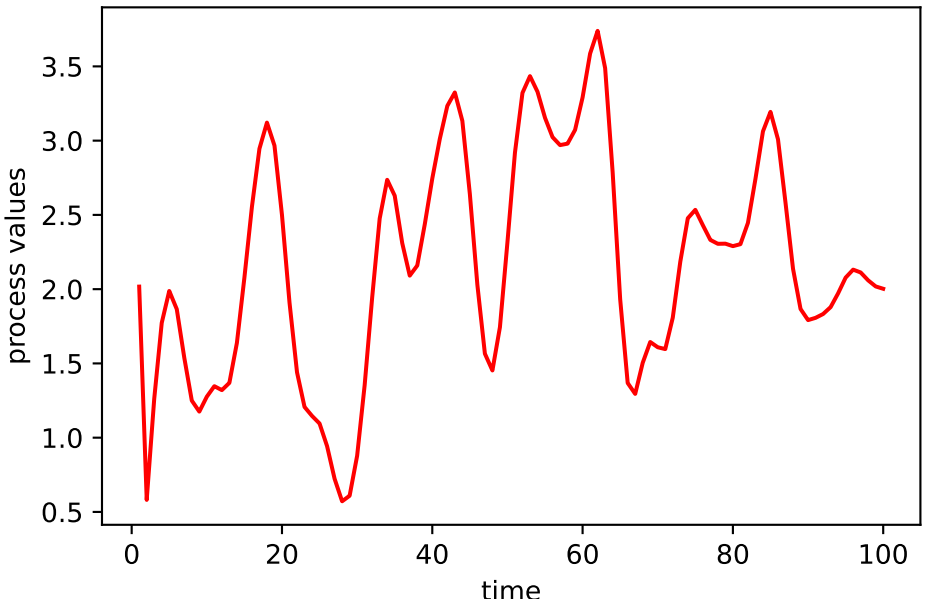
We could set up a different object that has different parameter values. That new simulated time series is less wiggly because the `corParam` value is larger than before.

```
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 (corParam)
y3 = myts2.simulate()

plt.plot(myts2.getTimes(), 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 `mytsRef` 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 `mytsRef` was done.

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

## Now let's change the values of a field
```

```
myts.setTimes(np.arange(1,1001,10))
```

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

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

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

```
mytsRef.getTimes()[0:4] # the same as `myts`
```

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

```
mytsFullCopy.getTimes()[0:4] # different from `myts`
```

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

In contrast `mytsFullCopy` is a reference to a different object, and all the data from `myts` had to be copied over to `mytsFullCopy`. 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 `mytsRef`. 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 vector 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 `setTimes`, which correctly keeps all the fields in the object internally consistent (by calling `_calcMats`). 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.

Challenge

Challenge

How would you get Python to quit immediately, without asking for any more information, when you simply type `q` (no parentheses!) instead of `quit()`? There are actually a couple ways to do this. (Hint: you can do this by understanding what happens when you type `q` and how to exploit the characteristics of Python classes.)

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.

```
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.

More relevant examples of inheritance in Python and R include how regression models are handled. E.g., in Python's `statsmodels`, the `OLS` class inherits from the `WLS` 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
```

```
smoke = Bear("Smoky the Bear", 77)
smoke.count
```

```
2
```

Adding attributes

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

```
yog.bizarre = 7
yog.bizarre
```

```
7
```

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

foo.bizarre = 3
foo.bizarre
```

```
3
```

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 two 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.

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 as well as a `__repr__` method giving what to display when the name of an object is typed.

```
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 0x7fce31a4e5d0>
```

```
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})")

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

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

```
yog          # Invokes __repr__
```

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

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.

Let's see an example of defining a dunder method for the `Bear` 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 __add__(self, value):
          self.age += value

yog = Bear("Yogi the Bear", 23)
yog + 12
print(yog)
```

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

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'>
```

```
type(Bear)
```

```
<class 'type'>
```

7. Functional programming

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.

Functional programming in Python and R

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])
```

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()`.

Challenge What are some other R functions that are called for the purpose of the side effects they produce? (For example, which functions change the state of your R session in some way?)

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(3)  # x is not a function (yet)
except Exception as error:
    print(error)
```

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

```
def x(val):
    return pow(val, 2)

x(3)
```

```
9
```

```
type(x)
```

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

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

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

```
2.0
```

We can also pass a function into another function as the actual function object. This is an important aspect of functional programming.

```
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

Operators are examples of generic function OOP, where the appropriate method of the class of the first object that is part of the operation 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.

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]
```

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 x: x * 2, x))
```

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? [**Fernando, could you check that this looks ok?**]

- 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 local scope. Any look-up of variables not found in the local scope (using the namespace that was created) is done using 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()

def function_b():
    function_c()

def function_c():
    traceback.print_stack()

function_a()
```

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

Function inputs and outputs

Arguments

Arguments can be specified by position (based on the order of the inputs) or by name, using `name=value`, with positional arguments appearing first. first. You can see the arguments and defaults for a function using the help system.

All positional arguments are required.

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


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'
```

Here's another error related to positional vs. keyword arguments.

```
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'
```

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)
y
```

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

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

```
def myfun(scalar_x, list_x, list_x_reassigned, nparray_x,
    nparray_x_newid, nparray_x_copy):
    scalar_x = 99
    list_x[0] = 99
    list_x_reassigned = [99,2,3]
    nparray_x[0] = 99
    newx = nparray_x_newid
    newx[0] = 99
    xcopy = nparray_x_copy.copy()
    xcopy[0] = 99


scalar_x = 1
list_x = [1,2,3]
list_x_reassigned = np.array([1,2,3])
nparray_x = np.array([1,2,3])
nparray_x_newid = np.array([1,2,3])
nparray_x_copy = np.array([1,2,3])


myfun(scalar_x, list_x, list_x_reassigned, nparray_x,
    nparray_x_newid, nparray_x_copy)
```

Here are the cases where state is preserved:

```
scalar_x
```

```
1
```

```
list_x_reassigned
```

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

```
nparray_x_copy
```

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

And here are the cases where state is modified:

```
list_x
```

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

```
nparray_x
```

```
array([99,  2,  3])
```

```
nparray_x_newid
```

```
array([99,  2,  3])
```

Basically if you replace the reference (object name) then the state outside the function is preserved. If you modify part of the object, state is not preserved.

Pointers

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
```

- The `int*` declares `ptr` to be a pointer to (the address of) the integer `x`.
- The `&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`.

Vectors 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 vector. 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.mysubmod.values 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.mysubmod.values
mypkg.mysubmod.values.x
```

```
999
```

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

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

```
['Bear', 'GrizzlyBear', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__pa
```

```
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 namespaces are searches:

- **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 in the 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), 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 `scope.py`:

```
cat test.py
```

```
magic_number = 7

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

Now suppose we also define `magic_number` in the local scope in which `myfun` is called from.

```
import test
magic_number = 900
test.myfun(3)
```

```
21
```

We see that Python uses `magic_number` from the What would be bad about using `magic_number` from the global scope of the Python session rather than the global scope of the module? Consider a case where instead of using the `scope.py` module we were using code from a package.

Lexical scoping (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)
```

Variables in the enclosing scope are available within a function. **The enclosing scope is the scope in which a function is defined, not the scope from which a function is called.**

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

The behavior of basing lookup 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:

```
## 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()
```

```
f() # what will happen?

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

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)
```

```
13
```

Let's work through this:

1. What is the enclosing scope for the function `g()`?
2. What does `g()` use for `y`?
3. When `fun_constructor()` finishes, does its namespace disappear? What would happen if it did?
4. What is the enclosing function (scope) for `myfun()`?

[Fernando: is there a way to access variables in a closure (i.e., in this case access 'y' based on `myfun`). Is there a way to find the enclosing scope for 'myfun' and look in it?]

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 and of enclosing scopes.

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  # Modifying 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 new function(s) is created and returned and one can then call that new function(s). The new function then can access objects in the enclosing scope (the scope of the original function) 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(input):
    data = input
    def g(param):
            return(param * data)
    return(g)

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

```
array([ 0.5081473 ,  2.22166935, -2.86110181, -0.79865552,  0.09784364])
```

```
scaler(6)
```

```
array([ 1.0162946 ,  4.44333871, -5.72220361, -1.59731104,  0.19568728])
```

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 see 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/dat
data = iris['SepalLength']
```

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


  bootmeans()
```

```
array([5.874     , 5.95533333, 5.86      , 5.754     , 5.77466667,
       5.81733333, 5.902     , 5.79933333, 5.842     , 5.81933333,
       5.854     , 5.97      , 5.84      , 5.80133333, 5.99333333,
       5.88133333, 5.83133333, 5.84533333, 5.91466667, 5.79666667])
```

```
  bootmeans.__closure__
```

```
(<cell at 0x7fce2b7262f0: int object at 0x7fce410dc968>, <cell at 0x7fce2b7266b0: numpy.ndarray objec
```

8. Memory and copies

Overview

The main things to remember when thinking about memory use are: (1) numeric vectors 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.

```
  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.

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` (the `-h` is for 'human-readable', i.e. show in GB (G)) on Linux machine.

```
       total used free shared buff/cache available
Mem:   251G 998M 221G   2.6G        29G      247G
Swap:  7.6G 210M 7.4G
```

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 only about 1 GB is in use (indicated in the `used` column).

`top` (Linux or Mac) and `vmstat` (on Linux) both show overall memory use, but remember that the amount actually available to you is the amount free plus any buff/cache usage. Here is some example output from `vmstat`:

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
r b   swpd      free   buff    cache si so bi bo in cs us sy id wa st
1 0 215140 231655120 677944 30660296  0  0  1  2  0  0 18  0 82  0  0
```

It shows 232 GB free and 31 GB used for cache and therefore available, for a total of 263 GB available.

Here are some example lines from `top`:

```
KiB Mem : 26413715+total, 23180236+free, 999704 used, 31335072 buff/cache
KiB Swap:  7999484 total,  7784336 free, 215148 used. 25953483+avail Mem
```

We see that this machine has 264 GB RAM (the total column in the `Mem` row), with 259.5 GB available (232 GB free plus 31 GB buff/cache as seen in the `Mem` row). (I realize the numbers don't quite add up for reasons I don't fully understand, but we probably don't need to worry about that degree of exactness.) Only 1 GB 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 never want your machine to be using swap for memory because your jobs will slow to a crawl. As seen above, the `swap` line in both `free` and `top` shows 8 GB swap space, with very little in use, as desired.

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.")

# Let's turn that into a function for later use:
```

```
Memory usage: 485.982208  Mb.
```

```
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)
```

```
80000201
```

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)
```

```
140523295107248
```

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

```
80000112
```

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

```
140523295107248
```

```
x is y
```

```
True
```

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

```
80000112
```

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

```
140523295107632
```

```
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)
```

```
140523295475824
```

```
id(obj)
```

```
140523293501504
```

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

```
140523295475824
```

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

```
140523295475824
```

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

```
140523295485520
```

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

```
140523295748480
```

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

```
140523293503920
```

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

```
True
```

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

```
False
```

What do we notice?

- The list itself appears to be a vector of 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).

How character strings are stored.

Similar tricks are used for storing character vectors. We'll explore this in a problem on PS4.

Modifying elements in place

What do these simple experiments tell us?

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

```
140523295485616
```

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

```
140523295485616
```

```
y = x

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

```
140523295475728
```

```
id(y)
```

```
140523295485616
```

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

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.

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

```
Memory usage: 1446.703104  Mb.
```

```
y = x
mem_used()
```

```
Memory usage: 1446.703104  Mb.
```

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

```
Memory usage: 2246.7584  Mb.
```

```
del y
mem_used()
```

```
Memory usage: 1446.858752  Mb.
```

Only if we re-assign to `x` 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.

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

```
Memory usage: 1446.87104  Mb.
```

```
del x
mem_used()
```

```
Memory usage: 1446.87104  Mb.
```

```
del y
mem_used()
```

```
Memory usage: 646.873088  Mb.
```

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)  # 2
```

```
2
```

```
y = x
sys.getrefcount(x)  # 3
```

```
3
```

```
sys.getrefcount(y)  # 3
```

```
3
```

```
del y
sys.getrefcount(x)  # 2
```

```
2
```

```
y = x
x = np.random.normal(size = 5)
sys.getrefcount(y)  # 2
```

```
2
```

```
sys.getrefcount(x)  # 2
```

```
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 couple 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.

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
```

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, when copies are made, and lazy evaluation. This code (translated from the original R code) is courtesy of Yuval Benjamini. For our purposes here, let's assume that `xvar` and `yvar` are very long vectors using a lot of memory.

```
def fastcount(xvar, yvar):
    naline = np.isnan(xvar)
    naline[np.isnan(yvar)] = True
    localx = xvar
    localy = yvar
    localx[naline] = 0
    localy[naline] = 0
    useline = ~naline
    ## We'll ignore the rest of the code.
    ## ....
```

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
abs(x)
x = 'hi'
try:
    abs(x)
except Exception as error:
    print(error)
x*3
```

Because of dynamic typing, when the interpreter sees `abs(x)` it needs to check if `x` is something to on the absolute value function be used, 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:

```
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 a vector 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.

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
```

```
0.7152149677276611
```

```
t0 = time.time()
out = np.exp(x)
time.time() - t0
```

```
0.011996269226074219
```

```
import py_compile
py_compile.compile('vec.py')
```

```
'__pycache__/vec.cpython-311.pyc'
```

```
cp __pycache__/vec.cpython-311.pyc vec.pyc
rm vec.py    # make sure non-compiled module not loaded
```

```
import vec
vec.__file__
```

```
'/accounts/vis/paciorek/teaching/243fall23/stat243-fall-2023/units/vec.pyc'
```

```
t0 = time.time()
out = vec.f(x)
time.time() - t0
```

```
0.7163336277008057
```

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 very little.

Timing your code

There are a few ways to time code:

```
import time
t0 = time.time()
x = 3
t1 = time.time()

print(f"Execution time: {t1-t0} seconds.")
```

```
Execution time: 0.004244089126586914 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.

Using `time` is fine for code that takes a little while to run, but for code that is really fast, 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)
```

```
8.278153836727142e-05
```

```
code = '''
x = np.exp(3.)
'''

timeit.timeit(code, setup = 'import numpy as np', number = 100)
```

```
6.929691880941391e-05
```

That reports the **total** time for the 100 replications.

We can run it from the command line.

```
python -m timeit -s 'import numpy' -n 1000 'x = numpy.exp(3.)'
```

```
1000 loops, best of 5: 655 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.

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
y = np.random.normal(size = 5000)
x = np.random.normal(size = (5000,1000))

import cProfile

cProfile.run('lr_slow(y,x)')
```

The `cumtime` column includes the time spent in subfunction calls 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)

cProfile.run('lr_fast(y,x)')
```

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.

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

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
```

```
0.07516932487487793
```

```
t0 = time.time()
out2 = fun_append(z)
time.time() - t0
```

```
0.07385683059692383
```

```
t0 = time.time()
out3 = fun_append_np(z)
time.time() - t0
```

```
2.4546351432800293
```

So what's going on? First let's consider what is happening with the use of `np.append`. What must be happening in terms of memory use and copying when we append an element?

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…

```
def fun_append2(vals):
    n = len(vals)
    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}")
    return(x)

z = np.random.normal(size = 10000)
out = fun_append2(z)
```

```
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
iteration 1100: size 10008
iteration 1244: size 11288
iteration 1404: size 12728
iteration 1584: size 14360
iteration 1788: size 16184
iteration 2016: size 18232
iteration 2272: size 20536
iteration 2560: size 23128
iteration 2884: size 26040
iteration 3248: size 29336
iteration 3660: size 33048
iteration 4124: size 37208
iteration 4644: size 41880
iteration 5228: size 47160
iteration 5888: size 53080
iteration 6628: size 59736
iteration 7460: size 67224
iteration 8396: size 75672
iteration 9452: size 85176
```

Strangely the id of `x` doesn't seem to change, even though we are allocating new memory. Perhaps there is something more complicated than I would expect going on in terms of how the list stores pointers to the underlying elements.

[Fernando, do you know why the id doesn't change? There must be a copy being made as the list gets longer.]

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)
```

```
0.012318611145019531
```

```
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)
```

```
0.8638944625854492
```

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.

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:
cat /usr/local/linux/mambaforge-3.11/lib/python3.11/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})
```

```
1.8504798538051546
```

```
## Vectorized slice replacement
timeit.timeit('truncx = x; truncx[x < 0] = 0', number = 10, globals = {'x':x})
```

```
0.01011103531345725
```

```
## Vectorized math trick
timeit.timeit('truncx = x * x>0', number = 10, globals = {'x':x})
```

```
0.018781783059239388
```

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 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 Python's row-major ordering (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 Section 4.6 on the cache), such that it may not matter for numpy calculations.
- You can use direct arithmetic operations to add/subtract/multiply/divide a vector by each column of a matrix, e.g. `A*b` does element-wise multiplication of each column of A by a vector b. If you need to operate by row, 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 vector 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.

Are map operations faster than loops?

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. If you're worried about speed, it's a good idea to benchmark `map` or `pandas.apply` against looping.

Here's an example where the bulk of time is in the actual computation and not in the looping itself. Here `map` is not faster than a loop.

```
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
```

```
0.18105006217956543
```

```
t0 = time.time()
out2 = np.zeros(nCalcs)
for i in range(nCalcs):
    out2[i] = regrFun(i)


time.time() - t0
```

```
0.058990478515625
```

And here's an example, where (unlike the previous example) 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
```

```
0.012933015823364258
```

```
t0 = time.time()
vals = np.zeros(n)
for i in range(n):
    vals[i] = np.exp(x[i])


time.time() - t0
```

```
0.7939445972442627
```

```
t0 = time.time()
vals = [np.exp(v) for v in x]
time.time() - t0
```

```
0.6351616382598877
```

```
t0 = time.time()
vals = list(map(np.exp, x))
time.time() - t0
```

```
0.5969336032867432
```

In fact, 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).

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 vector of ones.

```
mat = np.random.normal(size=(500,500))

timeit.timeit('mat.dot(np.ones(500))', setup = 'import numpy as np', number = 1000, globals = {'ma
```

```
0.02382083795964718
```

```
timeit.timeit('np.sum(mat, axis = 1)', setup = 'import numpy as np', number = 1000, globals = {'ma
```

```
0.12136216554790735
```

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)
```

```
1.7429442405700684
```

```
t0 = time.time()
res1 = A @ (B @ x)
print(time.time() - t0)
```

```
0.030786752700805664
```

Why is the second order much faster?

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?

Finally, 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. **Challenge**: How?

```
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
```

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))
keys = [str(v) for v in range(n)]
xD = dict(zip(keys, x))

timeit.timeit("x[500]", number = 10**6, globals = {'x':x})
```

```
0.017121989279985428
```

```
timeit.timeit("xD['500']", number=10**6, globals = {'xD':xD})
```

```
0.027082541957497597
```

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 key in turn, that is O(n), which is much slower:

```
timeit.timeit("x[keys.index('500')]", number = 10**6, globals = {'x':x, 'keys':keys})
```

```
5.7154008792713284
```

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.

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 row because the values 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.

However with numpy, numpy is smart enough in some cases to figure out how to rearrange the looping in concordance with the calculation being done and how the data are laid out in memory.

```
nr = 800000
nc = 100

A = np.random.normal(size=(nr, nc))
tA = A.T

# Define the mean calculations as functions
def mean_by_column():
    return np.mean(A, axis=0)

def mean_by_row():
    return np.mean(tA, axis=1)

timeit.timeit(mean_by_column, number=10)
```

```
0.5042988602072
```

```
timeit.timeit(mean_by_row, number=10)
```

```
0.5089347762987018
```

The analogous code in R would show a substantial difference, so in this case numpy is more sophisticated than R.

Suppose we instead do the looping manually to see if it matters if we don't let numpy do any tricks with the looping:

```
timeit.timeit('[np.mean(A[:,col]) for col in range(A.shape[1])]',
    setup = 'import numpy as np', number=10, globals = {'A': A})   # 5.10 sec.
```

```
5.671365459449589
```

```
timeit.timeit('[np.mean(tA[row,:]) for row in range(tA.shape[0])]',
    setup = 'import numpy as np', number=10, globals = {'tA': tA})  # 5.06 sec.
```

```
5.837595785036683
```

Strangely, the row-wise and column-wise operations take the same amount of time. I don't know why that is. [**Fernando - do you have any ideas?**]

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 Tensorflow, so one approach to getting loop fusion in Python is to use Tensorflow for such calculations within Python rather than simply using numpy.

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 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.

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. 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.

Lazy evaluation

What's strange about this R code?

```
f <- function(x) print("hi")
system.time(mean(rnorm(1000000)))
```

```
   user  system elapsed
  0.061   0.000   0.062
```

```
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
```

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.