Jonathan Goodman, February, 2024

Python For Smarties

1 Introduction

These notes are for people who want to know what they're doing when they code in Python. The idea is not to do the "for dummies" cookbook approach, but to explain how Python works to people who want to know. If you understand some basic principles of the Python language, you can write and read code reliably. It's often better, in the long run, that guessing at code from examples that don't quite do what you want.

My experience with math grad students who use Python is that they know specific facts but not the basic principles of Python. Many are unaware of the basic differences between Python and other interpreted languages such as R, Matlab, and Julia. These misunderstandings lead to hard-to-find bugs.

Python has many features that go unmentioned here. I want to describe the principles, not specific details that you get better from reading the documentation.

There are many online Python resources. I recommend, as much as possible, using only the official Python documentation and StackOverflow.

2 Environment and the Python Command Line

Python is a *scripting language*, which means that it *interprets* then *executes* individual *commands* one at a time. Each command is executed in the *environment* that the interpreter has at that time. Executing the command can change the environment.

The *command line* is a simple way to interact with the Python interpreter. In a terminal window.[1] On an Apple or Linux computer, you can open a terminal window and type the command that opens the command line interpreter, as shown in Figure 1. A Python *command* is a Python expression that the interpreter can *evaluate*. I type the command `2*(4+5)` then "return". The Python interpreter "figures out" that the value is `18`, which it prints. After printing `18`, the Python interpreter gives me another command prompt.

The *environment* is, roughly speaking, a collection of *names* that are *bound* to *objects*. A *name* is just a character string, such as `x` or `output_file`. An *object* is a piece of data or a collection of data. It could be as simple as a single number or something more complex, such as the information needed to access a computer file. A *binding* associated to a name, *pointer* is a term more familiar to C programmers, "points to" the object the name is bound to. For example,

[1] Instructions below suggest various ways to open and enter a terminal window and launch the Python command line interpreter in it.

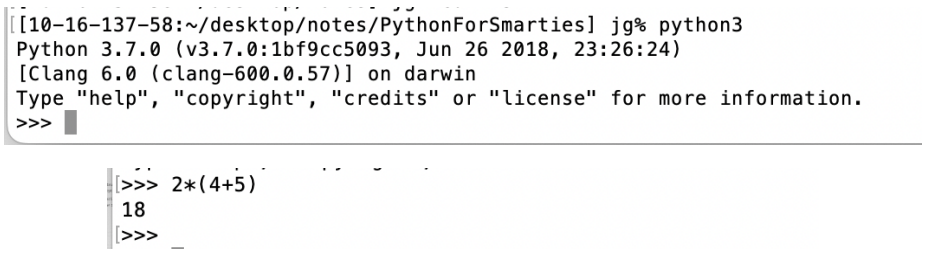

Figure 1: Typing at the command line. In a terminal window on my mac I type the command `python3`, which starts the Python interpreter. The `>>>` at the bottom is the command prompt. At the Python command prompt I type: `2*(4+5`. The interpreter "interprets" this and returns the value `18`.

the name `x` might be bound to an object that contains a single number. Any name can be bound to any kind of object. For example, the name `x` could be bound to a character string.

There may be more than one name bound to the same object. Each object "knows" the number of names bound to it. This number goes up when another name is bound to it and goes down when a name bound to it is "re-bound" to a different object. The only to access an object is through a name bound to it. Therefore, if the number of names bound to an object goes to zero, the object can never be accessed. When that happens, the interpreter can *garbage collect*, which means re-claiming the memory occupied by the now useless object.

Figure 2 illustrates some of this. The Python command `dir()` returns a list of names in the environment (this will be said more correctly later). When the command line interpreter starts up, it creates some default names: `__annotations__`, $\cdots$, `__spec__` . A name is a character string, so the interpreter prints them out with quotes. The double underscores before and after are a Python programming convention to indicate to a programmer not to "touch" them. You could, for example, type the commend `>>>__spec__ = 4`, but you should never do that. The command `x` (typed at the command prompe `>>>`) tells the interpreter to return the value in the object the name `'x'` is bound to. But that name is not defined so the interpreter returns an error message. The command `x=2` tells the interpreter put the name `'x'` into the environment, create an object whose value is 2, then bind the name to the object. The next `dir()` command shows that the environment now contains the name `'x'`. The command `x=2` changed the environment. Finally, typing the command `x` now gives the value of the object `'x'` is bound to, which has the value 2.

Figure 3 illustrates more about the "names bound to objects" system in Python. Every object has a type, which you can access using the `type(name)` function. After the command `x=2`, the command `type(x)` returns the information that the object `'x'` points to is of the class `int`. (more about *classes* in Python later). The command `x = "Hello world"` creates a new object of type `str` (for *string*). The object `'x'` used to point to (the number 2) now

```
[10-16-137-58:~/desktop/notes/PythonForSmarties] jg% python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x=2
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'x']
>>> x
2
>>>
```

Figure 2: Names in the environment. The Python command `dir()` returns a list of all the names in the environment. The command `x=2` creates the name `x` and an object whose value is `2`, then it binds the name to the object. See the text for a more detailed explanation.

has no names bound to it and can be garbage collected. Unlike *strongly typed* languages, each assignment can change the type of the object a name points to. This is a convenience, and also a common source of bugs.

The interpreter infers the type of an object from the command that creates it. It correctly inferred[2] that 2 is an integer and "Hello world" is a string. The command `L = [1, "Hello"]` creates an object of type `list` (the square brackets `[...]` and the comma separated values say "list"). The elements of a list can have different types. The assignment command `M=L` creates a new name, `M` but it does not create a new object. Instead, the name `M` created and bound to the same object the name `L` is bound to. The command `L.append(3.14)` modifies this object by *appending* `3.14` to the end of the list object. Both names `M` and `L` point to this object. That's why, the command `M`, which returns the value of the object `M` is bound to, now gives the three element list. Matlab and Julia are different: The command `M=L` in Matlab would create a new object for `M` that was given the value contained in the object `L` is bound to. Thus, changing an entry in `L` would not change `M`.

The last commands of Figure 3 illustrate that this behavior is different for *immutable* objects like strings. The command `x = "Goodbye world"` does not change the object `'x'` was bound to. Instead, it creates a new object (the string `"Goodbye world"` and binds the name `x` to this new object. The object `y` was bound to is not modified. Strings and numbers are immutable but most other types are *mutable*. The command `L.append(3.14` changes ("mutes" or "mutates"?) the list object `L` is bound to, but it does not create a new object.

[2]The interpreter has rules that determine the type of something whose type is not already determined. For example, a string of digits is assigned the type `int` and a string of characters in quotes is assigned the type `str`.

```
[10-16-137-58:~/desktop/notes/PythonForSmarties] jg% python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 2
>>> type(x)
<class 'int'>
>>> x = "Hello world"
>>> type(x)
<class 'str'>
>>> L = [1, "Hello"]
>>> type(L)
<class 'list'>
>>> M = L
>>> L.append(3.14)
>>> L
[1, 'Hello', 3.14]
>>> M
[1, 'Hello', 3.14]
>>> type(L[2])
<class 'float'>
>>> y = x
>>> x = "Goodbye world"
>>> y
'Hello world'
>>>
```

Figure 3: Re-assigning names, types of objects, immutable and mutable objects. The Python command `type(name)` returns the type of the object the name is bound to. An assignment statement can change this type without changing the name. A Python `list` is *mutable*: such an object can be modified. A Python "number" (object of type `int` or `float`, etc.) is *immutable*: it cannot be changed, only created accessed, or destroyed.

3 OS Command line and file system

Many readers already know most of the material in this section, and many do not. Many Python users prefer to use a complete IDE such as Jupyter notebooks or PyCharm. These have convenient features. But, if you're not a dummy, the operating system file manager, a code editor, and the command line together is a more powerful coding environment.

The Python code for doing a task is probably contained in more than one file, or *module*. An *operating system* has a *file manager* that can be used for storing code modules, data and output. The command line allows you to copy and move files, and to move around and look around in the file system. This is a powerful way to organize and run your computing project.

Most computers have an *operating system* that either is a form of Microsoft Windows or Posix. *Posix* is a generic term for operating systems that are based on the *Unix* system developed in the 1970's. An Apple laptop probably runs a version of MacOS, which is Posix. Hard core hackers may use Linux, which is even closer to the original Unix.

Both Windows and Posix systems come with *window managers* that allow you to open a *terminal window* in which you can enter commands to the operating system at the *command line*. This is the operating system (OS) command line, not the Python command line discussed in Section 2. Figure 1 has an example of this. The top line has a prompt that ends with[3] `jg%`. I typed the command `python3` and then "return" (often called "enter"). This tells the operating system (a version of MacOS in my case) to find the command `python3` and execute it. If you or your computer cannot find the command to run the Python program, you should consult someone who knows a bit more. In Posix systems, it is likely that the appropriate command is not in your *path*.

The OS (Windows or Posix) has a *file system* which is a collection of files and *directories* (called *folders* in MacOS). A directory (folder) is a file that contains other files, some of which can be other directories. The system of directories and files in directories it the file system *hierarchy*. The hierarchy starts at a *root* directory. Any file in the hierarchy has a *path* from the root directory. When you open a Posix terminal window, you can enter the command `pwd` (for "print working directory") and it should print the path, which is a sequence of directory names separated by slashes. For example, I got `/Users/jg/desktop/notes/PythonForSmarties`, which means a directory called `Users` in the root directory, then a directory called `jg` in the `Users` directory, and so on. The directory `Users` is a *subdirectory* of the root directory, and `jg` is a subdirectory of `Users`, and so on. The command `ls` (for "list") prints a list of files in the current directory. Some of these files could be subdirectories. The Windows OS has commands with different names that do these things (`cd` and `dir`).

Your OS has commands to create new subdirectories and to move up and down in the file system hierarchy. You can do these things directly at the com-

[3] My username on my laptop is `jg`.

mand line or using your window/file manager. You probably want to create a new subdirectory for your Python modules and output. The Python interpreter typically accesses files in the *current directory* (the one you're in).

4 Modules

A Python script, the technical term is *module*, is a file that contains a sequence of commands. The Python *interpreter* reads the lines of a module and executes them one by one. Most Python programing involves editing and running modules. You edit a module using a code editor. I use `xcode`, but there are others, each with advantages and disadvantages. You can "run" a module by giving it as an argument to the Python command.

The Python functions `print()` and `str()` can be used to get output when the interpreter is working on a module. The command `print([string])` prints the argument `string` to the terminal. The command `str([name])` creates a string to represent the object that `'name'` points to. The operator `+` "catenates" two strings.[4]

Figure 4 shows a Python module open in a code editor (`xcode` on a mac) window. The file is called `first_demo.py`. The filetype `.py` indicates that it is a Python module. The first three lines are *comments*. A comment starts with the `#` symbol. The interpreter ignores the `#` and anything after it. Coders use comments to help someone reading the module understand it. Good comments are an essential part of good code (more on this later). The interpreter also ignores blank lines, such as line 4. Line 5 is a command, which the interpreter will evaluate just as it would have at the Python command line. This command creates the name `'greeting'` in the environment, it creates an object which is the string `"Hello world"` and binds the name to the object.

Figure 5 shows the *execution* ("interpretation" would be more correct) of the module. This takes place in a `terminal window`. The command `python3 first_demo.py` finds the command[5] `python3` and passes it the file `first_demo.py`. The interpreter then reads the lines of the file `first_demo.py` one-by-one and interprets them much as it would have done at the Python command line as i Figure 3. Look at Figures 4 and 5 together. Line 6 of Figure 4 tells the interpreter to print the string that the name `'greeting'` is bound to. Figure 5 shows that this resulted in `Hello world` appearing in the terminal window. Line 7 creates the name `'x'` and binds it to the value 2 (actually, to an object with type `int` and value 2). Line 8 shows one way to print this value. You can't print it directly (maybe you can in some forgiving Python systems) because it isn't a string. The function `str()` creates a string to represent the integer 2, which is the string `'2'`. The function `print` then prints this string. Lines 9 and 10 illustrate this process. Line 9 creates the name `'x_string'` and binds it to a string object. Line 10 then "passes" that string object to the `print` function.

[4] The correct English word is *concatenate*, but computer people say *catenate* instead.

[5] I renamed the Python 3 command `python3` to distinguish it from the earlier `python2`. You probably do not need to do this.

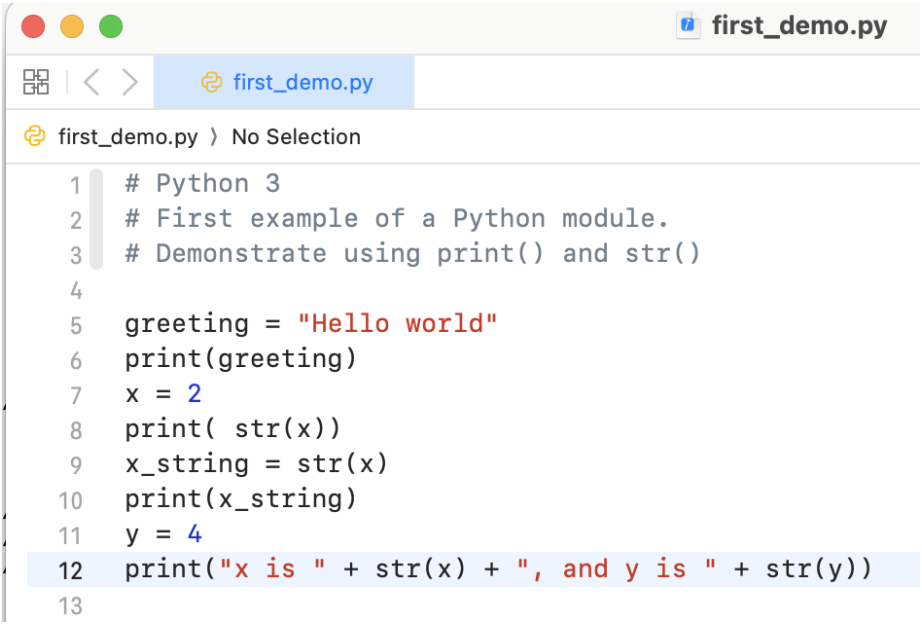

Figure 4: A Python module open for editing in a code editor. Most code editors put line numbers on the left and have "code aware" color schemes. This one makes comments grey, code in black, and strings in red.

Figure 5 shows that you again get `2` in the output. Line 12 creates an output line that is easy to read (more on this later). It catenates the string `'x is '` with the string `'2'`, which makes the string `'x is 2'`. Note the space between `'is'` and `2`, and the comma after the `2`, both of which make the output line easier to read.

5 Control flow

Control flow is code that tells the computer (the Python interpreter) which lines of code (commands) to execute (interpret). A *code block* is a sequence of lines of code that can be controlled in this way. In Python (and only Python, as far as I know) a code block is determined by a colon ":" and indentation. In Figure 6, line 8 ends with a colon. That tells the Python interpreter that all the following lines that are indented are part of a code block. That code block is lines 9 through 13, because they all are indented. Line 14 is not part of this code block because it is not indented. Lines 11 through 13 are a *nested* code block that is contained in the *outer* block starting at line 9. Lines 11 through 13 are indented by 6 spaces (marking the nested inner block) while lines 9 and 10 are indented by 3 spaces, indicating that they are in the outer code block but not the inner one. All lines at the same level (inner or outer in this case)

```
[10-16-137-58:~/desktop/notes/PythonForSmarties] jg% python3 first_demo.py
Hello world
2
2
x is 2, and y is 4
[10-16-137-58:~/desktop/notes/PythonForSmarties] jg%
```

Figure 5: Execute the module `first_demo.py` in a terminal window. I issue the command `python3 first_demo.py` to the operating system and the output appears below as the module executes.

must be indented the same number of spaces.[6]

Lines 8, 10, 13, and 14 create the *control flow* for this code. Line 8 creates a *loop* out of the code block in lines 9 to 13. The function `range(L)` returns a *list* with elements `0, 1, ··· L-1` (more on lists below). The list includes its starting value, 0, but excludes its ending value, L. This means that there are L items in the list, but the largest is $L-1$. They sometimes call lists like this *half open* because they are *closed* at the left end (starting value) and *open* on the right (leave out the ending value). It is a quirk of Python to choose to have the "right" number of elements in the list at the expense of having the "wrong" ending value. The Python *for loop* starts with the *keyword* `for`. You say: `for [name] in [list]:` to tell Python to execute the code block that follows with `[name]` (`n` in this example) bound to each object in the list, and in the order they are in the list. In this example, the code block of lines 9 t0 13 is executed first with `n` bound to 0, then bound to 1, and so on. Line 8 says `range(n_max+1)` so that it will execute with `n` equal to `n_max`. To summarize: a *loop* is a code block that is executed many times with one or more variables in the loop taking different values.

Line 10 of Figure 6 is an `if` test. The *keyword* `if` says: evaluate the logical expression. If it evaluates to `true`, then execute the code block that comes after (note that the line ends with a colon). Otherwise, skip the code block and go to the next line at this indentation level. In this case, if $s > s_{\text{target}}$, then execute the statement `print(···`. Otherwise (if $s \leq s_{\text{target}}$) skip to the next statement at this level in the current code block. But there are no more statements at this level in this code block, so it skips to the end and starts the `for` loop code block again with the next value in the `range` list.

Line 13 of the inner code block in Figure 6 is just the keyword `break`. That tells the interpreter to "break" out of the loop in the outer code block (the one that started at line 9). If the `break` command is executed, the next line executed will be line 14. You might wonder why `break` breaks out of both the inner and outer code blocks. One reason is that you don't need a special statement to

[6] You can indent using tabs instead of spaces. Excellent Python programmers argue with each other, a lot, about which is better. Personally, I go with spaces. Click on the link to see a *serious* discussion of this issue.
https://www.thewrap.com/silicon-valley-fact-check-why-richard-is-wrong-on-tabs-versus-spaces/ .

```python
# Python 3
# Illustrate a loop, control flow, and a code block

n_max    = 50        # the max number to square
ssq      = 0         # sum of squares
s_target = 300000    # step when ssq exceeds the target

for n in range(n_max+1):
   ssq = ssq + n*n     # ssq = sum of squares 1 + 4 + ... + n^2
   if ( ssq > s_target ):
      print("got ssq = " + str(ssq) + " with n = " + str(n))
      print("   the target was " + str(s_target))
      break
if ( n == n_max ):
   print("did not reach target.  ssq = " + str(ssq) + " at n_max = " +
         str(n_max))
   print("increase n_max or go home")

```

Figure 6: Code to illustrate code blocks and control flow.

```
[JonathansMBP20:~/desktop/notes/PythonForSmarties] jg% python3 loop_demo.py
got ssq = 31395 with n = 45
   the target was 30000
[JonathansMBP20:~/desktop/notes/PythonForSmarties] jg% python3 loop_demo.py
did not reach target.  ssq = 42925 at n_max = 50
increase n_max or go home
[JonathansMBP20:~/desktop/notes/PythonForSmarties] jg%
```

Figure 7: Run the code in Figure 6, at the command line, with three different values of `n_max` to illustrate conditional execution and branching.

break out of the inner code block, just stop indenting.

Now you can see what the code does. It gives `n` the values `0, 1, 2,`$\cdots$ and computes `ssq` 0, then 1, then $1 + 4 = 5$, then $5 + 9 = 14$, and so on. It keeps doing this until the `range` lest is exhausted when `n` is `n_max`, or until the sum of squares is larger than the target.

6 Namespace and importing

The module file system in Python allows you to *modularize* your code by organizing different parts of the code into separate files. This kind of *modularity* is a basic principle of software design. It allows you to package your code into modules that others can use and it allows others to do this for you. Commonly used Python packages (modules) include `numpy` (scientific and numerical computing), `pandas` (for storing and manipulating data), etc.

A *namespace* is a list of names. The namespace itself is an object that must have at least one name bound to it. Python uses `namespace_name.object_name` to refer to the name `object_name` in the namespace object that `namespace_name`

is bound to.

The namespace mechanism in Python helps organize the information one module learns by *importing* another module. The command `import [file_name] as [name]` tells the interpreter to execute the code in the module `file_name.py`. The file `file_name.py` should contain a sequence of Python commands. The interpreter executes these commands. Some of these commands create new names. When the import is finished, the interpreter creates a namespace object in the calling module's environment with the name `name`. All the names created executing the imported module go there.

Figures 8 and 9 illustrate the import and namespace system. Figure 9 is a module (file) whose filename is `importModuleDemo.py`. This file has three commands. Lines 5 and 6 create names `greeting` and `saying`. Lines 8 through 12 print the names[7] in the environment. The first lines of output (Figure 10) have those names. Line 8 of the imported module `importModuleDemo.py` (Figure 9) makes a list of names in the environment created by executing the commands of `importModuleDemo.py`. The command on line 5 in `namespaceDemo.py` (Figure 8) tells the interpreter to execute the commands in `importModuleDemo.py` and to create a namespace `id` for the names created. The output (Figure 10) shows that `importModuleDemo.py` is executed first, so its output appears before the output created by later commands in `namespaceDemo.py`.

The rest of `namespaceDemo.py` illustrates the namespace mechanism. Line 13 prints the names in the environment after the import (line 5) and the assignment (line 7). These are `greeting` and `id`. The name `greeting` appears both in the environment and in the `id` namespace. The output from lines 15 and 16 shows that these names are bound to different objects. Line 18 adds a new name to the `id` namespace. Line 23 prints names in the `id` namespace (see Figure 10). You find the names `greeting` and `saying` that were created in lines 5 and 6 of `importModuleDemo.py` (Figure 9). The name `y` was just explained. The name `names` was added to the environment of `importModuleDemo.py` by line 8 there, and `name` was created at line 10. People often avoid clutter in modules used as imports to avoid cluttering the resulting namespace. It's also bad form to have the import module make output. The person using the module may have trouble understanding such output, particularly if the import module was written by someone else in a different place and time. Lines 25 and 26 of `namespaceDemo.py` illustrate that names in a namespace can be re-bound just as names in the environment can be.

7 Keywords and language aware code editors

A *keyword* is a character string (word) that has a reserved meaning and cannot be used as a name. Some keywords have pre-assigned values. For example, the keyword `True` always refers to the boolean "true". A *boolean* variable can only be "true" or "false". A keyword in a command tells the interpreter how to interpret

[7] The conditional on line 11 avoids printing names such as `__spec__` that are automatically in the environment but which the user is not supposed to access.

```python
# Python 3
# Illustrate importing a module and the namespace it makes
# filename: namespaceDemo.py

import importModuleDemo as id

greeting = "I say hello"

names = dir()
print("In namespaceDemo.py, the top level names are:")
for name in names:
   if ( name[0] != '_' ):   # don't print names starting with underscore
      print("  " + name)    # print the name indented by three spaces

print("greeting is:     " + greeting)
print("id.greeting is:  " + id.greeting)

id.y = 5
print("In namespaceDemo.py, the names in id are:")
id_names = dir(id)
for name in id_names:
   if ( name[0] != '_' ):
      print("  " + name)

id.greeting = "I forgot"
print("id.greeting is:  " + id.greeting)
print("id has type: " + str(type(id)))
```

Figure 8: Code that imports the module `importModuleDemo.py`.

```python
# Python 3
# A demo module to be imported
# filename: importModuleDemo.py

greeting = "The module says hello"
saying   = "learn Python"

names = dir()
print("In importModuleDemo.py, the top level names are:")
for name in names:
   if ( name[0] != '_' ):   # don't print names starting with underscore
      print("  " + name)    # print the name indented by three spaces
```

Figure 9: The module being imported in Figure 8.

```
[[JonathansMBP20:~/desktop/notes/PythonForSmarties] jg% python3 namespaceDemo.py
In importModuleDemo.py, the top level names are:
  greeting
  saying
In namespaceDemo.py, the top level names are:
  greeting
  id
greeting is:     I say hello
id.greeting is:  The module says hello
In namespaceDemo.py, the names in id are:
  greeting
  name
  names
  saying
  y
id.greeting is:  I forgot
id has type: <class 'module'>
```

Figure 10: Output from running `namespaceDemo.py` (Figure 8).

the part of the command that comes after the keyword. For example, the command `import numpy as np` starts with the keyword `import`. This tells the interpreter to look for a file (module) with filename `numpy.py` and to "import" it (see Section 6). The word `as` is another keyword, which tells the interpreter to use the following name as the name of the resulting namespace.

Most programming languages use keywords. A language aware editor usually has a distinctive color for keywords in the language being edited. Figure 6 shows that `xcode` uses magenta for keywords in Python. This shows that `for` and `in` in line 8 and `break` in line 13 are keywords. Language aware editors try to help the programmer in other ways. Figure 6 shows that default code is in black. Character strings are red, numbers are blue and comments are gray. Line 12 in Figure 4 shows how this can be helpful. If the end quote after the string `x is` were missing, the part `+ str(x)` would be red, showing it's part of the string, rather than black. Fancier language aware editors try to be even more helpful. Some will raise a popup box. if your mouse hovers over a name, telling you something about that name. Experienced coders usually develop a strong preferences for a specific code editor or set of code editor features. You should try a few to see which ones are most helpful to you.

A common mistake is using a keyword as a name by mistake. For example, suppose there is an index i that depends on other indices k and n in formulas with expressions i_k and i_n. It might be natural to code these using names `ik` and `in`.

```
ik = 4       # fine
in = 9       # error, "in" is a keyword.
```

You can do a web search to find the keywords in the version of Python you are using.

8 Defining functions

Most programming languages have a notion of *function* (also called *method* or *procedure* or *subroutine*). The function mechanism is supposed to provide two services. It should allow you to create code once that to do something and then apply that code repeatedly in different situations. It also should allow you to apply the operation on different pieces of data with different names.

A function is defined by a block of code starting with the `def` keyword. What follows is the name of the function, its arguments, a colon, then an indented code block that defines the function. The code in the function definition is executed by the interpreter every time the function is *called* (invoked). The environment for this execution is a modification of the environment that exists when the function is called.

9 Data Structures

A *data structure*, in programming, is a systematic way to store data. A good data structure lets you add, remove, and access data in the structure in the simplest way possible for the specific application. Python has native data structures that coders use all the time. Here are a few of them and a few of their features. The Python documentation has more information.

9.1 List

A *list* is a sequence of objects, which you might think of as $[\mathcal{O}_0, \cdots, \mathcal{O}_{n-1}]$. A list has a name. You access object k by putting `[k]` after the name. For example, if the name `friends` is bound to a list, then `friends[3]` is bound to $\mathcal{O}_3$ in that list (the fourth object in the list). A list object is *mutable*, which means a command can modify an existing list.

The objects in a list do not have to have the same type. For example, `rlist = [3, "tummy", ["a", "z"]]` is a list with three objects. The first is an integer. The second is a string. The third is a list. Good coders would not use this feature very often. I mention it mainly to emphasize the simplicity: a list is just a list of objects. Each object has its own type information, as all Python objects to.

List is a built-in datatype in Python that comes with many operations. some important ones are

- `[]` is an *empty list*. The command `friends = []` binds the name `friends` to an object that is a list with no elements.
- `append` is an attribute of any list objectc that adds an object to the end of a list. For example, if `friends` is the list `["David", "Mark"]`, then the command `friends.append("Danny")` changes (mutates? mutes?) the object to `["David", "Mark", "Danny"]`

- In this example, `friends[2]` is bound to `"Danny"`. You can treat `friends[2]` as a name, which can be bound to another object. For for example, `friends[2] = "Daniel"` changes the list to `["David", "Mark", "Daniel"]`.

- A *for loop* can "iterate" over the objects in the list. For example, writing `for person in friends:` executes the code block (the next lines properly indented) with the name `person` bound to each object in `friends`. For example, the code

```
friends = []
friends.append("David")
friends.append("Mark")
for person in friends:
   print(person)
print("Some of my friends")
```

  produces the output

```
David
Mark
Some of my friends
```

- `len` is a function that returns the number the number of elements in its argument. For example, `len(friends)` would be 2 before appending `"Danny"` and 3 after.

The Python documentation has many more attributes of list objects.

9.2 Tuple

A *tuple* is a list of names bound to objects. Some differences between tuples and lists are

- A tuple is not mutable. A tuple object cannot be modified once it's created. For example, you can add to a list as in `friends.append("Danny")` but not a tuple.

- Tuples use parentheses (round braces, parens) instead of square braces. Thus `tup = (1,2,3)` makes a tuple while `lis = [1,2,3]` makes a list.

- A one element tuple needs a comma to tell the interpreter it's not just an object in parens. Thus `obj = (2)` creates an integer whose value is 2, while `tup = (2,)` creates a one element tuple.

- The parentheses are optional. Thus, `tup = 1,2,3` creates a three element tuple. Style guides tell you to use the parens, but some of the Python code you read doesn't follow this rule.

9.3 Unpacking

Putting data items into a data structure is *packing*. *Unpacking* is binding names to the data items in a structure. For example, `lis = [1,2]` creates a list data structure with two data items. Then `[one, two]= lis` binds the name `one` to the data item `1` and the name `two` to the object `2`.

Methods can return more than one object by packing and unpacking into and out of a tuple. For example, imagine a method `intersection` that determines where two lines in the plane cross. It can return the x and y coordinates using a tuple, as

```
return (x,y)
```

The object returned is a two element tuple. The calling code can receive these values using

```
(x,y) = intersection(  ... some data ... )
```

As another example, you can create plots using the plot package

```
import matplotlib.pyplot as plt
     ... lots of code ...
fig,ax = plt.subplots()
     ... code for plotting ..
```

The `fig,ax` on the left unpacks the two element tuple returned by `plt.subplots()`. The style guide says it should be `( fig, ax) = ...`, but whoever wrote this line of code (copied from the official matplotlib documentation, actually) did not follow the style guide.

9.4 Dictionary

A *dictionary* is a collection of names bound to objects. It acts like a list except that objects are accessed by name rather than by number. Python uses *curly braces* (*curlys*) {} for dictionaries. If `glossary` is a dictionary and `name` is a string, then `glossary[name]` is bound to an object. Lists work like this, except that the *index* of a list is an integer rather than a string. In this example, the first line creates a dictionary object (which is empty) and binds the name `glossary` to it. The next three lines modify the dictionary object by adding new entries. The "index" (a string instead of a number) goes in square braces. The last line shows that you can access an item in the dictionary by name.

```
glossary = {}              # a dictionary with no entries
glossary["geek"] = "A person who likes coding"
glossary["nerd"] = "A person who likes math"
glossary["snob"] = "A person who hates cheap restaurants"
print(glossary["geek"])  # get: A person who likes coding
```

The dictionary data structure is a Python implementation of what algorithms people call a *hash table*, or *associative array*. A hash table uses an actual array such as a Python list together with a *hash function*. In Python dictionaries, the hash function takes a string and produces an integer that can be used as an array index. It is possible that two strings "hash" to the same index (the has function gives the same integer output for distinct strings). This is a *collision*. A hashing system needs a way to *resolve* collisions (which must happen sometimes). A book on computer algorithms will describe several collision resolution strategies. I don't know which one Python uses.

You can iterate over the entries in a dictionary

```
for key in glossary:
   print(key + " means: " + glossary[key])
```

The name `key` will be bound to each of the keys in the dictionary. Each key is a string, so you get

```
geek means: A person who likes coding
nerd means: A person who likes math
snob means: A person who hates cheap restaurants
```

The "definition" does not have to be a string and different definitions don't have to have the same type. Figure 11 illustrates some of these features. Line 6 creates an empty dictionary. Lines 7 through 9 add entries. The `"name"` entry is a string. The `"age"` entry is an integer (and I don't know whether Roger Federer is 48 years old). The `"big wins"` entry is a list. Line 11 shows that you can define a dictionary all at once. The syntax involves curlys (like all dictionaries), with a list of pairs separated by commas. Each pair has the form `name:value`. Again, the values have different types. Line 13 creates a list with two objects in it, each object being a dictionary. Line 14 starts a standard `for` loop, in which `star` will be bound to the objects in the dictionary one by one. Line 15 accesses two entries in each dictionary. Note that the key is always a string while the value can have any type. The last part is a little Python exercise. First, `star["big wins"]` produces the list of big wins (each star, obviously has more big wins than these). You access the first element of the `"big wins"` list with `[0]`. Using parentheses, as `(star["big wins"])`, says to apply the `[0]` to that list. These parens are not necessary, but I hope they make the code clearer.

A dictionary is a way to bind names to objects. The key of a dictionary entry is the name and the value is the object. That is also what a namelist does. In fact, namelists are implemented using dictionaries. In particular, a value of an entry in a dictionary could be another dictionary, just as a name in a namelist can be bound to another namelist.

9.5 Numpy arrays

Numpy (for *num*erical *Py*thon) is a module that defines mathematical constants (such as `np.pi` $\approx \pi$), functions (such as `np.sqrt(x)` $\approx \sqrt{x}$) and data structures

Figure 11: Creating and working with a dictionary.

such as arrays and matrices. The Numpy documentation describes these in detail. The `ndarray` (for n dimensional array) data structure is much like an *array* in Java or Fortran and not like a Python list. For example,

```
A = np.zeros([2,3], np.float64)
```

creates an object of type `<class 'numpy.ndarray'>` (a numpy `ndarray`). The first argument is the two element list `[2,3]`, which is the *shape* of the array. In this case, `A` will by a two index array with two rows and three columns. The second argument is a type for the entries of this array. Here, `np.float64` is a type defined in numpy (so it's in the `np` namespace) that is a 64 bit float. The 64 bit float is the default for numbers in numpy arrays, so `A = np.zeros([2,3]` does the same thing.

Figure 12 illustrates some features of the numpy array class. Line 6 creates a one index array with 4 elements. It specifies that these elements are 32 bit integers. Line 7 makes the second entry equal to the integer 4. Printing (line 8) shows that there are four entries, three zeros and one 4. Line 9 creates a new object, which is an array but now with the default type `np.float64`. Line 10 tries to assign `x[1]` to the integer 4. The output shows that this did not happen. Numpy knows the entries of this `x` are floats, so the integer 4 is first converted to the float 4., the decimal point indicating that it's a float. The zeros also are 0. (indicating the floating point zero) rather than 0 (the integer zero. The two lines of out are at the bottom of Figure 12. Line 10 assigns `x[1]` the integer value 4. If `x` were a `list` rather than a numpy array, the value of `x[1]` would have become 4 (the int) rather than 4. (the float).

You should use numpy arrays rather than lists for one index arrays in numerical calculations. Some reasons are:

- Performance: operations on arrays are faster because
  - Arrays are more "light weight". Every array element has the same size and type so they can be arranged more systematically in computer memory (less work needed to find them) and the type does not need to be looked up for each operation.

– Operations can be *vectorized*.

Performance is a more important issue in Python than in many other languages because Python is *interpreted* rather than *compiled*. Most of the time in executing the command `x = y+z` is spent figuring out what the command is. That means reading the characters, deciding which are names and which are operations, what types the variables have, where they are in memory, etc. The actual addition takes less than 1% of that time. In many cases code written in Python can execute almost as fast as code written in high performance languages (C, Fortran, etc.), but you have to avoid "scalar" code. Python's integrated linear algebra methods are (for large problems) fast. Vectorized code can use single commands to do a lot of arithmetic.

- There are many numpy methods that take numpy arrays as arguments or produce arrays as output. Simple examples include vector dot product and matrix vector multiplication. The methods in Python that do these things are much much faster than methods you would write yourself.

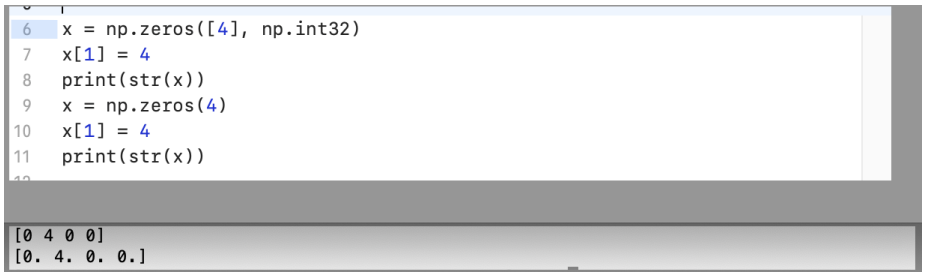

Figure 12: Illustrating some aspects of Numpy arrays.

10 Classes

Classes in most programming languages are a way for the coder to define a new kind of object, a *data type*, and give it properties for some specific purpose. A data type is defined by what it can do and the kind of data it holds. The *built-in* data types in Python include various kinds of numbers strings and container classes (lists, dictionaries, etc.). A *class* is a programmer-defined data type. An object whose data type is that class is an *instance* of the class. For example, financial software might have a `Transaction` class[8] whose instances describe financial transactions.

The class mechanism in Python is well described in the Python documentation[9]. It is simple and "lightweight". This simplicity allows a lot of freedom

[8]The *Style Guide* says names of classes should be capitalized, as in `Transaction` rather than `transaction`. See https://peps.python.org/pep-0008/#class-names.

[9] https://docs.python.org/3/tutorial/classes.html

in defining and modifying class objects. Instead of *protections* offered by, say, C++, it is up to the class class designer and user to follow coding guidelines and not to do things that are forbidden in other languages. For example, C++ allows a class instance to have *private* data that cannot be accessed by code outside the class. Instead of that, Python style guides say a name you don't want accessed from the outside should start with an underscore character. For example the name `_rank` in a class is intended to be private while `rank` is not.

Suppose `Transaction` is a class, then the command

```
sale = Transaction( price, date, customer)
```

creates an object whose type `Transaction`. The *constructor* code (see below) for the class will do something with the input data `price`, etc. There will be a namespace associated to this object. Names in that namespace are *attributes* of the object. For example, there might be a method `customer_type()` that returns information about the customer. The command

```
is_happy = sale.customer_type("mood")
```

might return `True` if the customer of that particular transaction is happy. Part of executing this command is "asking" the object `sale` for an attribute `customer_type` that is a function. The interpreter doesn't know or care what the type or class of `sale` is. It only needs the `sale` object to have a `customer_type` attribute.

Figure Figure 13 illustrates how a class can be defined and used. In line 6, `class` is a *keyword* that says the command is defining a class. The name of the class is `poly` (for "polynomial"). The colon says that indented lines 7 to 19 define the class. The names defined in these commands become attributes of any instance of the `poly` class.

Line 7 defines a method object called[10] `__init__`. This is the *constructor*, which is called whenever a new instance of the class is created. Line 46 creates an object of the class `poly` and binds the name `p` to it. You can think of `poly(a)` (the right side of line 46) as a call to the method[11] `__init__` in the `poly` class definition (starting on line 7). The constructor "constructs" the new class instance. In a more complicated class, the constructor might ask for memory or open a file, record the existence of the new instance in a database, etc.

[10]The underscore characters before and after `init` are a warning that you should never call `__init__` yourself. If you create a class that has an attribute you don't want anyone to use (except you), you put one or two underscore character before its name. For example, if you want each instance to have a unique serial number that never changes, you can call it `_serialNumber`. If `car` is an instance of this class, the command `car._serialNumber=0` would set `_serialNumber` to zero. Python programmers have to promise never to do this, even though they can. The Python interpreter calls `__init__` whenever you create a new instance of the class, but you should not call `__init__` yourself.

[11]The full story is a little more complicated because there are initialization/constructor actions involved besides those in the `__init__` definition.

Notice that the call on line 46 has only one argument, `a`, while the `__init__` method definition on line 7 has two arguments, `self` and a. The `self` argument[12] is a namespace that holds names bound to data that a class instance needs to "remember". This makes a class instance a good place to store information. Lines 8 stores the given coefficient array `a` in the `self` namespace. The constructor also computes and stores the degree of the polynomial (lines 9 and 10). About line 9: the `shape` attribute of (the numpy `ndarray`) `a` is a tuple that contains the dimensions of the array. In this case `a` has only one index and only one dimension. The left side "unpacks" this tuple into a name `dp1` (for "degree plus one"). Line 10 creates the name `deg` in the `self` namespace and binds it to the degree of `p`. This illustrates the idea that a constructor can do more than just store arguments in the `self` namespace.

Line 11 creates a function `f` that is an attribute of any instance of the class `poly`. The first argument is the `self` namespace, which is where `__init__` put the polynomial coefficients. In Line 47, `p.f` refers to the `f` attribute of the object `p`. This attribute is a function that takes one argument, when it is called from the "outside" like this. The interpreter adds the `self` argument before executing the function definition in lines 12 to 19. Lines 12 and 13 unpack data stored in `self`. The rest evaluates and returns the value of $p(x)$ at the given x.

The `integrate` function called on lines 58 and 61 illustrate how this class mechanism can be used. The code calculates (estimates) the integral using a version of the *trapezoid rule* from calculus.

$$I = \int_a^b f(x)dx \approx \sum_{k=0}^{n-1} f(x_k)\Delta x \quad , \quad \Delta x = \frac{b-a}{n} \quad , \quad x_k = (k+\tfrac{1}{2})\Delta x \; . \qquad (1)$$

The last argument to the `integrate` method defined on line 36 is `f`. The object passed as `f` is supposed to have an attribute (also called `f`, which is common in Python code). This attribute, `f.f` is assumed to be a function that returns a value for any argument `x`. Line 41 uses `f.f(xk)` to get the value of $f(x_k)$ that is used in the integration formula (1). The Python interpreter does not ask what type the object passed as `f` is. Indeed, lines 59 and 61 pass objects of two different types, `poly` for line 58 and `es` for line 61. When executing line 41, Python will ask the object that the name `f` points to for its attribute `f`. The `poly` class defines such an attribute starting on line 11. The `es` class defines its `f` attribute starting on line 27. Both of these definitions use the data stored in the `self` namelist. When this code is executed, `self` will be the namelist for the instance passed to `integrate`, either `p` (line 58) or `cosh` (line 61). It also has a value of `x` passed from line 41.

This is a common pattern in numerical software. There is a computational method for learning something about a function, its integral in this case. This method needs to know what function it is working on. The user needs a way to specify the target function to the computational routine (`integrate` in this example). It does this by passing an object that "knows" the function. In this

[12]The name `self` is a convention. An evil Python coder could give it another name.

case, "knowing" f means being able to evaluate $f(x)$ for any x. The object passed has to have an attribute `f` that appears to `integrate` to be a function of one variable `x`. The class mechanism with its `self` namespace allows the function to know the rest of the data that defines it. The `integrate` function does not know how f is defined. Indeed, the two functions are defined in two ways. One is a polynomial

$$f(x) = a_0 + a_1 x + \cdots + a_d x^d \; .$$

The other is an exponential sum

$$f(x) = \sum_{k=1}^{n} c_k e^{a_k x} \; .$$

The data needed to define these functions is different. A polynomial requires a single array for the coefficients a_k. An exponential sum requires one array for the exponential rates a_k and another for the amplitude coefficients c_k.

The specific polynomial in this case is

$$p(x) = 1 + 2x + 3x^2 + 4x^3 \; .$$

The exact integral is (dust off your calculus book)

$$\int_0^1 \left(1 + 2x + 3x^2 + 4x^3 \right) dx = 4 \; .$$

The exponential sum is the hyperbolic cosine defined by

$$\cosh(x) = \frac{e^x + e^{-x}}{2} \; .$$

The integral being calculated is

$$\int_{-1}^{1} \cosh(x) \, dx = e - \frac{1}{e} = 2.350402 \cdots \; . \tag{2}$$

The output from this code is in Figure 14. The first two lines of output test that the `f` attributes of the polynomial and the exponential sum give the right function values. The numerical approximation to the integral of the polynomial is 3.9925, which is close to the exact answer 4. The numerical approximation of the cosh integral is $2.35036 \cdots$, which is close to the exact answer (2).

```python
# Python 3
# Illustrate a loop, control flow, and a code block

import numpy as np

class poly:                #  polynomial: a[0] + a[1]*x + .. + a[d]*x^d
   def __init__(self, a):
      self.a = a
      (dp1,) = a.shape
      self.deg = dp1 - 1
   def f(self,x):
      a   = self.a
      deg = self.deg
      f = a[0]
      xk = 1.    # will be x^k
      for k in range(1, (deg+1)):
         xk = x*xk
         f = f + a[k]*xk
      return f

class es:                  # exponential sum: c[0]*e^{a[0]*x} + ...
   def __init__(self, a,c):
      self.a = a
      self.c = c
      (n,) = a.shape
      self.n = n
   def f(self,x):
      a   = self.a
      c   = self.c
      n = self.n
      f = 0.
      for k in range(n):
         f = f + c[k]*np.exp(a[k]*x)
      return f

def integrate(a,b,n,f):      # integrate f from a to b using n points
   dx = (b-a)/n
   int = 0.
   for k in range(n):
      xk = a + (k+.5)*dx
      int = int + dx*f.f(xk)
   return int

deg = 3
a   = np.linspace(1., 4., deg+1)              # get a = [1,2,3,4]
p   = poly(a)                                  # p(x) = 1+2x+3x^2+4x^3
print("polynomial at x=2 is " + str( p.f(2)) )  # get 1+2*2+3*4+4*8 = 49

a = np.zeros(2)
a[0] = -1.
a[1] =  1.
c = np.zeros(2)
c[0] = .5
c[1] = .5
cosh = es(a,c)
print("cosh.f(.5) is " + str( cosh.f(.5) ))

ip = integrate(0., 1., 10, p)
print("the integral of the polynomial is " + str(ip))

ic = integrate(-1., 1., 100, cosh)
print("the integral of the hyperbolic cosine is " + str(ic))
```


Figure 13: A module that illustrates some basics of classes in Python There are no comments of docstrings because everything is explained in the text. Real code would have comments and docstings.

```
[10-17-15-126:~/desktop/notes/PythonForSmarties] jg% python3 classDemo.py
polynomial at x=2 is 49.0
cosh.f(.5) is 1.1276259652063807
the integral of the polynomial is 3.9925000000000006
the integral of the hyperbolic cosine is 2.350363214371499
```

Figure 14: The output from the code in Figure 13.