Scientific Computing, Courant Institute, Fall 2021
http://www.math.nyu.edu/faculty/goodman/teaching/SciComp2021/index.html

Python Part 1

Jonathan Goodman, September, 2021

1 About coding, Python and these notes

I have seen many smart students learn to code and do scientific computing. The process can be frustrating in the face of misunderstandings and subtle bugs. These notes represent my view of how to (learn to) code in Python in a way that is as efficient as possible in the long run. Please pay attention to everything here. The parts you already know will be quick reading. Most people in scientific computing have little formal training in coding, beyond a possible low level college class.

You need to be a good programmer to be good at scientific computing. You should understanding the language and software you're using. I have seen students lose much time trying to code something using trial and error from "how to" samples on the internet. You should always keep in mind quirks of numerical computing, such as the fact that numbers in the computer are (almost) never equal to the values in mathematical formulas. You should have a clear set of good programming practices that you follow always, even when they seem to slow you down slightly. These save time in the long run.

These notes are an introduction to Python programming designed for people learning to do scientific computing in Python. They describe the basic principles of the language in a way that *for dummies* descriptions may not. Experience shows that most new scientific Python programmers make serious coding errors (bugs and coding choices with serious negative consequences) as a result of learning by example, by imitating Matlab or C++, or from "for dummies" tutorials. Even if you think you know enough Python to get by, you should check that you understand the points made here. *If you are in one of my classes, points will be deducted from your homework scores if your codes violate some of the coding practices listed here.*

Python is a programming language well suited for *lightweight* computing. It has good *productivity*, meaning that it takes less time to code and debug in Python than, say, in C++ or FORTRAN. One way Python becomes lightweight is by making inferences about your intentions that C++ or FORTRAN force you to to "tell" the compiler. This forces the Python *interpreter* (the app that runs your Python code) to look up or infer the types of variables every time they are used. Such work done "under the hood" (referring to the moving parts "under the hood" of a car) often takes more time than the arithmetic operations that constitute most computational algorithms. If your code is very slow, find ways to make the computer do less of this.

You are not a dummy.
Don't program like one.

2 Getting set up

There are different *environments* for developing and running Python code. "For dummies" IDE environments can be easy to set up, but I urge you to avoid them while you are learning Python. Instead, code using three or four basic tools: a command line *terminal window*, a text editor (preferably but not necessarily, a "smart" one that gives Python specific colorings), a displayer for viewing plots in .pdf format, and (possibly) a window based file manager. More substantial computing projects, especially those involving more than one person, usually use a code management tool such as `git`.

If you use an Apple computer running a version of MacOS, a command line terminal window app called `terminal` comes with it. To use it, you will have to learn some `UNIX` commands such as `cd` and `ls` (possibly only these two!), and the structure of a `UNIX` file system. `Xcode` is an Apple supplied set of coding tools that includes a (somewhat) smart code editor. You may have to download it from Apple, but it's free. You may have to download and install a new version of Python together with the basic packages (`numpy`, `scipy`, `matplotlib`, etc.). You may have to manipulate your `PATH` variable so that the command `python` finds your downloaded Python rather than the "native" Python that comes with MacOS. This is because the native Python does not "see"' the packages that are necessary for scientific computing. Find an expert (instructor, classmate, older student, probably not an Apple store "genius) to help you with this. You can view a `.pdf` file by opening the icon of the file in file manager window, or by using the `open` command in the terminal window.

If you use a Windows operating system, you either can use the Windows OS terminal window or you can download the Unix emulator app `cygwin`. Many people prefer the cygwin and Unix environment to the native Windows environment. You also need to download and install a Python environment with the important packages.

Many hardcore developers, but a minority of students, prefer a pure Unix operating environment such as *Red Hat Linux*. This has the advantage that MacOS and Windows junk does not get in the way as much. For those who otherwise would use Apple systems, the Linux approach is less expensive because it does not require Apple hardware (a Macbook). The disadvantage is that you have to know how to get the other tools you need and there are fewer experts to help you when you get stuck.

I discourage students from using Jupyter notebooks. As with other IDE systems, this seems to make certain basic operations quicker and simpler. But it also implicitly discourages automation and using combinations of tools. For most students, working with collections of files (`modules`) seems to lead to better organization and understanding of your code. I will not accept assignments in the form of Jupyter notebooks.

3 Commands and the environment

sec:Commands

I urge scientific computing students to work at the command line of a terminal window, at least at first, rather than using an integrated IDE. The file structure of the operating system (Windows or Unix like) will help you organize Python modules and run output into directories (folders). Individual specialized tools will allow you to visualize and organize files.

A simple text editor will allow you to create and modify Python modules. I use the "code aware" editor `xcode` on my Mac laptop, but there are other editors for Mac, or Windows or Linux platforms that many people prefer. I might have an editor window open on my desktop for each module I's working on. A single terminal window, set to the correct working directory or folder, lets me run (interpret) a module by typing at the command line prompt: `python` `[`*moduleName*`].py`. The output will be displayed at the command line or put into plots. I look at the output, edit a module, and run again by simply typing ↑ [*enter*] (to re-run the most recent command).

The `python` app is an *interpreter* that *evaluates* ("interprets") Python *commands* in an *environment*. Figure fig:commandLine 1 illustrates the process. It is a "conversation" between me (the user) and the computer at a terminal window. It's worthwhile to follow this, even if you have lots of experience with Python or Matlab or R, especially if you are not used to using the bare command line version. I usually urge people to use *modules* rather than typing directly "at the command line" like this, but you need to understand the command line to understand modules.

The top line started with the general prompt from a terminal window running (for me) the Mac operating system OSX:

```
[JonathansMBP20:~] jg%
```

I typed

`python3` [*return*]

This is a command to the OSX operating system to start the Python interpreter. It printed two lines of information and then the *prompt*

```
>>>
```

I typed a command to the interpreter

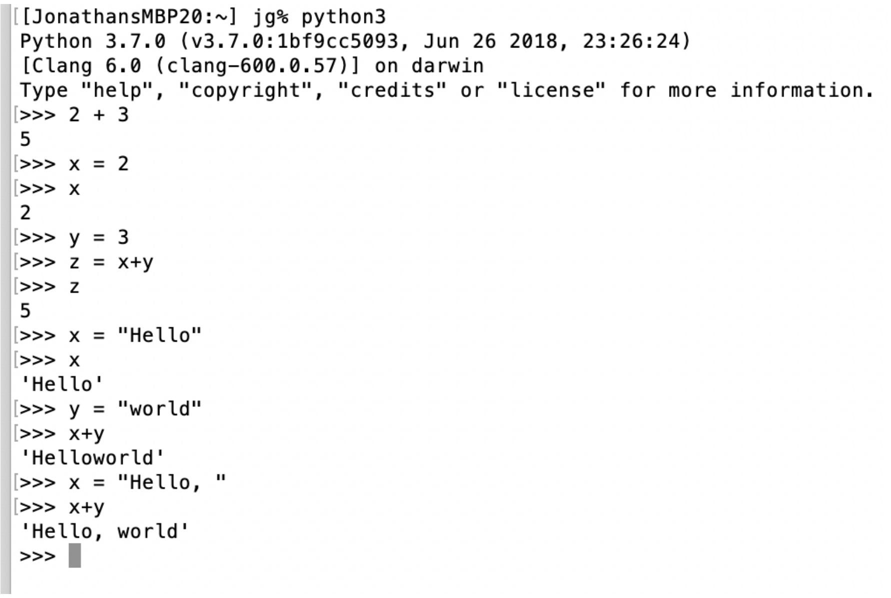
`2 + 3` [*return*]

The interpreter answered: `5`. At the next `>>>` prompt I typed `x = 2`. The interpreter created a *name*, `x`, and *bound* this name to the value `2`. [Names, binding, and values are explained more below.] At the next `>>>` prompt, I typed the expression `x`. The interpreter evaluated that expression by finding the value bound to the name `x`, which is `2`. My next command created a new name, `y` and bound it to the value `3`. After that, `z = x+y` made the interpreter create a new name `z` and bind it to the value it got by evaluating the expression `x+y`, which is `5`.

My next command was

```
x = "Hello"
```

The name `x` did not need to be created, but the interpreter created a new *object* `"Hello"`, which is a *string* made of the *characters* `H`, `e`, etc. [We call numbers *values* and other things *objects*, but the mechanism for handling them is the same.] C++ and Fortran programmers, pay attention to this: unless you tell the interpreter otherwise, a name can be *bound* to an *object* of any *type*. At one point, `x` is bound to a number or a string or any other kind of object. Don't assume `x` "is" (is bound to) a number. The next command bound `y` to the string `"world`. After that, the `+` between these strings *catenates* ("concatenates" would be the correct English word) the strings, putting the second after the first. This illustrates the principle that the meaning of an operation symbol, such as `+`, depends on the types of the objects it "operates" on. The name `z` was bound to this "catenated" string. The command `z` tells the interpreter to print (the print value of the object bound to by) `z`. The result was `’Helloworld’`. This was a bug because I forgot to leave a space. I corrected that by adding a comma followed by a space character to the string `x` and now the concatenated string is correct.



```
[[JonathansMBP20:~] 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.
[>>> 2 + 3
5
[>>> x = 2
[>>> x
2
[>>> y = 3
[>>> z = x+y
[>>> z
5
[>>> x = "Hello"
[>>> x
'Hello'
[>>> y = "world"
[>>> x+y
'Helloworld'
[>>> x = "Hello, "
[>>> x+y
'Hello, world'
>>> 
```

Figure 1: Python commands on the command line.

`fig:commandLine`

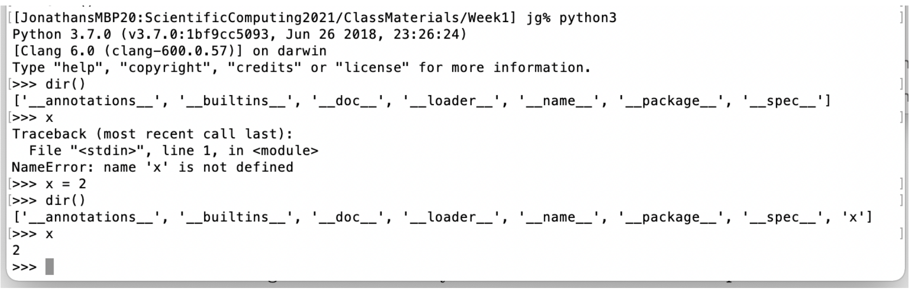
Here is some more detail about what the interpreter does. A command may have effects and *side effects* (effects that are less obvious), some of which change the environment. The environment consists of *names*, which are bound to *objects* (values). The *local namespace* is a list of names that are accessed directly by the interpreter. An *expression* is part of (or all of) a command that can be *evaluated* by the interpreter. Evaluation involves finding the objects (values) that the names refer to (are *bound* to) and doing what the expression says to do with them. This gives the *value* of the expression.

Figure fig:dir 2 is session with the Python interpreter at the command line. The first line started the interpreter, as before. I typed the command `dir()` ("dir" is for "directory" and the parens `()` tell the interpreter that it's a function.) The interpreter returned a list of names in the local namespace, which is

`['__anotations__',` *etc.* `]`

The interpreter puts these names into the local namespace when it starts up. They don't mean much to people who are not Python experts.

Executing a command involves evaluating *expressions*, which consist of names and *operations*. The command `x` causes the interpreter to find the object that the name `x` is bound to and print the value of that object. In this case, the name `x` is not in the local namespace, so the interpreter returns the error message: `Traceback ... name 'x' is not defined`. Then, as in the first session, I typed a command that defined the name `x` and bound it to the value `2`. After this, the `dir()` command shows that the name `x` has been added to the local namespace (the last name in the list of names). The command `x` then "executes" (in interpreted by the interpreter) normally and produces the value `2`.



```
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] 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: Typing at the command window, variables and names

fig:dir

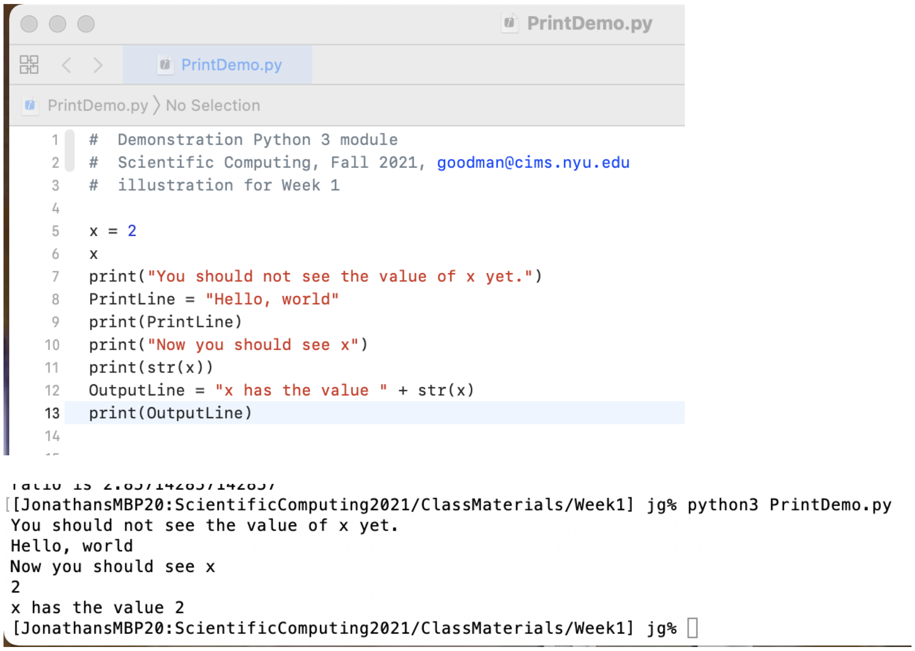
4 The Command Line, Files and Modules

A Python *module* is a file *ModuleName*.py that is a sequence of Python commands. Instead of typing a sequence of commands directly to the Python interpreter, you can put them into a file using a text editor (see below). The command line command: `python [`*`ModuleName`*`].py` has almost the same effect as typing the commands, one by one, to the interpreter. The difference (one of the differences) is that if you make a mistake, you just fix it in the text editor and try again. You don't have to re-type the sequence of commands.

A difference between typing commands one by one and putting them into a module is that a command that consists of an expression does not cause the value of the expression to be sent to the terminal window. The command at line 6 in the module, `x`, does not cause the value bound to `x` to appear at the terminal. Instead, a module can use the print() function. The command print(*StringExpression*) causes the string that is the value of the string expression to be printed at the terminal.

Figure fig:PrintDemo 3 illustrates this. The top of the figure shows a code editor (xcode) with the file PrintDemo.py open in it. This file is a sequence of commands. Lines 1 to 3 are comments. Line 4 is blank, to make the module easier to read. Lines 5 and 6, if typed directly at an interpreter prompt, would have led to the output 2 from the interpreter. Line 7 is a print() command with the expression "You ...". It causes the value of this expression, which is the string, to be printed in the terminal window. Line 11 uses the Python function str() ("str" is for "string"), which returns a string that represents the value of its argument. In this case, the argument in the integer 2. What is returned is the string "2".

The bottom part of Figure fig:PrintDemo 3 shows the command line execution of a Python module. At the (silly long) command line prompt: [Jonathan... jg%, I typed python3 PrintDemo.py. This ran the command python3 with argument PrintDemo.py. The command takes its argument to be a Python module and executes the commands in the module one by one. The results of the print() commands appeared at the terminal as they were executed by the Python interpreter. You can see that the Python command x (line 6 in the module) did not cause anything to be printed at the terminal window. The command at line 11 causes the one character string "2'' to be printed at the terminal. This demonstrates that the command of line 5 was executed by the Python interpreter. You can see that the last line of output: "x has the value 2" might be more helpful to the person running the module than just "2" above it.



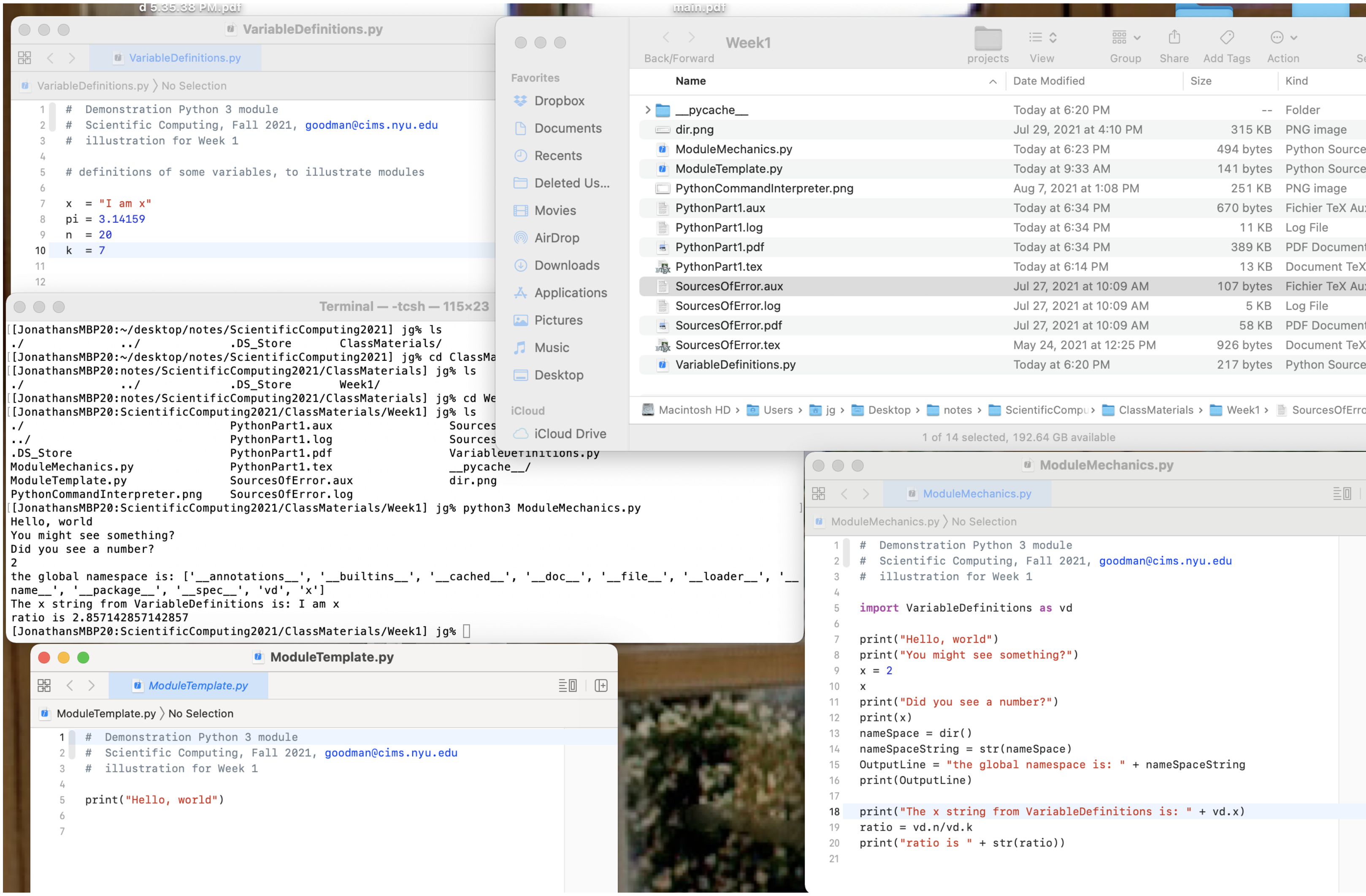
```
#   Demonstration Python 3 module
#   Scientific Computing, Fall 2021, goodman@cims.nyu.edu
#   illustration for Week 1

x = 2
x
print("You should not see the value of x yet.")
PrintLine = "Hello, world"
print(PrintLine)
print("Now you should see x")
print(str(x))
OutputLine = "x has the value " + str(x)
print(OutputLine)

```

```
ratio is 2.857142857142857
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] jg% python3 PrintDemo.py
You should not see the value of x yet.
Hello, world
Now you should see x
2
x has the value 2
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] jg% 
```

Figure 3: A Python module that prints “to” the terminal command window. Top: the module `PrintDemo.py` in a text editor window. Bottom: interpreting this module at the command line. The text editor is “code aware” in that it understands Python well enough to use different fonts and colors for different kinds of code. The comments at the top are dim. The strings are in red.

fig:PrintDemo

Figure 4 illustrates working with a Python “program” consisting of several modules. There are five open windows in this desktop. Three of them are modules, which are described in Section 5. One is a file manager open to this directory. It shows the Python modules and some other files related to the first week of Scientific Computing. One is a terminal window with a command line. This shows that I have “navigated” through the file directory (folder) hierarchy to get to the directory with the code I was working on. The last think in the terminal window is using the Python interpreter to run the module *ModuleMechanics.py*, which is explained in Section 5. The point here is that this setup is as convenient as an IDE in terms of seeing all the code files, but it is more convenient in terms of “knowing where you are” (what directories files are in) and using the same apps (file manager, terminal, text editor) that you use for other purposes. You don’t have to learn a new set of editor instructions for each new language.



d 5.35.38 PM.pdf
main.pdf
VariableDefinitions.py
VariableDefinitions.py ⟩ No Selection

```
#  Demonstration Python 3 module
#  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
#  illustration for Week 1

# definitions of some variables, to illustrate modules

x  = "I am x"
pi = 3.14159
n  = 20
k  = 7

```

Terminal — -tcsh — 115×23

```
[JonathansMBP20:~/desktop/notes/ScientificComputing2021] jg% ls
./               ../              .DS_Store        ClassMaterials/
[JonathansMBP20:~/desktop/notes/ScientificComputing2021] jg% cd ClassMa
[JonathansMBP20:notes/ScientificComputing2021/ClassMaterials] jg% ls
./               ../              .DS_Store        Week1/
[JonathansMBP20:notes/ScientificComputing2021/ClassMaterials] jg% cd We
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] jg% ls
./                               PythonPart1.aux                  Sources
../                              PythonPart1.log                  Sources
.DS_Store                        PythonPart1.pdf                  VariableDefinitions.py
ModuleMechanics.py               PythonPart1.tex                  __pycache__/
ModuleTemplate.py                SourcesOfError.aux               dir.png
PythonCommandInterpreter.png     SourcesOfError.log
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] jg% python3 ModuleMechanics.py
Hello, world
You might see something?
Did you see a number?
2
the global namespace is: ['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__
name__', '__package__', '__spec__', 'vd', 'x']
The x string from VariableDefinitions is: I am x
ratio is 2.857142857142857
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] jg% 
```

Name	Date Modified	Size	Kind
__pycache__	Today at 6:20 PM	--	Folder
dir.png	Jul 29, 2021 at 4:10 PM	315 KB	PNG image
ModuleMechanics.py	Today at 6:23 PM	494 bytes	Python Source
ModuleTemplate.py	Today at 9:33 AM	141 bytes	Python Source
PythonCommandInterpreter.png	Aug 7, 2021 at 1:08 PM	251 KB	PNG image
PythonPart1.aux	Today at 6:34 PM	670 bytes	Fichier TeX Au
PythonPart1.log	Today at 6:34 PM	11 KB	Log File
PythonPart1.pdf	Today at 6:34 PM	389 KB	PDF Documen
PythonPart1.tex	Today at 6:14 PM	13 KB	Document TeX
SourcesOfError.aux	Jul 27, 2021 at 10:09 AM	107 bytes	Fichier TeX Au
SourcesOfError.log	Jul 27, 2021 at 10:09 AM	5 KB	Log File
SourcesOfError.pdf	Jul 27, 2021 at 10:09 AM	58 KB	PDF Documen
SourcesOfError.tex	May 24, 2021 at 12:25 PM	926 bytes	Document TeX
VariableDefinitions.py	Today at 6:20 PM	217 bytes	Python Source

Macintosh HD › Users › jg › Desktop › notes › ScientificCompu › ClassMaterials › Week1 › SourcesOfErro

1 of 14 selected, 192.64 GB available

```
#  Demonstration Python 3 module
#  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
#  illustration for Week 1

print("Hello, world")


```

```
1   #  Demonstration Python 3 module
2   #  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
3   #  illustration for Week 1
4
5   import VariableDefinitions as vd
6
7   print("Hello, world")
8   print("You might see something?")
9   x = 2
10  x
11  print("Did you see a number?")
12  print(x)
13  nameSpace = dir()
14  nameSpaceString = str(nameSpace)
15  OutputLine = "the global namespace is: " + nameSpaceString
16  print(OutputLine)
17
18  print("The x string from VariableDefinitions is: " + vd.x)
19  ratio = vd.n/vd.k
20  print("ratio is " + str(ratio))
21
```

Figure 4: Picture of a desktop illustrating Python coding at the command line. There are three text edit windows (`xcode`), the command line window (`terminal`), and a file manager window (`filemanager`).

`fig:Desktop`

5 Namespaces, Importing, `numpy`

`sec:Namespace`

A *namespace* in Python is something like a directory (folder) in a file system. It is an object that contains a list of names. Each name in a namespace is bound to an object in the same way names in the local namespace (Section 3) are. Look at the `ModuleMechanics.py` window in Figure 4. Line 5 of this module (`import VariableDefinitions as vd`) creates a name `vd` and binds it

`sec:Commands` `fig:Desktop`

to a namespace object. The command: `import [`*module*`] as [`*name*`]` tells the interpreter to execute the commands in the file names `[`*module*`].py`, and to put all the resulting names into the namespace `[`*name*`]`. A namespace name often is a two or three letter abbreviation of the name of the module, as `vd` abbreviates `VariableDefinitions`. The module `VariableDefinitions.py` is open in the top left window of Figure 4. You can see that executing this module will create four names, `x, pi, n, k`. The interpreter puts these names into the namespace `vd`. After this, commands in the module `ModuleMechanics.py` can access these names using `vd.[`*name*`]`. For example, line 18 of `ModuleMechanics.py` has the name `vd.x`, which is the name `x` in the namespace `vd`. Line 7 of the module `VariableDefinitions.py` binds this name to the string `"I am x`. The name `x` in the namespace `vd` and the name `x` in the local namespace are different, and they are bound to different objects. The output line: `The x string from ....` in the terminal window of Figure 4 shows that vd.x is bound to the string `I am x`, not 2. Line 19 of `ModuleMechanics.py` illustrates this again. Line 20 is another example of printing a number with an explanation (`ratio is`) after creating a string to represent the number (`str(ratio)`).

Lines 16 and 17 of `ModuleMechanics.py` illustrate the namespace mechanism. The function `dir()` returns a *list* of all the names in the local namespace. Line 17 creates a printable version of this list. The command `print(dir())` generates an error message, because `dir()` returns a list, not a string. The output from the print command, in the terminal window of Figure 4, shows the names we got in Figure 2, plus the mane `vd`. This is the name that the `import` command added to the local namespace.

The bottom left window of Figure 4 contains a file `ModuleTemplate.py`. This illustrates something I do to create good new modules easily. I make a copy of the "Template" module and rename it. This saves me from having to reproduce the header information (comments at the top). I also could create a new .py file and copy/paste a header from an existing module, but `xcode` makes it harder to create new files than to copy existing ones. A requirement for the Scientific Computing class is that every module have a header that identifies the author, dates the file, states the purpose, and (if necessary) gives more history (created, modified, adapted, ...).

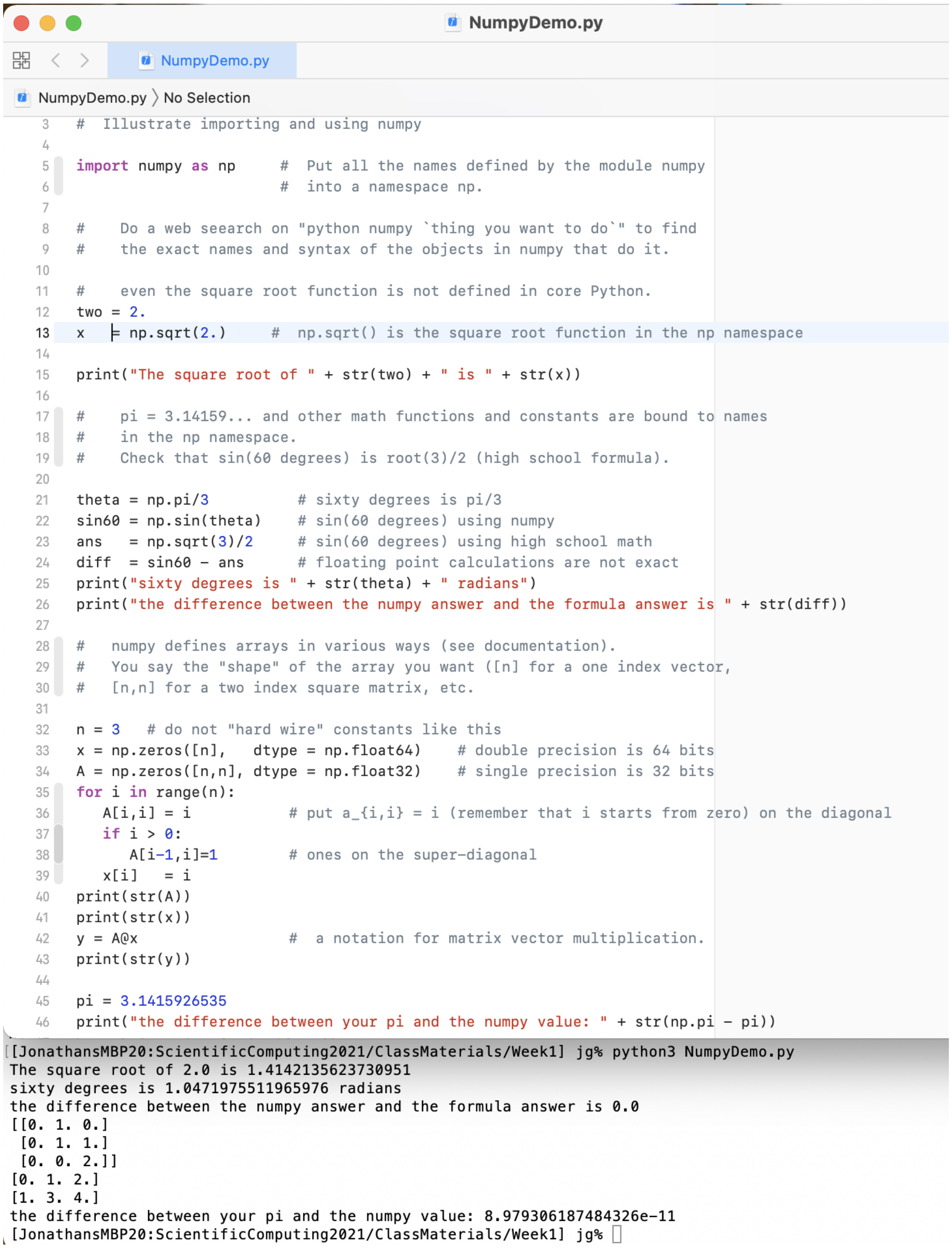
The core Python language lacks many things that are necessary for scientific computing. The module `numpy` supplies many of these. A module or collection of modules supplied by someone else is a *package*. By tradition, you import this into the namespace `np` by putting, early in your module, `import numpy as np`. A web search on *python numpy* will point you to detailed documentation of the *numpy* package.

Figure 5 illustrates the numpy package. Line 5 tells the interpreter to execute the commands in the `numpy.py` module (it knows how to find that module) and put the names created into the namespace `np`. Some of these names are bound to ("point to") objects that are functions, such as the function `sqrt()` in the namespace `np` that is used on line 14. Other names point to objects that are numbers, such as the name `pi` used on line 22. [Hint: Don't type in

$\pi = 3.1415826\ldots$ yourself. The value in `numpy` is more accurate and doesn't have a typo.] `Numpy` also defines objects that act like mathematical vectors and matrices. Line 34 makes `x` point to a vector of length n. The function `zeros()` (in the namespace `np`) creates this vector and sets the values to zero. The part `dtype = np.float64` tells `zeros()` that the entries of `x` are double precision (64 bit) floating point numbers. Core Python has a `list` data type that looks similar to a `numpy` array, but it is very different. Do not use a `list` for a mathematical vector or matrix. This course will explain some of the reasons not to. One is illustrated on line 43: there is a matrix/vector multiply function defined for `numpy` arrays that does not work for core Python lists. Note that the output that uses the simple `str()` function (bottom of Figure fig:Numpy 5) is hard to read.

A *function* (prodecure, subroutine, method) is a type of Python object. A name bound to a function object can be in any namespace. For example, line 23 of Figure fig:Numpy 5 has the name `sin` in the namespace `np`. As a function, it takes arguments, which go in the argument list after the name. The argument list here is `(theta)`, and `theta` is a name bound to a number representing the angle θ. Lines 34 and 35 refer to the function `zeros` in the namespace `np`. The argument list starts with a *shape*, either `[n]` for the vector or `[n,n]` for the matrix. The second argument is a *keyword argument*, which is another name in the `np` namespace. The name `float64` is bound to an object telling numpy to use double precision floating point numbers. A keyword argument takes the form `[`*name*`]=[`*value*`]`. In Line 34, the name is `dtype` (for "data type") and the value is `np.float64` (the value of `float64` in namespace `np`). You can look up more about *positional* arguments (the usual kind) and *keyword* arguments. Routines for making plots typically have a lot of keyword arguments that allow you to fine-tune a plot.

Lines 65 and 66 illustrate that the same name (`pi` in this case) can be in more than one namespace, just as the same filename can appear in more than one directory (folder). When that happens, the names are usually bound to different objects. Line 65 puts the name `pi` into the local namespace and binds it to the value `3.1415926535`. The `numpy` module creates the name `pi` and gives it a more accurate value. Line 66 prints the difference, which is small but much larger than double precision roundoff error.



```python
#  Illustrate importing and using numpy

import numpy as np     #  Put all the names defined by the module numpy
                       #  into a namespace np.

#    Do a web seearch on "python numpy `thing you want to do`" to find
#    the exact names and syntax of the objects in numpy that do it.

#    even the square root function is not defined in core Python.
two = 2.
x   = np.sqrt(2.)      #  np.sqrt() is the square root function in the np namespace

print("The square root of " + str(two) + " is " + str(x))

#    pi = 3.14159... and other math functions and constants are bound to names
#    in the np namespace.
#    Check that sin(60 degrees) is root(3)/2 (high school formula).

theta = np.pi/3          # sixty degrees is pi/3
sin60 = np.sin(theta)    # sin(60 degrees) using numpy
ans   = np.sqrt(3)/2     # sin(60 degrees) using high school math
diff  = sin60 - ans      # floating point calculations are not exact
print("sixty degrees is " + str(theta) + " radians")
print("the difference between the numpy answer and the formula answer is " + str(diff))

#   numpy defines arrays in various ways (see documentation).
#   You say the "shape" of the array you want ([n] for a one index vector,
#   [n,n] for a two index square matrix, etc.

n = 3   # do not "hard wire" constants like this
x = np.zeros([n],   dtype = np.float64)    # double precision is 64 bits
A = np.zeros([n,n], dtype = np.float32)    # single precision is 32 bits
for i in range(n):
   A[i,i] = i           # put a_{i,i} = i (remember that i starts from zero) on the diagonal
   if i > 0:
      A[i-1,i]=1        # ones on the super-diagonal
   x[i]   = i
print(str(A))
print(str(x))
y = A@x                 #  a notation for matrix vector multiplication.
print(str(y))

pi = 3.1415926535
print("the difference between your pi and the numpy value: " + str(np.pi - pi))
```

```
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] jg% python3 NumpyDemo.py
The square root of 2.0 is 1.4142135623730951
sixty degrees is 1.0471975511965976 radians
the difference between the numpy answer and the formula answer is 0.0
[[0. 1. 0.]
 [0. 1. 1.]
 [0. 0. 2.]]
[0. 1. 2.]
[1. 3. 4.]
the difference between your pi and the numpy value: 8.979306187484326e-11
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] jg%
```

Figure 5: Illustration of some features of the numpy package. The top is the module and the bottom is running it at the command line. fig:Numpy

6 Objects, mutable and immutable

The Python relation between *names* and *objects* is different from other interpreted languages such as Matlab and R. Look at Figure fig:NoCopy 6. In Matlab or R, line 12 would copy the entries of vector x into a new vector called y. In Python, line 11 changes the value of x[0] and *also changes the value of* y[0]. This is because x and y are different names bound to the same object. Here is a quick over-simplified explanation. You can find a more accurate version in a fuller discussion of Python.

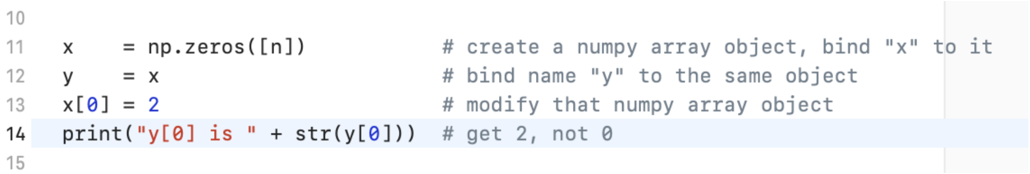
Every *name* in Python is bound to an *object*. Objects contain information, numbers, strings, matrix entries, etc. Names are used to identify objects. A simple assignment *command* ("statement") in Python takes the form: [*name*] = [*expression*]. The interpreter evaluates the expression on the right and produces an object that is (or contains) that value. The assignment part, the = sign, tells the interpreter to bind the name on the left to that object. For example, `greeting = "Hello " + "world"` creates two strings (`"Hello "` and `"world"`) and then creates a new string object by concatenating these. The result is an object that is (contains) the string `"Hello world"`. The name `greeting` is then bound to that object.

Figure fig:NoCopy 6 illustrates the fact that more than one name can be bound to the same object. Each object has its unique object id, which is returned using the `id()` function. Lines 15, 16, and 17 print the ids of the objects pointed to by names x, y and z. The output (command line output at the top of the figure) shows that x and y point to the same object, while z points to a different object. To understand when this can happen, you should know the difference between *mutable* and *immutable* objects in Python.

An object is *mutable* if a command can change it after it has been created. For example, line 37 of Figure fig:Numpy 5 changes the object A by changing the value of the (i,i) entry A_{ii}. It does not create a new object. Line 37 is not a simple assignment command because the left side is more than just a name. It is the name, A, followed by an index reference, [i,i]. Executing this command changes ("mutates"?) the existing object without creating a new one. This also happens in line 11 of Figure fig:NoCopy 6. There, the object point to by x is changed (first entry changed from 0 to 2) but the object itself is not created and keeps its object id. Since the name y is bound to this same object, the value of y[0] continues to be the same as the value of x[0], which is now 2. The immutable types are (mainly) numbers and character strings. The mutable types are (almost) all the more complex data types, including numpy arrays, lists, dictionaries, etc. As a general rule, a simple assignment (such as lines 7 and 9 in Figure fig:NoCopy 6 create new objects while more complex assignments (like line 11, changing one component of the vector), modify an object without creating a new one. If in doubt, try printing the object id numbers.

There are commands to create a new object with the same data as an existing object. The numpy function `array()`, line 10 of Figure fig:NoCopy 6, is an example. It creates a new array of the same size and shape (on column with n entries) as x with the same numbers in the entries. Line 43 of Figure fig:Numpy 5 also creates a new object. It is a simple assignment, though not with a simple data type. Evaluating the right side, which is a numpy array that contains the matrix/vector product `A@x`. The name `y` is bound to this object. The command `x = A@x` evaluate the matrix/vector product and bind the name `x` to the result. The old vector object would be lost.

The rules for copying objects in Python are complex and subtle. At least in the beginning, it will suffice to copy numpy arrays using the `array` function or by first creating the new array (using `zeros()` for example) and then copying the entries. If you want to, you can learn more by exploring the Python `deepcopy` package.



```
x    = np.zeros([n])           # create a numpy array object, bind "x" to it
y    = x                       # bind name "y" to the same object
x[0] = 2                       # modify that numpy array object
print("y[0] is " + str(y[0])) # get 2, not 0
```

Figure 6: Names `x` and `y` are bound to the same object. The command window output is at the top. The object id numbers printed there show that the names tt x and `y` point to (are bound to) the same object and name `z` is bound to a different object.

fig:NoCopy

7 Output formatting

Figure fig:format 7 presents a computational experiment regarding the function $f(x) = e^x - 1$. It shows that the direct formula (line 29) has low relative accuracy when x is small. This is an example of *catastrophic cancellation* discussed in Week 1 (this week). The most interesting part (to me) is left out. It is the fact (see homework exercises for Week 1) that the formula $f(x) = x + \frac{1}{2}x^2 + \frac{1}{6}x^3$ can have better relative accuracy when x is small. In this experiment, we use the Taylor series to $n = 20$ terms as the exact answer. It is accurate to double precision floating point accuracy for the x values in the experiment and it does not suffer from cancellation.

Output formatting is a detail that at first ay seem like a waste of time. It pays off in the long run. Normally, you spend more time looking at output from a code than you spend writing it. Lining up numbers, as in Figure fig:format 7 makes them easier to compare, as in the bottom part of the $f(x)$ column of the output (try to spot and explain the pattern). Figure fig:Numpy 5 has some unformatted Python output. The numbers are easy to make out when there are just a few simple numbers. But unformatted output from more complex computations can be hard to read.

Python has gone through several output formatting systems. Here is a quick partial explanation of one of the less bad ones. The Python documentation has more detail, including all number formats. Lines 39 and 40 of Figure fig:format 7 show the basic idea. You define a string (character string) that contains formatting that will be replaced by the actual numbers. The formatting instructions are in "curlies" (curly braces {}) in the right side of line 39. Line 40 says which numbers are formatted into which place in the string.

The syntax for formatting instructions is

$$\{\texttt{[}\textit{name}\texttt{]:[}\textit{format code}\texttt{]}\}.$$

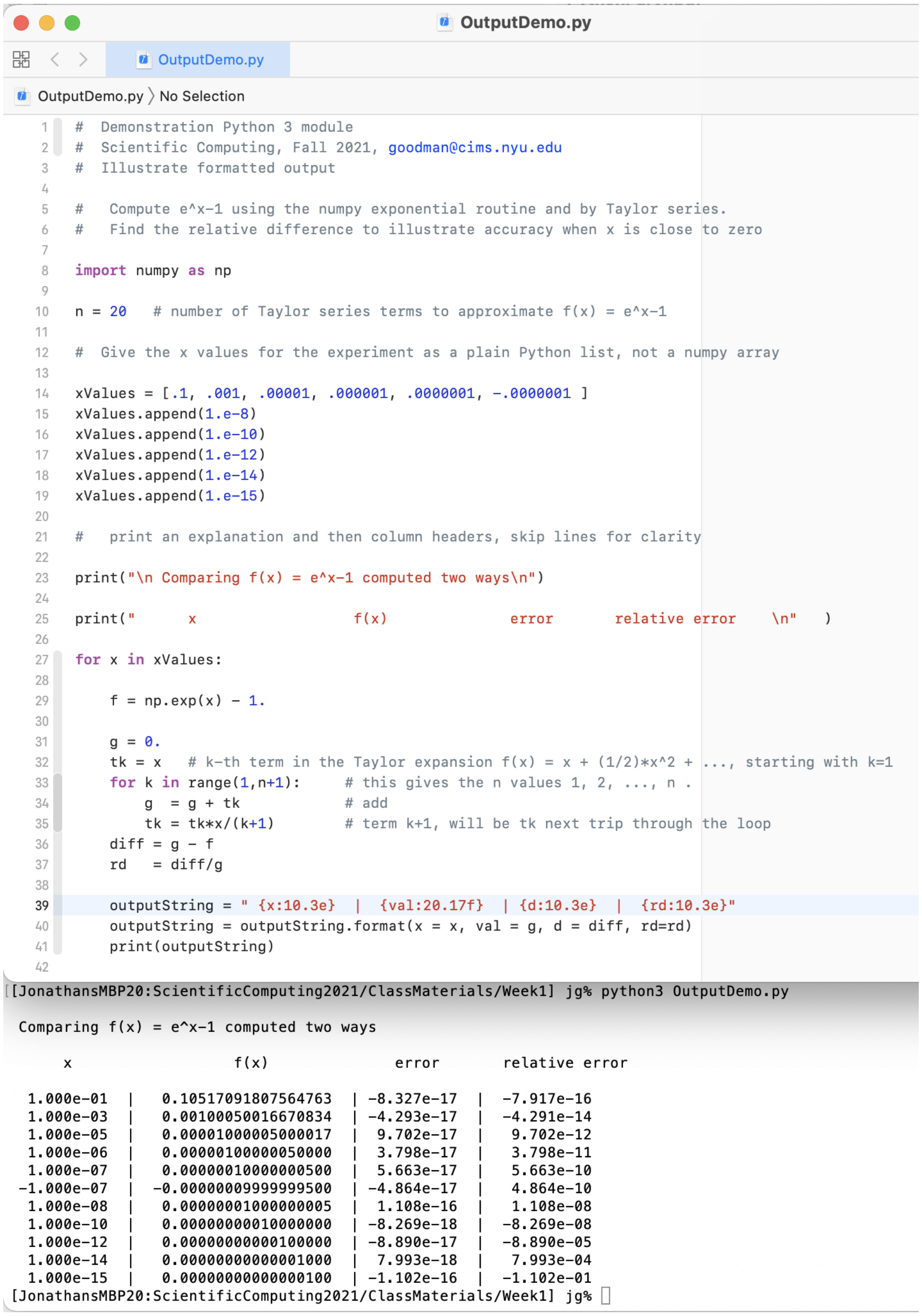
The first one is {`x:10.3e`}. The name is `x`. This will be bound to a floating point number in line 40. The format code is `10.3e`. This is a description of how the floating point number `x` will be represented in the output. The 10 means that the whole thing takes ten characters. These include the possible sign, the digits, and the exponent. The 3 means that there will be three digits after the decimal point. The `e` at the end is for "exponential format". The first number in the `error` column is `-8.327e-17`. This refers to $-8.327 \cdot 10^{-17}$. The three digits after the decimal point are "327". The decimal points line up, which makes it easy to glance down this column and see that the numbers have fluctuating signs and near the 10^{-17} range in magnitude. Characters that are not used by the formatting will be blank and can be used for spacing. All ten characters are used in this case (count them), but if had been {`x:12.3e`} there would have been two blanks.

The right side of line 40 refers to the name `format` in a namespace associated to the string `outputString`. The name `format` is bound to a function that formats its arguments according to the directions in the string. Any object of type `string` comes with a namespace that contains the name `format`. This is part of the definition of the `string` datatype in Python. You can learn more about this by reading about *classes* in Python. For now, it is enough to know that line 40 applies the format function to the string `outputString` using data in the argument list `( x = x, val = g,` $\cdots$`)`. It returns an object that is a string with the format instructions replaced with the formatted numbers. The name `outputString` is bound to this new string. Line 41 prints it, resulting in the data lines in the output table.

The arguments to `format` on line 40 are all *keyword arguments.* The end of Section `sec:Namespace` 5 describes these a little. The keywords go into a namespace for the function call. The first one is `x = x`. In my (computational mathematician) mind, the two `x` names are the same because they refer to the same number. In Python, they are in different namespaces and therefore can refer to different objects. The first `x` is a name in the unnamed namespace for the function `format`. The second `x` is in the local namespace and is bound to the value that the first `x` will be bound to. The second keyword is `val`, and `val = g` binds this name to the value of `g` in the local namespace. Among other things, the this call to the `format` function will format the value of `val` in the argument keyword namespace using the format code `20.17f`. The results are in the second output column.

You choose the format for formatted output to make it easy and quick to learn about the numbers you are printing. Print what you want to know and don't print what you don't want to know. Don't print a lot of digits that don't matter. They just get in the way.

In this example, the $f(x)$ column prints 17 digits (roughly double precision accuracy) to bring out the pattern that is obvious in all but the first few rows. The `error` and `relative error` columns print just three (and even three may be too many) because we just want to know roughly how big these numbers are.



```python
#  Demonstration Python 3 module
#  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
#  Illustrate formatted output

#   Compute e^x-1 using the numpy exponential routine and by Taylor series.
#   Find the relative difference to illustrate accuracy when x is close to zero

import numpy as np

n = 20   # number of Taylor series terms to approximate f(x) = e^x-1

#  Give the x values for the experiment as a plain Python list, not a numpy array

xValues = [.1, .001, .00001, .000001, .0000001, -.0000001 ]
xValues.append(1.e-8)
xValues.append(1.e-10)
xValues.append(1.e-12)
xValues.append(1.e-14)
xValues.append(1.e-15)

#   print an explanation and then column headers, skip lines for clarity

print("\n Comparing f(x) = e^x-1 computed two ways\n")

print("      x                   f(x)               error       relative error      \n"    )

for x in xValues:

    f = np.exp(x) - 1.

    g = 0.
    tk = x   # k-th term in the Taylor expansion f(x) = x + (1/2)*x^2 + ..., starting with k=1
    for k in range(1,n+1):      # this gives the n values 1, 2, ..., n .
        g  = g + tk             # add
        tk = tk*x/(k+1)         # term k+1, will be tk next trip through the loop
    diff = g - f
    rd   = diff/g

    outputString = " {x:10.3e}  |  {val:20.17f}  | {d:10.3e}  |  {rd:10.3e}"
    outputString = outputString.format(x = x, val = g, d = diff, rd=rd)
    print(outputString)
```

```
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] jg% python3 OutputDemo.py

 Comparing f(x) = e^x-1 computed two ways

      x                   f(x)               error       relative error

  1.000e-01  |   0.10517091807564763  | -8.327e-17  |  -7.917e-16
  1.000e-03  |   0.00100050016670834  | -4.293e-17  |  -4.291e-14
  1.000e-05  |   0.00001000005000017  |  9.702e-17  |   9.702e-12
  1.000e-06  |   0.00000100000050000  |  3.798e-17  |   3.798e-11
  1.000e-07  |   0.00000010000000500  |  5.663e-17  |   5.663e-10
 -1.000e-07  |  -0.00000009999999500  | -4.864e-17  |   4.864e-10
  1.000e-08  |   0.00000001000000005  |  1.108e-16  |   1.108e-08
  1.000e-10  |   0.00000000010000000  | -8.269e-18  |  -8.269e-08
  1.000e-12  |   0.00000000000100000  | -8.890e-17  |  -8.890e-05
  1.000e-14  |   0.00000000000001000  |  7.993e-18  |   7.993e-04
  1.000e-15  |   0.00000000000000100  | -1.102e-16  |  -1.102e-01
[JonathansMBP20:ScientificComputing2021/ClassMaterials/Week1] jg% ▯
```

Figure 7: A demonstration of formatted output in a table.

fig:format