

# Programming for Medical Physics

Neil Kirby, Ph.D.

Assistant Professor

The University of Texas Health Science Center San Antonio

*Edited for Python by Noah Bice*

## Lecture 1: Introduction to ~~MATLAB~~ Python

- 1.1 Purpose of the Course
- 1.2 Choice of Python
- 1.3 Getting Started with Python
- 1.4 Python Syntax Overview
- 1.5 Python Prompt
- 1.6 Variables
- 1.7 Built-in Data Types and Functions
- 1.8 Importing Modules
- 1.9 NumPy

### **1.1 Purpose of the Course**

If I were to look back at all the courses that I ever took and decide which was the most useful, it would be the typing class that I took as a sophomore in high school. A significant portion of the work I do on a day-to-day basis involves typing. In the typing class we were taught to use the home keys and to do so without looking down at the keyboard. This has since saved me countless hours compared to my previous hunt-and-peck typing style. The reason though that I can point to this class being my most useful one is that this was the only typing class that I ever took. If I hadn't taken that class in the 10<sup>th</sup> grade, I would still be hunting and pecking. For comparison, I've taken six different courses on quantum mechanics, but only one for the very practical skill of typing. It is for this reason that the class has had such a singular effect on my day-to-day life.

My second most useful class would be the computational physics course I took at UT Austin from Prof. Richard Fitzpatrick, <http://farside.ph.utexas.edu/teaching/329/329.pdf>. The instructor for this class was excellent, but the usefulness of this class happens due to similar reasons as that for the typing class. I program almost on a daily basis and this was the only class I ever took on this topic. Most of the class dealt with intense computational physics problems, such as solving the wave equation and particle-in-cell problems. The truth is that I no longer use programming to solve the same types of problems that I did in the physics course. I do, however, use the techniques I learned in this class daily.

You will not master programming by the end of this course. The only way to become a proficient programmer is by using it frequently to solve problems. Programming is a language, and just like any other language, if you don't use it you will lose it. The purpose of this course is to give you some formal programming experience to start you on the path to becoming a proficient programmer.

## 1.2 Choice of Python

Alan Turing invented an abstract device called a Turing machine. This device consists of three components: a machine head, an infinite tape, and a set of instructions (see Fig. 1). The machine head has various internal states (4 in Fig. 1) and the tape also has different possible states at each position (0 or 1 in example). Based on the combination of the state of the head and tape at that location, the instructions would determine whether the head would change the tape bit, its internal state, and where it moves to on the tape next. Turing was able to prove that such a device could compute any computable sequence. Although this was an abstract device, it ultimately laid the foundation for computer science and the modern computer.

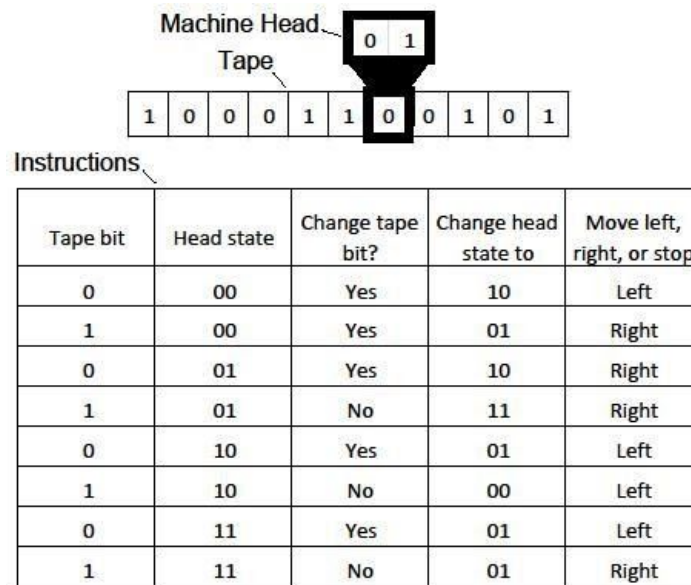


FIG. 1. A Turing machine illustration.

If you ever come to ask me whether you can make a program to perform a specific purpose, the answer will always be yes. Any algorithm that you want to create can be made with any programming language that can code a Turing machine. This means though that we could choose almost any programming language for this course. However, if we were to code Turing machines, we would effectively be writing machine code and it would be a very inefficient use of our time. For this reason, programming languages utilize higher level syntax that is closer to how we write mathematics and our spoken language. This syntax is then converted to machine code for computation.

There are several considerations for choosing the language used for this course. The chosen language must be efficient for scientific computations, supported by a large community of programmers, easy to pick up, and readily able to handle medical physics data (such as DICOM). For these purposes, there are really two choices: Python and MATLAB. MATLAB was originally chosen for this course, but due to popular demand, these notes have been transcribed for Python. Whether you are choosing to complete homework assignments in MATLAB or Python, your experience is going to be mostly the same. The nuances of your chosen language's syntax are only a small part of what makes programming tricky.

By choosing to read this version of the lecture notes, you are probably at least considering using Python for this class. Python and MATLAB both have a lot to offer, but each language has its subtle advantages and disadvantages. Below are some things to consider before committing to a language.

#### MATLAB:

- Of the two languages, MATLAB has the simplest syntax. When you are writing a program, you will likely have to spend less time nitpicking minor issues.
- MATLAB also has paid employees that maintain it. There is thorough documentation and customer service provided by professionals. You can also trust that computations are implemented efficiently.

#### Python:

- Python is a very widely-used, simple, and free programming language. It is currently the most popular language taught in introductory programming courses at universities.
- Many packages will offer the same functionality implemented in different ways. Because you will rely heavily on open-source libraries for many functions, you will have to become familiar with other people's styles of code and documentation.
- Python is widely used in artificial intelligence, robotics, and cybersecurity. Google Brain released TensorFlow1.0.0 in 2017, which has since become the backbone for many higher-level deep learning libraries written in Python. MATLAB also has some deep learning potential, but creative freedoms are sacrificed for ease-of-use, and the machine learning toolbox comes at an extra cost.

### **1.3 Getting Started with Python**

Python is an interpreted language. Your commands are collected in a **script**, which is passed to the Python **interpreter** as a text file with a ".py" extension. The Python interpreter goes through your script and compiles your prompts as machine code, which is then executed. The only thing one needs to run Python scripts is therefore a working Python interpreter and some knowledge about how to communicate with the interpreter, i.e. the Python language.

However, a bare-bones installation of the Python interpreter will likely be unsatisfactory. The Python language itself is stripped of everything except for the essentials. Unlike MATLAB, there is no default variable "pi," there is no default function "cos," and there is no default object "cell." When writing in Python, we often rely heavily on the work of others who have gone through the trouble of making complex objects and functions, like a "pi" constant and a "cos" function. Software developers will create a collection of scripts (called **modules**) and gather them in directories called **packages/libraries** to be distributed. Python allows us to use objects and functions from modules with an **import** statement, but we must first download them and store them in a place that the Python interpreter knows to look for them. For this reason, it is useful to install a Python **package manager**.

Lastly, one complication of learning to write in Python is having to learn the syntax. If you forget to place a colon, parentheses, or some other detail, your Python script will fail to run. Having to read through your code and look for syntax errors can be frustrating, but the burden can be lightened by editing your code in an **Integrated Development Environment (IDE)**. There are several IDEs such as Spyder, Pycharm, IDLE, and Notepad++. Each IDE offers a different layout and tools to edit scripts, and the IDE you use is just a matter of preference. Most IDEs will offer at least color-coated syntax and tools to organize Python projects that span many directories.

The trouble of installing the Python interpreter, a package manager, and an IDE can be handled in one step by downloading and installing **Anaconda**. Anaconda is a Python package manager that allows one to keep many versions of Python with different versions of packages installed at the same time. To install Anaconda, simply navigate to <https://docs.anaconda.com/anaconda/install/>, select your operating system, and follow the prompts. If you install Anaconda in full (as opposed to Miniconda), the Spyder IDE installer and several common Python packages will be downloaded as well. To install Spyder, open the “Anaconda Prompt” application and run the command “conda install -c anaconda spyder.” When you open Spyder, you’ll find something like this.

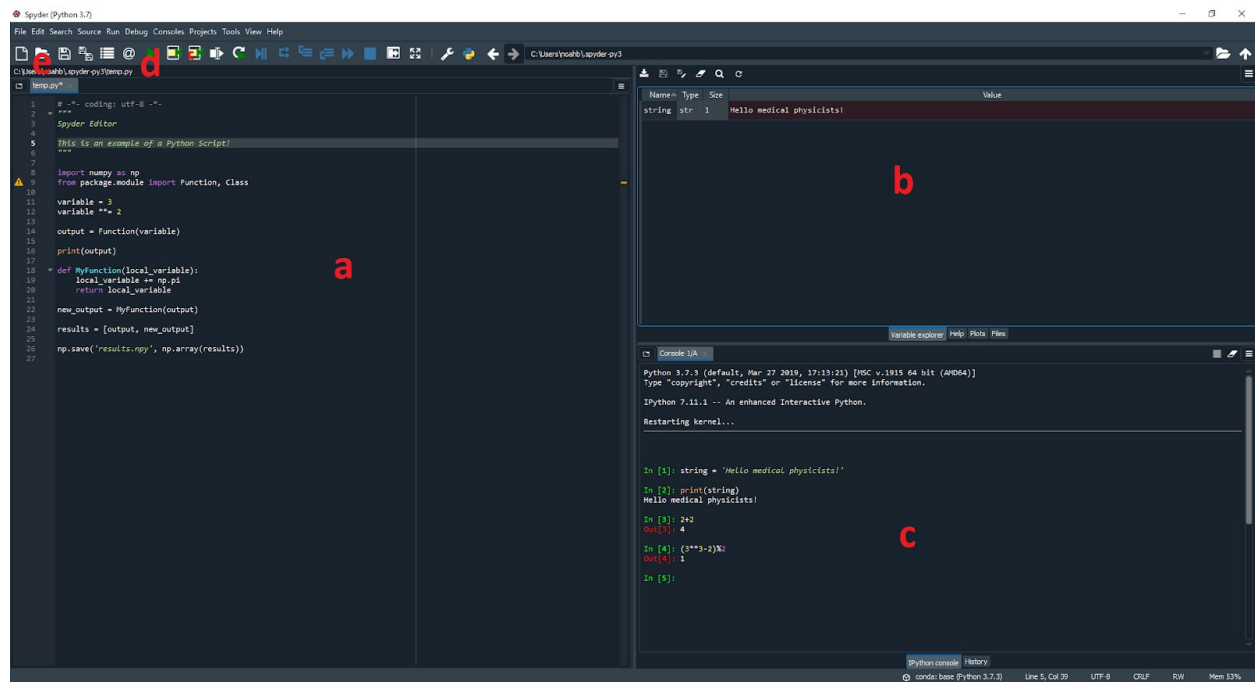


FIG. 2. Spyder IDE.

Important parts of the Spyder application:

- a. Script editor: a space to edit Python scripts
- b. Variable explorer: displays variables in your workspace
- c. Python prompt: useful for quick calculations that don't require rerunning the script
- d. Run: run the selected script
- e. File explorer: navigate to desired files

## 1.4 Python Syntax Overview

The sad fact of the matter is that no matter how elegant of a description of Python syntax I write here, you will not benefit as much as you would by simply opening a Python prompt and screwing around. Python syntax needs to be drilled into your head through repetition. For this purpose, I highly recommend the online exercises at <https://www.codecademy.com/>. This site offers a free interactive course in Python 2, which is not too different from the current Python 3.7. The exercises are sometimes tedious, but you will greatly benefit from the repetitive practice compared to just reading what I've written here. I'll try to supplement their material and expand on parts that I think are especially important to us as medical physicists.

## 1.5 Python Prompt

The Python prompt is a direct interface to the Python programming environment. Your inputs are immediately computed and the output is displayed. To access a Python prompt, either click in the Spyder window (labeled Fig. 2c) or open the application “Anaconda Prompt” and type “python”. The simplest use of this interface is as a calculator. The following basic operations are supported in the Python language:

- Addition (+)
- Subtraction (-)
- Multiplication (\*)
- Division (/)
- Exponentiation (\*\*)
- Modulus (%)

To perform a calculation, simply type an expression in the Python prompt and hit return. These calculations follow the typical rules for order of operations.

```
>>> 2+2
4
>>> (3**3-4)%2
1
```

## 1.6 Variables

In Python, we can assign names to values and store them for manipulation later on. Those stored values are called **variables**. Python and MATLAB are convenient compared to some languages because we don't need to initialize the variable before we use it. We can simply type “variable = value” and use it later on. For example:

```
>>> x = 3
>>> 2+x
5
```

If we want to update the value of a variable, we can do so:

```
>>> x = x+3
>>> x
6
```

Python also accepts the shorthand operators +=, -=, \*=, /=, \*\*=, and %=:

```
>>> y = 3
>>> y += 3
>>> y
6
```

Python will accept operations with multiple variables:

```
>>> x = y = z = 2
```

```
>>> x*y*z
8
```

A variable can be made from almost any string. However, you cannot start a variable with a numbers.

```
>>> x_0 = 1
>>> 0_x = 1
File "<stdin>", line 1
  0_x = 1
  ^
```

Note, we always define the variables and then use them for calculations. If you try to perform an operation with a variable that has not yet been defined, the Python console will throw the following error message:

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

One should be careful when defining names of variables. If you try to give a variable a name that has a special meaning in Python, the interpreter will throw an error.

```
>>> and = 1
File "<stdin>", line 1
  and = 1
  ^
```

In general, clearing variables in a workspace is not as much of an issue in Python as it is in MATLAB, since we are not allowed to use variable names with special meanings. If I am making a program where I need to be efficient with my supply of RAM, I can clear variables as I use them with the `del` statement.

```
>>> x = 1
>>> del x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

When we begin writing our own functions and classes, we should be conscious of the distinction between global and local variables, but we will discuss this later on.

## 1.7 Built-in Data Types and Functions

Python is an object-oriented language. Every time we create a variable in Python it will fall into some category called a "class." Every Python user by default will have access to some standard objects and actions called **classes** and **functions**, respectively. These classes and functions will automatically be understood by the Python interpreter (they do not require `import`). Below I have listed some built-in Python objects and a brief explanation for each of them. You can print a variable's class with the built-in `type()` function.

## 1. Numbers: *int*, *float*

As we have already seen, we are allowed to create numerical variables by simply typing in a number. The default is a signed 32-bit **integer**. A signed integer uses one bit to store whether the number is positive or negative, thus a signed integer can store a number from  $-2^{31}$  to  $2^{31} - 1$ .

```
>>> x = 1
>>> type(x)
<class 'int'>
```

We can also store non-integer numbers as **floats**. Floats are stored in scientific notation. The default number format in Python is a double-precision floating point number, or “double”, which uses 8 bytes (64 bits). One of these bits is used to store the sign of the number, 11 are used for the exponent, and 52 are used for the number.

```
>>> x = 2.718
>>> type(x)
<class 'float'>
```

We can indicate that the variable should be stored as a float by including a period when we declare it.

```
>>> x = 2.
>>> type(x)
<class 'float'>
```

## 2. Strings

Strings are ordered sequences of characters. We can declare a string with single or double quotes.

```
>>> x = 'A '
>>> y = "string."
```

Strings can be compounded and multiplied.

```
>>> x + y
'A string.'
>>> x*3
'A A A '
```

Strings can also be indexed with square brackets and integers. In Python, counting always starts at zero, so if we want the first element of the string, we can get it with `string[0]`, which is also a string.

```
>>> y[0]
's'
```

If we try to access an index beyond the length of the string, Python returns an error.

```
>>> y[7]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

We can get the last item or N-to-last item by using a minus sign before the index.

```
>>> y[-1]
'.'
>>> y[-2]
'g'
```

We can access a range of values in a string by index with the syntax string[first index: last index].

```
>>> y[0:4]
'stri'
```

Notice the last index included is 4-1 = 3. Lastly, we can choose to grab every other character, every third character, or even parse through the string backwards by adding another colon.

```
>>> a = 'This is a long string.'
>>> a[0:11:2]
'Ti sal'
>>> a[::3]
'Tss nsi.'
>>> a[::-1]
'.gnirts gnol a si sihT'
```

Another feature of strings worth noting is that they are not mutable; one cannot overwrite individual characters in a string.

```
>>> a[0] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Some common functions that are useful for manipulating strings are:

- `len()` - returns the length of the string
- `lower()` - returns the input string with all characters lowercase
- `count()` - the number of occurrences of a substring in a string

The syntax for using a function varies depending on whether the function is specific to that class or not. Functions that belong to all objects of one class are called **methods**. Every object which is a string can use the method `lower()` to make its contents lowercase. `count()` can be used in the same way.

```
>>> x.lower()
'a '
>>> a.count('i')
3
```

Functions which are not specific to objects of a certain class, like `len()` and `type()`, can be used as such:

```
>>> len(a)
22
>>> type(a)
```



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

### 3. Lists

Lists are sequences of values which are declared with square brackets. You can keep any collection of data types in a list. You can access items in a list with the same indexing syntax as in strings.

```
>>> my_list = [10, 3.14, 'listeroni pizza']
>>> my_list[0:2]
[10, 3.14]
>>> my_list[-1]
'listeroni pizza'
```

Unlike strings, we can overwrite items in a list by their index.

```
>>> my_list[0] = 'overwritten'
>>> my_list
['overwritten', 3.14, 'listeroni pizza']
```

We can also make lists of lists. This structure is useful for storing image data and tables.

```
>>> x = [[0,1],[2,3]]
>>> x[1][0]
2
>>> z = [[1,2,3]]*3
>>> z
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

Some useful functions for managing lists are:

- `len()` - returns the number of items in a list
- `append()` - adds an item to the end of a list
- `extend()` - append a list of values to the end of a list
- `insert()` - insert a value at an index
- `min()/max()` - returns the minimum or maximum value in a numerical list
- `sum()` - returns the sum of the values in a numerical list

```
>>> x = [1., 2.5, 4.]
>>> x.insert(1, 0.5)
>>> x.append(5)
>>> x
[1.0, 0.5, 2.5, 4.0, 5]
>>> sum(x)
13.0
```

### 4. Dictionaries

The last data type worth mentioning is Python dictionaries. Dictionaries are unordered collections of **keys and values**. Dictionaries are mutable, and can be created with curly brackets with the syntax `dict = {key0: value0, key1: value1,...}`. The values of the dictionary can be accessed with `dict[key]`.

```
>>> khan_final = {'Killian': 95, 'Noah': 90}
```

```
>>> khan_final['Killian']
95
```

Entries can be added to a dictionary by stating a new key and value. Entries can be removed with the `pop()` method.

```
>>> khan_final['Jake'] = 55
>>> khan_final
{'Killian': 95, 'Noah': 90, 'Jake': 55}
>>> khan_final.pop('Noah')
>>> khan_final
{'Killian': 95, 'Jake': 55}
```

Dictionaries are useful for keeping track of data that is inherently paired. We'll also find use for them when we discuss loops.

...

Note that you can change the data type (data-permitting) with the `int()`, `float()`, and `str()` functions.

```
>>> x = 3
>>> type(x)
<class 'int'>
>>> float(x)
3.0
>>> str(x)
'3'
```

Finally, we can `print` a variable's value or any value, so long as the argument of the `print()` function is of a consistent type. Note, since the single quote has a special meaning, we have to use `"\'"` to include it in a string. If we want the print statement to go to a new line, we can use `"\n"`.

```
>>> print('x')
x
>>> print(x)
3
>>> print('The value of \'x\' is ' + str(x))
The value of 'x' is 3
```

or equivalently, using the string's `format()` method

```
>>> print('{} is the value of \'x\''.format(x))
3 is the value of 'x'
>>> print('new \nline')
new
line
```

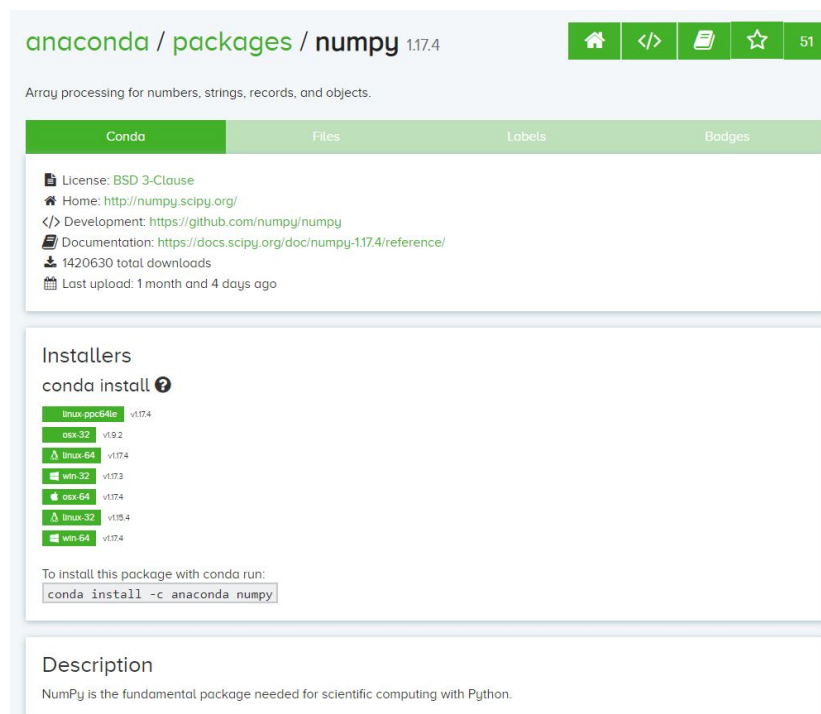
## 1.8 Importing Modules

As previously mentioned, a large fraction of classes and functions that you'll use from day to day in Python will be written by developers on top of the built-in Python classes and functions. When writing Python scripts, we often use their premade, optimized solutions. Every time we want to compute the sine of an expression, we would like to not have to write a function that does so. We stand on the shoulders of great Python libraries like NumPy, Matplotlib, and TensorFlow.

The Python Standard Library (PSL) is a collection of modules with common useful objects and functions. Python installations on most operating systems will automatically include the entire PSL. Some of the packages that I use the most from the PSL are

- **random** - implementations of pseudorandom number generators
- **csv** - comma-separated value file reading and writing
- **os** - miscellaneous operating system interfaces
- **time** - time access and conversions

Beyond the PSL, there are thousands of other useful packages for various purposes. Most Python packages you will need will be either already installed with Anaconda or available to you online through Anaconda. You can install packages easily via the Anaconda Prompt. If you're looking for a new Python package, searching "conda install [package name]" will usually return a webpage (see Fig. 3) with a command that will install the package for you. This page will also show the source (or "channel") from which you are downloading, available operating systems, the package's webpage, development, and documentation.



anaconda / packages / numpy 1.17.4

Array processing for numbers, strings, records, and objects.

Conda Files Labels Badges

License: BSD 3-Clause  
Home: <http://numpy.scipy.org/>  
</> Development: <https://github.com/numpy/numpy>  
Documentation: <https://docs.scipy.org/doc/numpy-1.17.4/reference/>  
1420630 total downloads  
Last upload: 1 month and 4 days ago

Installers

conda install ?

linux-ppc64le v1.17.4  
osx-32 v1.17.2  
linux-64 v1.17.4  
win-32 v1.17.3  
osx-64 v1.17.4  
linux-32 v1.17.4  
win-64 v1.17.4

To install this package with conda run:  
`conda install -c anaconda numpy`

Description

NumPy is the fundamental package needed for scientific computing with Python.

FIG. 3. Installing packages with the Anaconda prompt.

Upon installing a package, directories of Python scripts are usually dumped in `username/anaconda/lib/pythonx.x/site-packages/` (may vary based on choices at installation). When you tell the interpreter `import numpy`, it "looks" for a directory called `numpy`, and `username/anaconda/lib/pythonx.x/site-packages/` is one of the places it "knows" to search.

We make use of premade functions and classes by importing them into our workspace. The general syntax for importing is as follows: `from module.submodule import class/function as variable`. Not all of the components of this import statement need to be present. For example, let's imagine we're sociopaths and we want to use a Hankel function in one of our scripts. After a Google search of "python hankel," we find that SciPy offers an implementation.

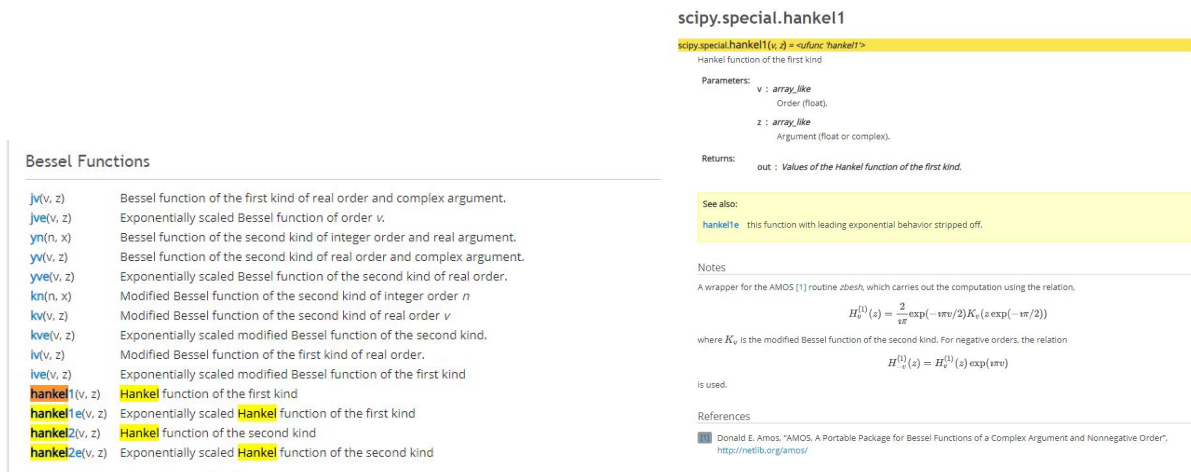


FIG. 4. Example documentation of the `scipy.special` module from `scipy.org`.

The function "`hankel1()`" is in the submodule `special` of the module `scipy`. Supposing we have SciPy installed, we can import `hankel1()` in a few ways.

```
>>> import scipy
>>> import scipy.special as special_function_toolbox
>>> from scipy.special import *
>>> from scipy.special import hankel1 as cursed_bessel
```

The import `*` can be used to import all classes and functions from a module. After the import statements, the Hankel function is in the global namespace and can be accessed anywhere in the script in different ways, depending on how it was imported.

```
>>> v = z = numpy.array([1.,2.,3.])
>>> scipy.special.hankel1(v,z)
array([0.44005059-0.78121282j, 0.35283403-0.6174081j ,
       0.30906272-0.53854162j])
>>> special_function_toolbox.hankel1(v,z)
array([0.44005059-0.78121282j, 0.35283403-0.6174081j ,
       0.30906272-0.53854162j])
>>> hankel1(v,z)
array([0.44005059-0.78121282j, 0.35283403-0.6174081j ,
       0.30906272-0.53854162j])
>>> cursed_bessel(v, z)
array([0.44005059-0.78121282j, 0.35283403-0.6174081j ,
       0.30906272-0.53854162j])
```

Python is a very popular language. Before you break your back creating a function, remember that there is a reasonable probability that someone has already created it and exists in Anaconda, conda-forge (the Anaconda community channel), or on GitHub.

Some notable packages are

- **NumPy** - see section 1.9
- **Matplotlib** - for all of your plotting/display needs
- **SciPy** - lots of useful functions and machine learning algorithms
- **Pandas** - great for managing lots of data
- **PyDicom** - for reading and writing DICOM data
- **TensorFlow** - major deep learning library

## 1.9 NumPy

Importing all of NumPy feels like holding a loaded gun. NumPy is a Python library with efficient implementations of universal functions, matrix operations, and at the center of it all is the NumPy array -- a data structure that is convenient to manipulate and comes with tons of useful methods. You'll find that many other Python packages require an installation of NumPy as a prerequisite, or use NumPy arrays as inputs to their functions.

```
>>> import numpy as np
```

All of the constants and functions that you feel should be built into Python are supplemented by NumPy. For example

```
>>> np.sin(np.pi)
1.2246467991473532e-16
>>> np.sin(np.pi/2)
1.0
>>> np.sqrt(np.e)
1.6487212707001282
>>> np.arcsinh(np.NaN)
nan
>>> np.e**(-np.inf)
0.0
```

The usual operand for NumPy operations is the NumPy array. We can make a NumPy array in various ways, the easiest is just calling **np.array()** on a list, or a list of lists. We can access the array's data type by printing its **dtype attribute**. Another useful attribute is the array's **shape**. The shape is the number of entries along each axis. You can access an object's attributes with the syntax **object.attribute**.

```
>>> a = [-1, 0, 1]
>>> a = np.array(a)
>>> type(a)
<class 'numpy.ndarray'>
>>> b = [[[1.,2.,3.]*3]*3
>>> b
[[[1.0, 2.0, 3.0], [1.0, 2.0, 3.0], [1.0, 2.0, 3.0]], [[1.0, 2.0, 3.0], [1.0, 2.0, 3.0], [1.0, 2.0, 3.0]], [[1.0, 2.0, 3.0], [1.0, 2.0, 3.0], [1.0, 2.0, 3.0]]]
>>> b = np.array(b)
>>> b.dtype
dtype('float64')
>>> b.shape
```

(3, 3, 3)

There are other convenient ways to create NumPy arrays. We can create lists of N integers (from 0 to N-1) with `np.arange()`, we can use the analog of the MATLAB linspace function: `np.linspace()`, and we can initialize arrays with `np.zeros()` and `np.ones()`.

```
>>> np.arange(5)
array([0, 1, 2, 3, 4])
>>> np.linspace(0,1,5)
array([0. , 0.25, 0.5 , 0.75, 1.  ])
>>> np.zeros((3,3))
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
>>> ct = np.ones((512,512,100), dtype='float32')
```

All of the outputs of these functions are NumPy arrays.

A typical patient CT image has 512x512x100 voxels. Much of the work we do in medical physics revolves around this type of data. There are times that I find myself using doubles to store information for each of these voxels when an 8-bit integer would actually work. This equates to using 210 MB instead of 26 MB. If I am using enough of these, my computer might need to use swap space (hard disk serving the purpose of RAM) and drastically slow down. If our CT is stored as a NumPy array, we can exploit many useful methods to make our data more manageable. Some useful methods are `moveaxis()` and `astype()`.

```
>>> ct = ct.moveaxis(2,0)
>>> ct.shape
(100, 512, 512)
>>> ct.dtype
dtype('float32')
>>> ct = ct.astype('float64')
>>> ct.dtype
dtype('float64')
```

NumPy arrays are mutable, and their data is accessible with indexing. For example, if we have a CT volume in the form of a NumPy array with shape (100, 512, 512), we can access the voxel that is

1. in the 50th slice
2. 256th in the 1st direction
3. 256th in the 2nd direction

with

```
>>> ct[49,255,255]
1.0
```

We can change this voxel's value if we wish.

```
>>> ct[49,255,255] = 0.
>>> ct[49,255,255]
0.0
```

Indexing along each axis is the same as list indexing. If we want all of the data along a given axis, we can use “:”. If we want

1. the 50th slice
2. voxels 200 through 300 in the 1st direction
3. all voxels in the 2nd direction

we can use

```
>>> ct[49,199:299,:]  
>>> ct[49,199:299,:].shape  
(100, 512)
```