```
import numpy as np
```

Week 2 Coding Lecture 3: Functions

Throughout this class, and throughout applied mathematics in general, we frequently need to work with mathematical functions like $f(x) = \sin(x)$ or $f(x) = ax^2 + bx + c$. As an example, in the activity this week you will write code that calculates the roots of these functions (i.e., values of x such that $f(x) = 0$). As you might imagine, the formula for the appropriate function will appear in numerous places in your code. Some functions are pre-defined in python, and many more (such as $\sin(x)$) are pre-defined in packages like `numpy`, but other functions (such as $ax^2 + bx + c$) are not. This can be problematic, because it means we have to write out the formula for those functions in many different places. Moreover, if we decide to change the formula then we will have to change many different lines of code, which is one of the most reliable ways to introduce typos and bugs into your code.

We already know a very common solution to this problem in mathematical notation: If we want to work with a complicated formula, we just give it a name. This is what we mean when we write $f(x) = \sin(x)$ or $g(x) = x^3$. The name f or g is a replacement for the full formula.

Ideally, we would like to be able to apply the same idea to code. In particular, it would be nice to say something like `f = np.sin(x)` or `f(x) = np.sin(x)` and then be able to use the variable `f` in place of the formula `sin(x)`. Unfortunately, this exact syntax would not do what we want. The line

```
f = np.sin(x)
```

will either cause an error (because `x` is not defined) or set `f` equal to a number (because python will plug the value of `x` into `sin`). Similarly, the line

```
f(x) = sin(x)
```

```
  File "<ipython-input-55-835c7207fefa>", line 1
    f(x) = sin(x)
    ^
SyntaxError: cannot assign to function call
```

will not work as intended. We will see what the phrase "function call" in the error message means later in these notes.

One thing that will work in this example is

```
f = np.sin
```

This actually does exactly what we want, and from now on we can use `f` in place of `np.sin`. For instance,

```
f(10)
```

```
-0.5440211108893698
```

Unfortunately, this only works because `np.sin` has already been defined, and so it won't help us with more complicated formulas.

Review

Before we talk about how to write a function, let's briefly review what they are and how to use them. Throughout this document, I will refer to functions in the numpy package as "python functions". As you know, in order to use those functions, you have to import the numpy package and then append the name of the function with `np.`. You have already seen quite a few python functions. For example, `sin` and `cos` are functions, as are `append` and `print`. Python functions are essentially just prewritten chunks of code that you can run by using their name. You can tell python to run a function by writing the name (possibly appended with a package name like `np.`) followed by parentheses and a list of some number of input values called "arguments". This process is known as "calling" the function. For example, the function `print` does not need any arguments, so you can call it by typing the name and a set of parentheses:

```
print()
```

(We've never actually used that feature before, but if you don't give an argument to `print`, it just prints a blank line.)

The function `sin` takes one argument (an angle in radians), so you can call it by typing the name and then a number in parentheses:

```
np.sin(4)
```

```
-0.7568024953079282
```

The function `append` takes two arguments - an array and a number - and so you can call it by typing the name `append`, followed by parentheses enclosing an array and a number, each separated by a comma. For instance:

```
y = np.array([1, 2, 3, 4])
np.append(y, 5)
```

```
array([1, 2, 3, 4, 5])
```

Functions also have an output (called a "return value"). You can assign that value to a variable when you call the function. For instance, the return value of `sin` is just the sine of the argument. As we have already seen, you can assign the return value of `sin` to a variable by writing

```
x = np.sin(4)
```

Similarly, the return value of the function `append` is an array with one more entry than the one given as an argument. You can assign this array to a variable like so:

```
z = np.append(y, 5)
```

Some functions, like `print`, aren't really meant to return anything. If you try to use them in an assignment then you will get a value called `None`. We will mostly ignore that feature in this class.

```
x = print("Some text")
```

```
Some text
```

```
print(x)
```

```
None
```

Functions can only ever have one return value, although that value is allowed to be a list or array (or any other type).

Writing your own functions

To define your own functions, you need to specify the function name, any arguments and return values, as well as the code that python will run when the function is called. There are two different ways to write a function in python (well, there are several more related versions, but we will only use these two): Anonymous functions and normal functions. (The word "normal" is not part of the name. Somewhat ironically, python doesn't have a name for non-anonymous functions. If I need to specify non-anonymous functions, I will say so explicitly.) Each of these has slightly different syntax and can be used in slightly different ways.

Anonymous functions

The first type of function we will encounter is called an "anonymous function". You can define it with the following syntax:

```
function_name = lambda list_of_argument_names: one_line_of_code
```

For example, we can define the function $f(x) = 3x + 5$ with the code

```
f = lambda x: 3 * x + 5
```

You can now call this function by writing its name followed by a value of `x` in parentheses. For example,

```
f(4)
```

```
r(4)
```

```
17
```

```
y = f(3)
print(y)
```

```
14
```

```
z = f(3) - f(2) + 3
print(z)
```

```
6
```

There are a few important things to notice about this definition.

1) `f` is now a variable in memory. Its type is `function`, which you can see with the `type` command.

```
print(type(f))
```

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

2) `x` is *not* a variable in memory. (Actually, this may or may not be true for you, because you might have already defined `x` somewhere else, but `x` does not have to be a variable. You can prove this to yourself by starting a blank script or a new console session, defining `f` as above and then trying to print the value of `x`.) This is very important - arguments like `x` can be used in the code for a function even though they are not already defined.

3) When you call a function, the argument you give in parentheses is used as the value of `x`. For instance, if you write `f(4)`, then python will use 4 as the value of `x`.

4) The return value is whatever the one line of code evaluates to. If you write `f(4)`, then the return value is `3 * 4 + 5`, which is `17`.

It is also worth noting that the `function_name` is optional (which is why these are called "anonymous"). It will occasionally be useful to create a function without actually naming it, such as

```
print(lambda x: 3 * x + 5)
```

```
<function <lambda> at 0x7efca01e0310>
```

(Don't worry about the numer that got printed out. It is irrelevant for our class.)

The behavior of the argument `x` might seem particularly strange. As we already saw, neither defining nor calling this function defines a variable named `x`. What's more, if you try this code:

```
x = 2
f = lambda x: 3 * x + 5
y = f(10)
print(x)
```

```
2
```

you can see that defining/calling `f` does not affect any other variables named `x`. This is because the arguments of the function have something called *local scope*. The variable `x` from the definition of `x` can only be seen by python while python is running `f`, and this `x` will not interfere with any variables that aren't in the function body. In essence, python is giving `f` its own private section of memory whenever you call it. For instance, when python executes the line `y = f(10)` above, it will make a private section of memory for `f` with a variable named `x` that has the value `10`. This private memory is completely separate from the memory for the rest of your script.

Other variables you define in your script (including the variable `f` itself) live in something called the *global scope*. As we have already seen, any code in your script can access and modify any other variables in the global scope. (As a side note, in python "global scope" is really restricted to a single file. If you write another script in another file, the two files cannot share variables. I mention this because it is not true in some other languages like MATLAB. If you run one MATLAB script and then another, the second script can both access and modify the variables created by the first script.)

As a general rule, functions can see/access variables from the global scope, but they cannot modify those variables. As an example,

```
a = 2
f = lambda x: a * x
```

```
print(f(3))
```

```
6
```

The function `f` can access the variable `a`, and so we are allowed to use `a` in the function definition, even though it is not one of the arguments. It is important to note that `f` looks up the value of `a` every time you call it. This means that if `a` changes, then the result of future calls of `f` will also change.

```
a = 2
f = lambda x: a * x
print(f(3))
a = 10
print(f(3))
```

```
6
30
```

This behavior is different than that of some other languages like MATLAB, where the value of `a` is essentially "locked in" when the function is defined.

It's also worth noting that the name of the argument does not matter, as long as you are consistent. That is, if you use the name `x` in the argument list then you also need to use the name `x` when you refer to that argument in the function body. For example, the following are exactly equivalent:

```
f = lambda x: 3 * x + 5
f = lambda z: 3 * z + 5
f = lambda this_is_a_silly_variable_name123: 3 * this_is_a_silly_variable_name123 + 5
```

However, the line

```
f = lambda x: 3 * y + 5
```

will not do what you want. Python will happily define the function `f` without checking the value of `y`. When try to call `f`, python will check the global scope for a variable named `y`. If it exists, python will use that value in the expression `3 * y + 5`, but if it does not exist then python will throw an error. It is almost always a mistake to write something like this.

It is also worth noting that you aren't restricted to just using hard coded numbers as arguments when you call a function. Anything that evaluates to a variable of the appropriate type (in most of our cases, the appropriate type is any numeric type like a `float` or an `int`) is fine. For instance,

```
f = lambda x: 3 * x + 5
x = 4
y = np.pi
print(f(x))
```

```
17
```

```
print(f(y))
```

```
14.42477796076938
```

```
print(f(x - 2 * y))
```

```
-1.8495559215387587
```

Again, notice that there is no confusion with variable names, even though we used `x` in the definition of `f` and also used it when calling the function. The `x` you defined outside the function is completely separate from the argument in your function definition. You can use this syntax to define functions with any number of arguments. For instance,

```
f = lambda x, y: x ** 2 + y
print(f(2, 3))
```

```
7
```

```
print(f(3, 2))
```

```
11
```

```
g = lambda a, b, c, x, y, z: a + 2 * b - c + x ** 2 - y ** 2 + 3 * z ** 3
print(g(1, 2, 3, 4, 5, 6))
```

```
641
```

```
h = lambda : 2 * np.pi
h()
```

```
6.283185307179586
```

Notice that the order of the argument matters. In the above example, the first argument of `f` is used as `x` and the second argument is used as `y`. That's why `f(2, 3)` and `f(3, 2)` are not the same.

Non-anonymous functions (aka "functions")

Anonymous functions are useful, but quite limited. The biggest problem is that they are limited to one line of code. (More specifically, they are limited to a single expression. This distinction doesn't really matter to us, but it's worth noting that an assignment like `x = 3` is not an expression, so you cannot use it in an anonymous function.) In particular, you cannot include an `if` statement or a loop inside an anonymous function, because those require more than one line of code. If you want to use a more complicated function in one of your scripts, you can write a normal function.

Function definitions have the following syntax:

`def function_name(list_of_arguments):

```
# As much code as you want
return return_value`
```

As an example, we can make a function for the formula $y = 2x^3 + 11$ with the code

```
def my_first_function(x):
    y = 2 * x ** 3 + 11
    return y
```

You can now call this function from anywhere in your script *after* the function definition by using

```
my_first_function(3)
```

```
65
```

The words `def` and `return` are keywords in python. The function definition must always start with a `def` and end with a `:`. The `function_name` can be any legal python name (letters, numbers and underscores are allowed and you have to start with a letter or underscore). All of the code in the function body, including the line with `return`, must be indented, just like in a loop or conditional statement.

The `list_of_argument_names` works exactly the same as in an anonymous function. We have only used one variable here (`x`), but we can list as many as we want and separate them with commas. Just like with anonymous functions, these arguments exist in their own local scope (i.e., a private section of memory) and cannot be accessed or modified by any code outside of the function.

The `return_value` is an expression that evaluates to whatever you want the function to return. It is common to assign your return value to a variable and then use that variable name as the `return_value`, but you can put any expression you want after the return. For instance, we could rewrite the same function as

```
def my_first_function(x):
    return 2 * x ** 3 + 11

print(my_first_function(3))
```

```
65
```

You are technically allowed to put code in lines after the `return`, but python will not run any code in a function after it encounters a `return`.

Let's try a more complicated example. In particular, let's make a function that will return the absolute value of its argument. Of course,

```
def my_abs(x):
    if x >= 0:
        abs_x = x
    else:
        abs_x = -x
    return abs_x
```

Now that this function is defined, we can call it as usual. For example,

```
print(my_abs(-12))
```

```
12
```

```
print(my_abs(5))
```

```
5
```

As another example, let's make a function that will return the first N Fibonacci numbers. We will use three arguments: `F1` and `F2` will be the first two Fibonacci numbers (these are usually both 1, but they don't have to be) and `N` will be the total number of values that we want to calculate. The output will be a 1D array with the first N Fibonacci numbers.

```
def first_N_fib(F1, F2, N):
    fib = np.zeros(N)
    fib[0] = F1
    fib[1] = F2

    for n in range(2, N):
        fib[n] = fib[n - 1] + fib[n - 2]
    return fib
```

Now that this function is defined, if we want to calculate the first ten Fibonacci numbers we can just use

```
print(first_N_fib(1, 1, 10))
```

```
[ 1.  1.  2.  3.  5.  8. 13. 21. 34. 55.]
```

Importing Functions

Very often you will want to write a function once and then use it in many different scripts. For example, in this week's activity you will write a function to calculate the roots of an arbitrary function. This is the sort of algorithm that might come up in many different applications. It would be tedious (and error prone) to rewrite this function definition in every script that needed to find the root of a function. Fortunately, there is a way to get function definitions from a different file into your script. For example, suppose you had a file called `fibonacci.py` that contained the above function definition for `first_N_fib`. You could use the code

```
import fibonacci
```

to gain access to this function. From then on, you could use the code

```
fibonacci.first_N_fib(1, 1, 10)
```

to calculate the first 10 Fibonacci numbers.

It's important to note that the `import` statement runs the entire file you are importing. In this case, the statement `import fibonacci` runs the entire file `fibonacci.py`. If `fibonacci.py` is nothing but function definitions, then this is probably exactly what you want, but if `fibonacci` is littered with `print` statements or other code, those will also be run.

We won't import our own functions very often in this class. For one thing, most of our code is small enough and self-contained enough for such a method to be unnecessary. For another thing, Gradescope does not allow uploading multiple files.

Shadowing

Now that you can define your own functions, it is possible for you to define a function with the same name as a builtin python function. As a silly example, you could define your own function `print` with the code

```
def print(x):
    return x ** 2
```

From now on, if you try to use the `print` function, python will find this version instead of the builtin version. This can lead to problems:

```
print(3)
```

```
    9
```

```
print("This is some text")
```

```
    ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)
    <ipython-input-93-c8f8ec63071e> in <module>
    ----> 1 print("This is some text")

    <ipython-input-91-caefd7b8c5ce> in print(x)
          1 def print(x):
    ----> 2     return x ** 2

    TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'

    SEARCH STACK OVERFLOW
```

We say that we have "shadowed" the original function. It still exists, but we cannot access it because we took the name `print` and used it for something else. (There isn't really anything special about defining a function here; if we had made any variable with the name `print`, it would have shadowed the builtin function.) In a script, you should change the name of your function if this comes up. If you accidentally do this in a console or Jupyter notebook, you can get rid of your version with the command

```
del print
```

Now we can use the original `print` command as normal.

```
print("This is some text")
```

```
This is some text
```