Week 2: more fun with code!

Amath 301

TA Session

Today:

0. Q&A - What do you want to know?
1. Write your own functions!
2. Multiple return
3. Basic design patterns

Why define your own functions?

1. Simplicity. Write it once, call it `n` times
2. Only one instance to debug (copy-paste bad!)
3. Modularity. Can export from a module (python) or locally (Matlab) for use in another code
4. Make big programs by combining little function!
5. Can handle multiple inputs/outputs

User defined functions (syntax)

Task: take in a number and add 1 to it, return the new value.

Python:

```python
f = lambda x: x + 1  # "lambda function"

def addone(x):  # standard definition
    return x + 1
```

Matlab:

```matlab
f = @(x) x + 1  % "anonymous function"

function x = addone(x)
    % standard definition
    x = x + 1;
end
```

Python function anatomy

```
def function_name(argument):
    """ The 'Document string'
    A short description of the function
    and any details about it for the user. """
    # The body of the function follows
    new_variable = argument_1 + argument_1
    # return keyword specifies the output
    return new_variable
```

new: `def` and `return` keywords.

The variables following the `return` keyword are returned to the user when the statement is reached.

Python function anatomy

```python
def compute_pairwise_sums(arg1, arg2, arg3):
    """ This function computes pairwise sums of 3 numbers.
    arg1: number
    arg2: number
    arg3: number
    """
    a = arg1 + arg2
    b = arg2 + arg3
    c = arg1 + arg3
    # return keyword specifies the output
    return a, b, c # python supports multiple returns
    a = 0  # This code is never executed
```

Call this with:

```python
a, b, c = compute_pairwise_sums(1,2,3)
```

Matlab function anatomy

```
function x = function_name(arg1, arg2)
    % Provide documentation here
    % Use a % at the start of each line
    x = arg1 + arg2;
    return
    x = 0  % never reached
end
```

The keyword `return` functions differently here.

Keyword `function` starts the function, `end` declares the end of the block. The returned value is whatever the specified variable is at the end of the block.

All matlab functions go at the end of the file. Python functions can go **anywhere** in the function (as long as they are defined before they are called).

Matlab function anatomy

```
function [a,b,c] = add_pairwise_sums(arg1, arg2, arg3)
    % Compute pairwise sums
    % arg1, arg2, arg3 are numbers
    a = arg1 + arg2;
    b = arg2 + arg3;
    c = arg1 + arg3;
end
```

Call this with

```
[a,b,c] = add_pairwise_sums(1,2,3)
```

Note the extra `[ ]` when compared to Python multiple returns.

A note of caution

What does

```
a = add_pairwise_sums(1,2,3)
```

return in python? Matlab?

Choices:

1. all three variables stored in a tuple / matrix `a`
2. Just the first returned variable (`a = 2`)
3. Syntax error

A note of caution

What does

```
a = add_pairwise_sums(1,2,3)
```

return in python? Matlab?

Choices:

1. (**Python**) all three variables stored in a tuple ~~/ matrix~~ `a = (3,5,4)`.
2. (**Matlab**) Just the first returned variable (`a = 2`)
3. ~~Syntax error~~

Unpack a tuple in **Python** by assignment:

`a1, a2, a3 = (3,5,4)` gives `a1=3`, `a2=5`, `a3=4`

Function best practices

1. One "idea" per function. Each function should solve exactly one task
2. Functions can/should call other functions!
3. Shorter functions are better. <15 lines of code is preferred.
4. Short, descriptive function names trump long and technical names:
   - `AddOne` > `AddTwoNumbersTogetherAndReturnTheSum`
5. Use one convention for names: snake_case **or** camelCase:
   - `sake_case_function_name` **or** `camelCaseFunctionName`

First character should be lowercase, `CamelCase` is reserved for classes in camel case.

code skeletons for your toolbox

(Python) Update until convergence

```
xnew = ...  # initial guess
while True:
    xold = xnew  # Move data
    xnew = update(xold)  # perform some update step
    if check_stop(xnew, xold, eps):  # check a stopping condition
        break  # exit the loop
```

Pros:

- Simple. Modularize your update step into `update` and check your stopping condition in `check_stop`.

Cons:

- Need more code to store all iteration
- may infinite loop

(Matlab) Update until convergence

```
xnew = ...  % initial guess
while true
    xold = xnew  % Move data
    xnew = update(xold)  % perform some update step
    if check_stop(xnew, xold, eps)  % check a stopping condition
        break  % exit the loop
    end
end
```

Pros:

- Simple. Modularize your update step into `update` and check your stopping condition in `check_stop`.

Cons:

- Need more code to store all iteration
- may infinite loop

code skeletons for your toolbox

(Python) Update until convergence - For loop variant

```python
xnew = ...  # initial guess
for index in range(max_number_of_iterations):  # change loop iterator
    xold = xnew  # Move data
    xnew = update(xold)  # perform some update step
    if check_stop(xnew, xold, eps):  # check a stopping condition
        break  # exit the loop
```

Break the loop after a maximum number of iterations, avoids infinite loop. May not converge if `max_number_of_iterations` is too small.

code skeletons for your toolbox

(matlab) Update until convergence - For loop variant

```
xnew = ...  % initial guess
for index in 1:max_number_of_iterations  % change loop iterator here
    xold = xnew  % Move data
    xnew = update(xold)  % perform some update step
    if check_stop(xnew, xold, eps):  % check a stopping condition
        break  % exit the loop
    end
end
```

Break the loop after a maximum number of iterations, avoids infinite loop. May not converge if `max_number_of_iterations` is too small.