Computer Methods (MAE 3403)

Chapter 1

Functions, Pseudo code, Good practice

Functions

- Programming requires repeating a set of tasks
  - Compute sin(x): math.sin(x) is a set of mathematical operations that approximately compute sin(x)

- Store a sequence of instructions as a function that can be called over and over again

- Most powerful use of computer programming: writing your own functions

Basics

- Function: a sequence of instructions that performs a specific task, a block of code that runs when called
- can have input arguments, output parameters
  - math.sin(x)
- Sequence of instructions: body of the function

Built-in functions

- type(len)

- Functions from some packages/modules
  - **import numpy as np**
    type(np.linspace)
  - math.sin, np.array

Define your own function

- Most common way: specify a function via *def*

header → **def** function_name(argument_1, argument_2, ...):

```
    '''descriptive string '''
    # comments about the statements
    function_statements
    return output_parameters (optional)
```

body → (descriptive string … return)

- Optional but customary: description of the function
- Commenting frequently

Example

```
def my_adder(a, b, c):
    """
    function to sum the 3 numbers
    Input: 3 numbers a, b, c
    Output: the sum of a, b, and c
    author: date:
    """

    # this is the summation
    out = a + b + c
    return out
```

- Indent your code (4 spaces = one level of indentation)
- block indentation: select + Tab (Shift + Tab)
- Avoid extra lines of code

What happens when calling a function

- d = my_adder(1, 2, 3)
- Assignment operator works from right to left.

1. Python finds the function *my_adder*.
2. *my_adder* takes the first input argument value 1 and assigns it to the variable with name *a* (first variable name in input argument list).
3. *Repeat the process to assign 2 and 3 to b and c in the function*, *respectively*.
4. *my_adder* computes the sum of *a*, *b*, and *c*, which is 1 + 2 + 3 = 6.
5. *my_adder* assigns the value 6 to the variable *out*.
6. *my_adder* outputs the value contained in the output variable *out*, which is 6.
7. *my_adder(1,2,3)* is equivalent to the value 6, and this value is assigned to the variable with name *d*.

Notes

- Pay attention to the data types of the input arguments
  - help(my_adder)
- Read the errors that Python gives you: usually tells you where the problem was.
- You can assign function calls and mathematical expressions as inputs
  - d = my_adder(np.sin(np.pi), np.cos(np.pi), np.tan(np.pi))
  - d = my_adder(5 + 2, 3 * 4, 12 / 6)

Multiple output parameters

- Separate the output parameters by commas
- Output returned as a tuple, unpack the returned tuple

```
def my_trig_sum(a, b):
    """
    """
    out1 = np.sin(a) + np.cos(b)
    out2 = np.sin(b) + np.cos(a)
    return out1, out2, [out1, out2]
c, d, e = my_trig_sum(2, 3)
print(f"c ={c}, d={d}, e={e}")
```

c =-0.0806950697747637, d=-0.2750268284872752, e=[-0.0806950697747637, -0.2750268284872752]

unpack (← `c, d, e = my_trig_sum(2, 3)`)

Default values

```
def print_greeting(day = 'Monday', name = 'Qingkai’):
    print(f'Greetings, {name}, today is {day}’)

print_greeting()
print_greeting(name = 'Timmy', day = 'Friday’)
print_greeting(name = 'Alex')
```

Number of input arguments

- Positional vs. excess parameters
- def func(x1,x2,*x3):
- If calling this function with func(a,b,c,d,e), which arguments are positional and excess respectively?

Variable scope

- A function has its own memory block reserved for variables created within that function.
- A variable with a given name can be assigned within a function without changing a variable with the same name outside the function.

```
def my_adder(a, b, c):
    out = a + b + c
    print(f'The value out within the function is {out}')
    return out

out = 1
d = my_adder(1, 2, 3)
print(f'The value out outside the function is {out}')
```

Example: intentionally confusing

```
def my_test(a, b):
    x = a + b
    y = x * b
    z = a + b
    m = 2
    print(f'Within function: x={x}, y={y}, z={z}')
    return x, y
a = 2
b = 3
z = 1
y, x = my_test(b, a)
print(f'Outside function: x={x}, y={y}, z={z}')
```

- What will the values of a, b, x, y, m, and z be after the code is run?

Mutable input argument

- If a mutable object, such as a list, is passed as input and modified in a function, the change will stay with the object.

```
def squares(a):
    for i in range(len(a)):
        a[i] = a[i]**2

a = [1, 2, 3, 4]
squares(a)
print(a) # ’a’ now contains ’a**2’
```

Lambda statement (function)

- Typically for one line function
- Defined using the *lambda* keyword

  lambda arguments: expression

- square = **lambda** x: x**2
  my_adder = **lambda** x, y: x + y
- Simplify code

Functions as arguments to functions

- Sometimes it is useful to pass a function as a variable to another function.

```
import numpy as np
def my_fun_plus_one(f, x):
    return f(x) + 1
print(my_fun_plus_one(np.sin, np.pi/2))
print(my_fun_plus_one(np.cos, np.pi/2))
print(my_fun_plus_one(np.sqrt, 25))
print(my_fun_plus_one(lambda x: x + 2, 2))
```

Nested functions

```
import numpy as np
def my_dist_xyz(x, y, z):
""" x, y, z are 2D coordinates contained in a tuple output: d -
list, where d[0] is the distance between x and y d[1] is the
distance between x and z d[2] is the distance between y and z
"""
    def my_dist(x, y):
""" subfunction for my_dist_xyz Output is the distance between
x and y, computed using the distance formula """
        out = np.sqrt((x[0]-y[0])**2+(x[1]-y[1])**2)
        return out
    d0 = my_dist(x, y)
    d1 = my_dist(x, z)
    d2 = my_dist(y, z)
    return [d0, d1, d2]
```

- my_dist defined in my_dist_xyz (parent function): separate memory block

Modules

- Store useful and related functions in modules
- A module is a file where the functions reside
- A module can be loaded as

  from module name import *

- Or a specific function from the module can be loaded:

  from module name import func name

- Modules can have alias

  import math as m

Related modules

- math (cmath) module: most mathematical functions
- Different modules may have different definitions of the same function: *sin* is available from math, cmath and numpy.
- Import selected functions:

  from math import log, sin, . . .
- Or import math, then use math.log, math.sin

numpy module

- Must be installed separately!
- Introduces “array” (can be used to represent matrix)

```
>>> from numpy import array
>>> a = array([[2.0, -1.0],[-1.0, 3.0]])
>>> print(a) [[ 2. -1.]
[-1. 3.]]
>>> b = array([[2, -1],[-1, 3]],float)
>>> print(b) [[ 2. -1.]
[-1. 3.]]
```

zeros((dim1,dim2),type)
ones((dim1,dim2),type)

Plotting

- matplotlib.pyplot is a collection of 2D plotting functions similar to MATALB style functionalities.
- Require separate installation
- Will discuss it separately.

Good practices

- Errors are unavoidable: can be frustrating
- What types of errors?
- Good practices reduce the chance of error happening
- Debugging tools

Error types

- Syntax errors: incorrect syntax and Python cannot understand, e.g., 1 = x, (1], if True, …
  - typically Python will return an error and point out where the error occurred (most of the times)
- Exceptions/runtime errors: errors that occur during execution, may not be fatal
  - 1/0 (ZeroDivisionError), print(a) (NameError), x = [2], x+2 (TypeError) …
  - Run a program multiple times, different settings, etc.

Logic error

- Code runs but does not produce expected solution

```
def my_bad_factorial(n):
    out = 0
    for i in range(1, n+1):
        out = out*i
    return out
```

- Easy to generate but hard to find:
  - meticulously go through each line of your code.
  - No assumptions.
  - Use Python Debugger

Avoid errors

- Start with an outline of your program (pseudo code)
  - Address all the tasks
  - In the order in which it should perform them
- Time spent planning is time well spent
  - Do not rush to programming without planning out tasks
- Design your program in terms of modules/functions that accomplish a small well-defined task and know as little information of other function as possible

Test everything often

- Test modules/functions using test cases (including corner cases) for which you know the answers.
  - Prime number: test with 0 (corner case), 1 (corner case), 2 (simple case), 97 (complicated case), etc.
  - Build your confidence.
  - Especially important if other modules depend on the current module.
- Test often: perform intermediate tests to make sure it is correct up to certain steps.

Keep your code clean

- Write your code in the fewest instructions possible, e.g., writing a complete expression rather than steps
- Using variables rather than values

```
import numpy as np
s = 0
a = np.random.rand(10)
for i in range(10):
    s = s + a[i]
```

✗

```
import numpy as np
n = 10
s = 0
a = np.random.rand(n)
for i in range(n):
    s = s + a[i]
```

✓

```
n = 10
s = sum(np.random.rand(n))
```

✓✓

Keep your code clean

- Use short descriptive names for variables: n vs. theNumberofRandomNumbersToBeAdded

- Comment frequently: no comment vs. over-comment

Catch runtime errors

- Handle errors or exceptions gracefully: try-except

```
try:
    code block 1
except ExceptionName:
    code block 2
```

```
x = '6'
try:
    if x > 3:
        print('X is larger than 3')
except TypeError:
    print("x was not a valid number. Try again...")
```

```
x = 's'
try:
    if x > 3:
        print(x)
except:
    print(f'Something wrong with x = {x}')
```

raise an exception

- Program will display an exception and stop running.

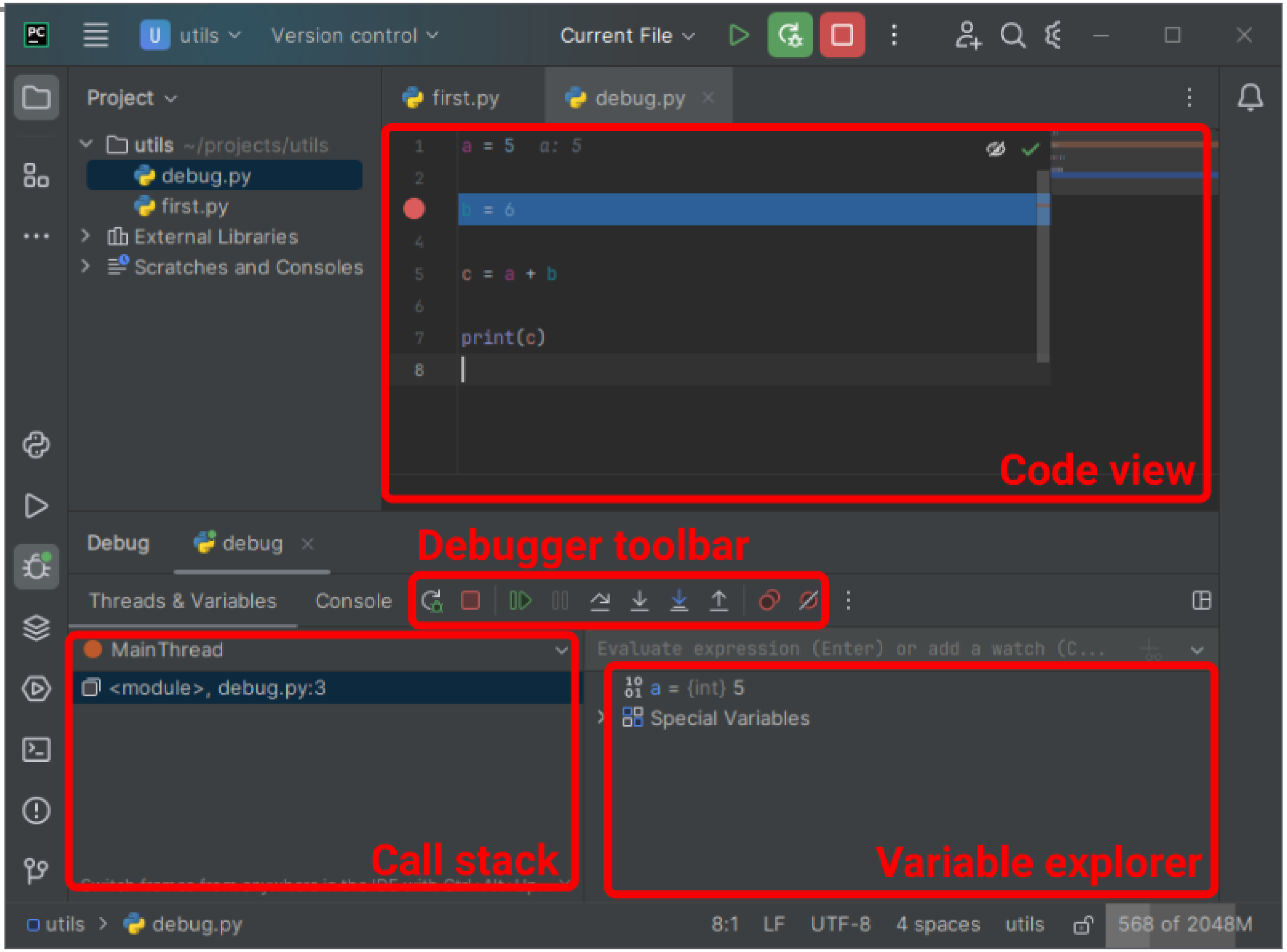
```
x = 10
if x > 5:
    raise(Exception('x should be less or equal to 5'))
```

- Do not overuse try-except or raise: they don’t replace good programming practice.

Use of a debugger (with a breakpoint)

- Breakpoint
- Stepping over
- Stepping in

See debug examples



Python: a computational tool

- Applications in lots of engineering and science applications
- Finding solutions to various types of equations (MAE 3013)
  - Ax = b (linear equations): Gauss elimination, Gauss Seidel, Jacobi
  - g(x) = 0 (nonlinear equation): iterative method, secant method, …
    - Using Python packages (numpy, scipy)
  - Integration of g(x): Simpson’s rule, using python packages (quad)
  - Ordinary differential equations (ODE): odeint
  - Specific problems: Least square fit, Fourier transform
- Machine learning
- Tools: plotting, read from/write to files, load data, interpolation

How to code an algorithm in Python (or any programming language)?

- Computer programs don't replace you: you are the critical thinker! Programs do the computation for you. You obtain and examine the answers and **iterate!**
- Start with understanding the process of solving a given problem
  - Formulate and identify the problem mathematically (if needed)
- **Create a pseudo code before the actual coding**: a 'recipe' on paper (**not in your mind**)
  - The identified process is converted to "sequential", "basic" steps that are programming-friendly. For example, *for* loops, *if-else,* etc.
  - Think about and **look up** what data structure best suits your need: lists, numbers, strings, etc.
- Program the pseudo code in Python: pay attention to syntax, **indices**, etc.
  - If you find logic errors, update your pseudo code (the 'recipe') first
- Debugging: first make sure your program compiles/runs, then debug based on the output of the programs.
  - Debug step by step: since you know how to solve the problem, you can expect the answers at each step. You may need to update the pseudo code!
  - Use simple problems (you know the answers or you can verify the answers by another program that is known to be correct!) to begin testing your program
  - Then increase the **complexity** of the testing problems! (blind test, corner cases)
- Optimization of the code/pseudo code