Covenant University

Raising a new Generation of Leaders

PET328

COMPUTER APPLICATIONS IN PETROLEUM ENGINEERING

PET328: COMPUTER APPLICATIONS IN PETROLEUM ENGINEERING

(With Python Programming)

python Programming

Olatunde O. Mosobalaje (PhD)

Department of Petroleum Engineering,
Covenant University, Ota
Nigeria

OUTLINE

- Preambles
  - The Appetizer
  - The Toolbox
  - The Embedded Course
  - Introduction to Computer Programming
- Getting Started with Python
  - Basic Python Objects
  - Conditional Execution
  - Repeated Execution
  - Functions
- Python Data Structures
  - Strings
  - Lists
  - Tuples
  - Dictionaries
- Some Python Libraries
  - NumPy
  - Matplotlib
  - Pandas
  - Scikit-learn
- Application Projects
  - Oil Reservoir Volumetrics
  - Material Balance Analysis
  - PVT Properties

The Appetizer – a presentation

ACQUIRING NASCENT SKILLS FOR EMERGING OIL AND GAS OPPORTUNITIES: DATA ANALYTICS, MACHINE LEARNING AND ARTIFICIAL INTELLIGENCE

The Toolbox

- For this course, the following tools would be needed:
  - Python 3
  - Python Integrated Development and Learning Environment (IDLE)
  - Git and GitHub

PREAMBLES

The Toolbox

- Installing Python 3

To install the latest release of Python 3, go to Python download website: https://www.python.org/downloads/

PREAMBLES

The Toolbox

- Installing Python 3

Launch the downloaded executable file by double-clicking the file in your download folder.

Follow the steps as the installer leads

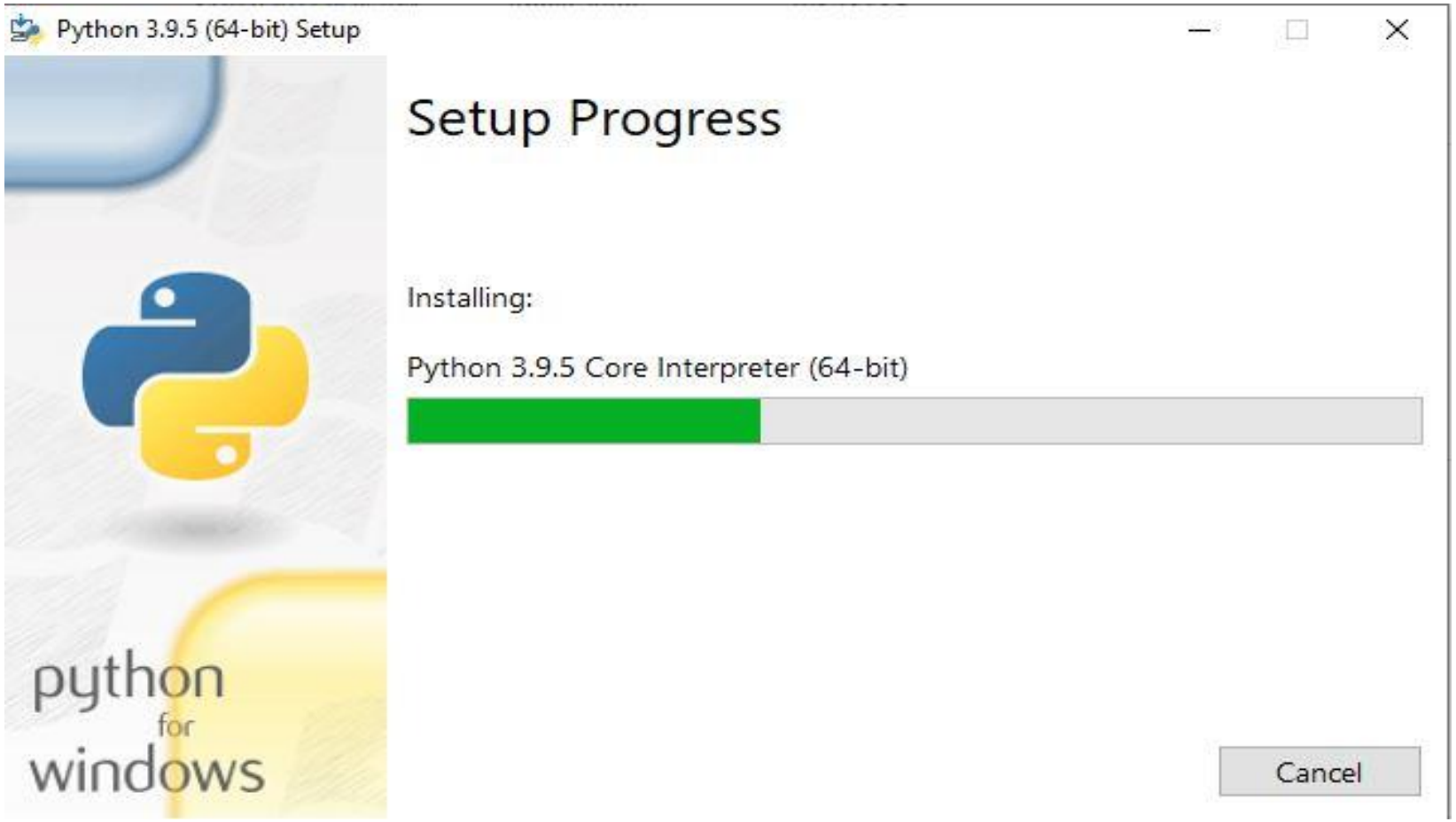
Click on the default installation option.

Ensure to check the Add Python 3.9 to PATH

PREAMBLES

The Toolbox

- Installing Python 3

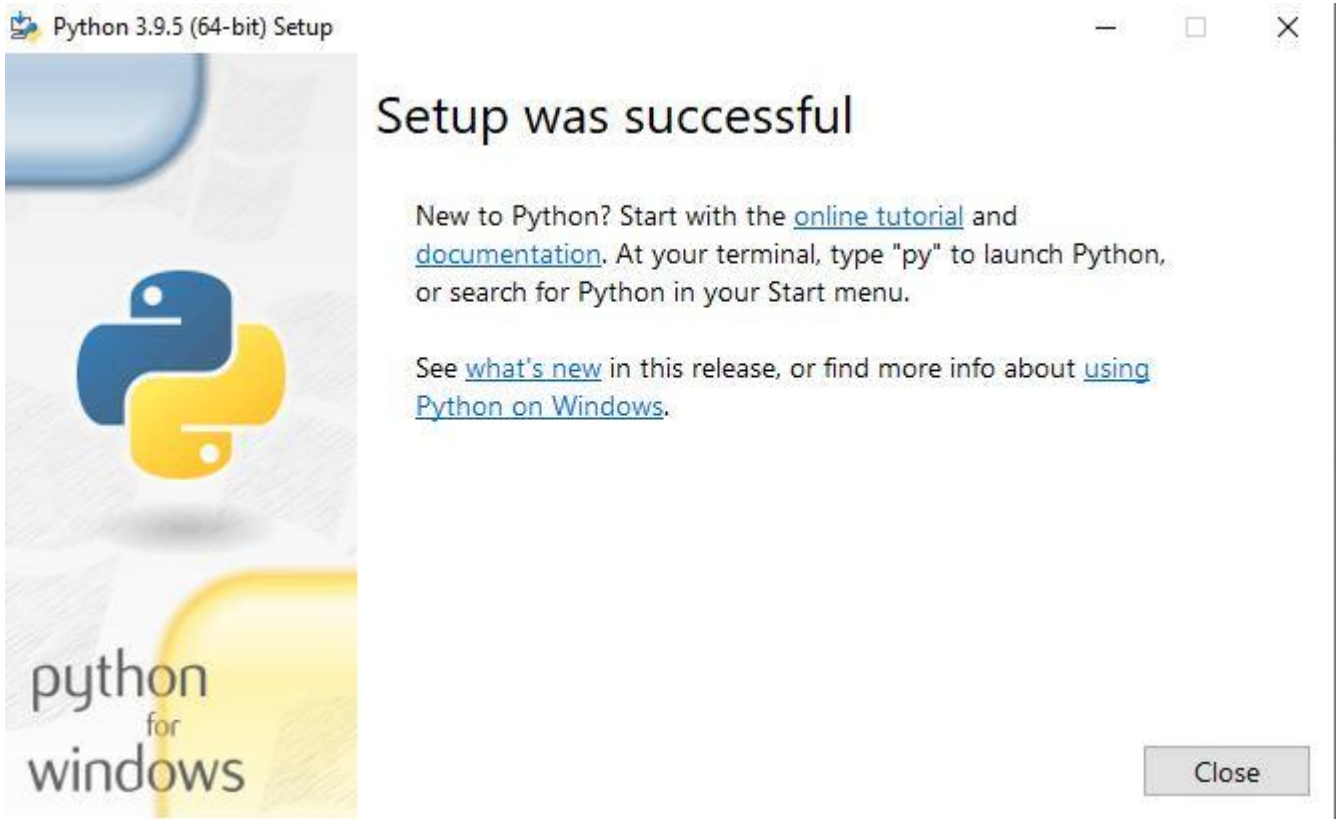


PREAMBLES

The Toolbox

- Installing Python 3

Click the close button when the installation is completed

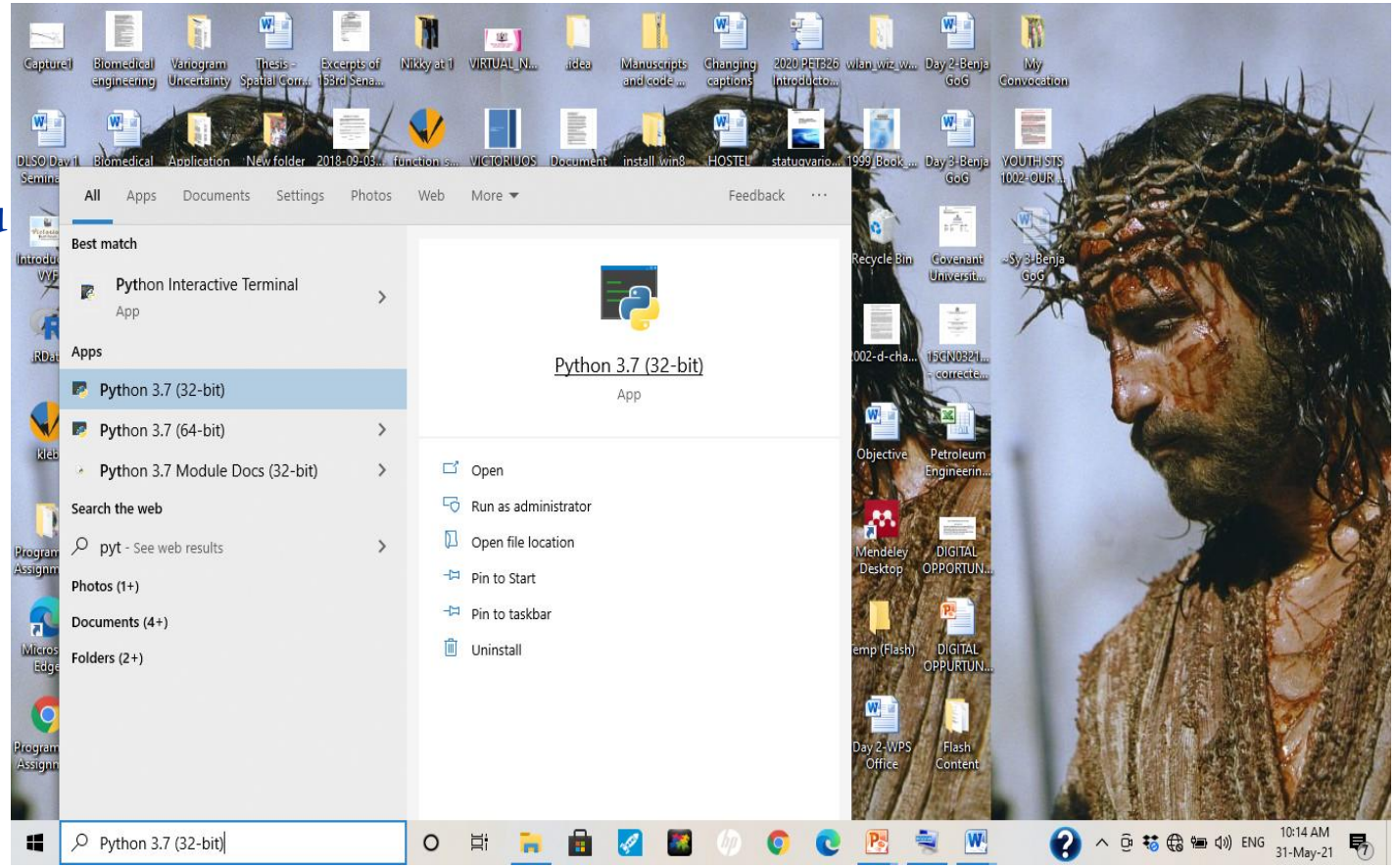


PREAMBLES

The Toolbox

- Launching Python 3

Simply type Python into the Start Menu search box and click the Python program.

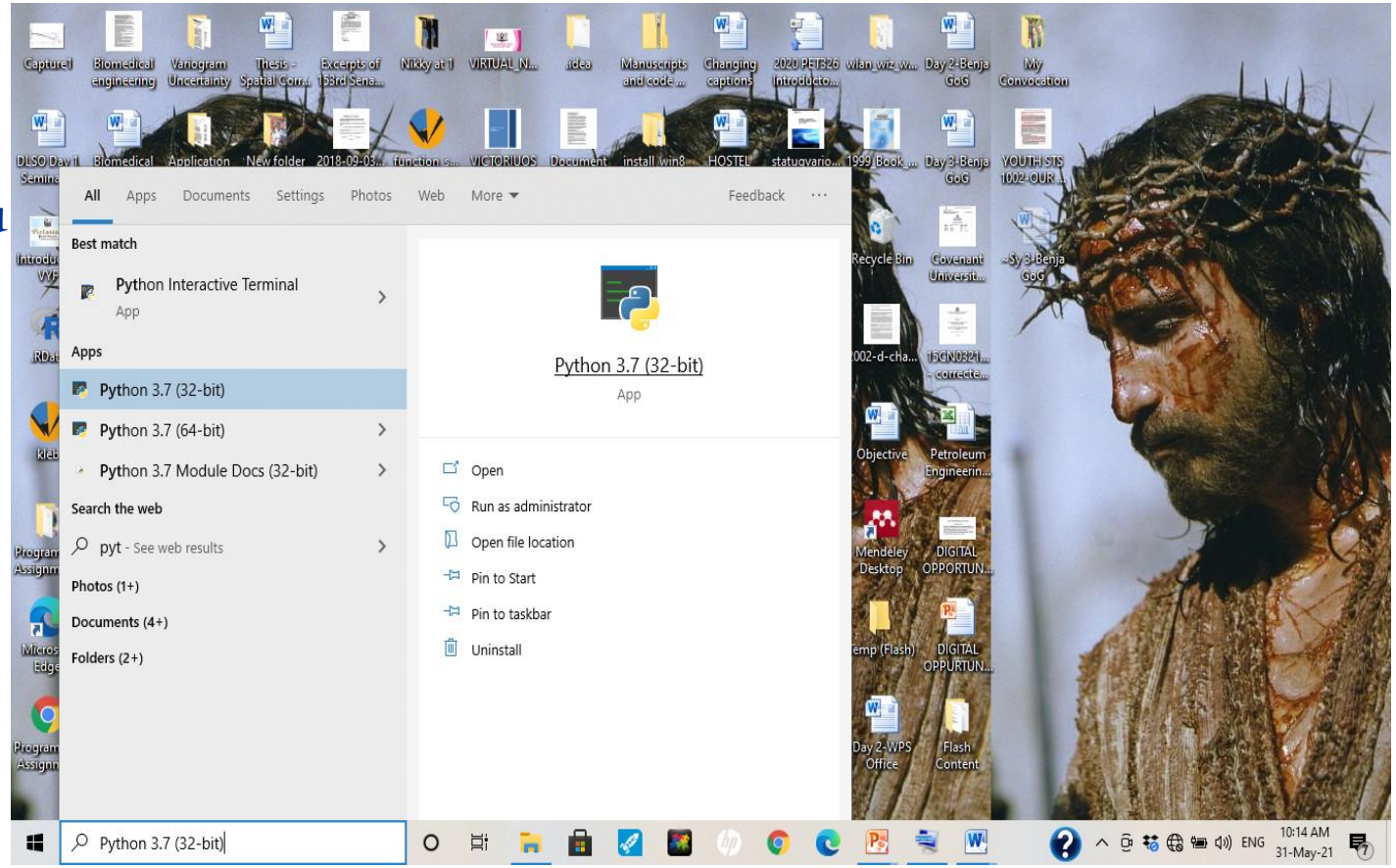


PREAMBLES

The Toolbox

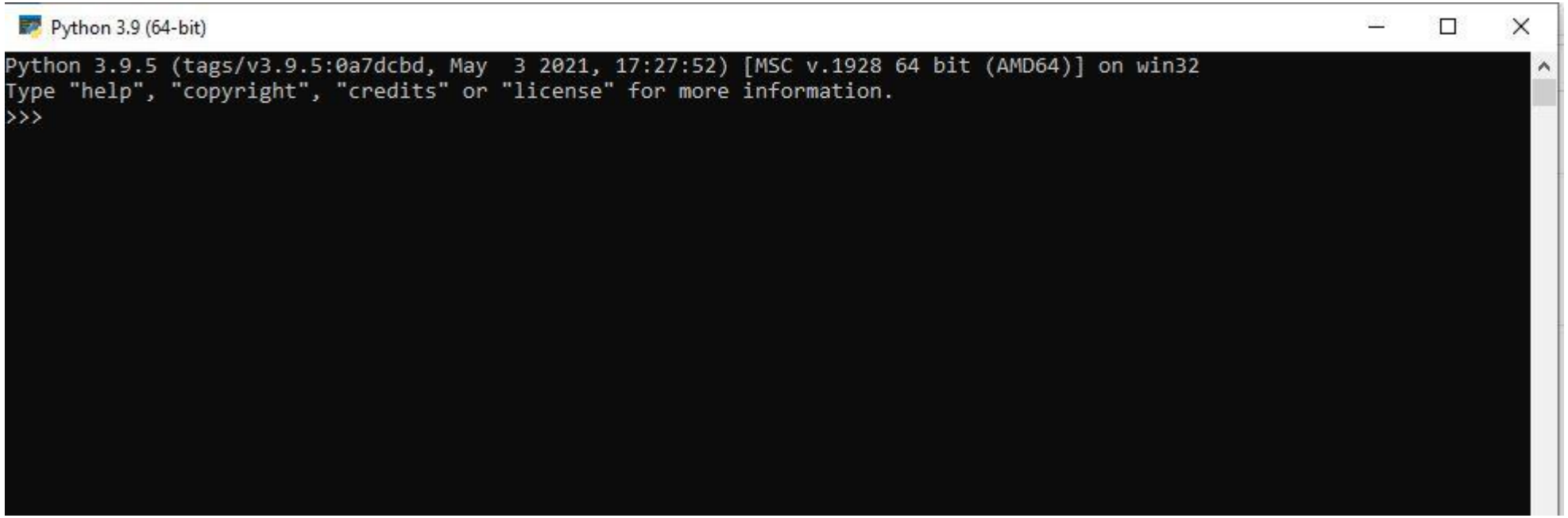
- Launching Python 3

Simply type Python into the Start Menu search box and click the Python program.



The Toolbox

- Launching Python 3



The Toolbox

- Python IDLE

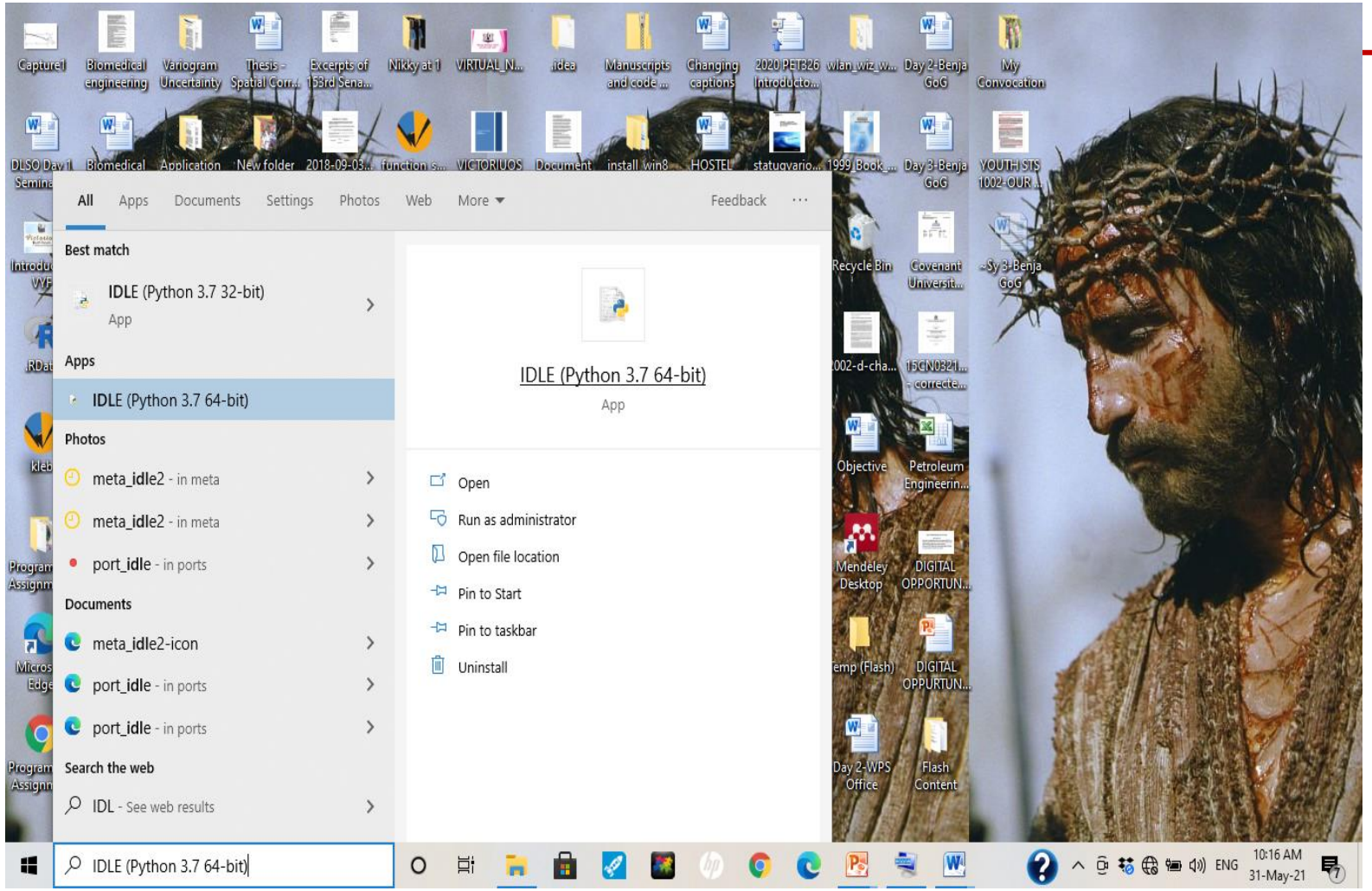
Now, the Python DOS-like environment seems boring. Good enough, we will typically not be working on that platform; rather we will interact with Python from a platform known as Interactive Development and Learning Environment (IDLE)

PREAMBLES

The Toolbox

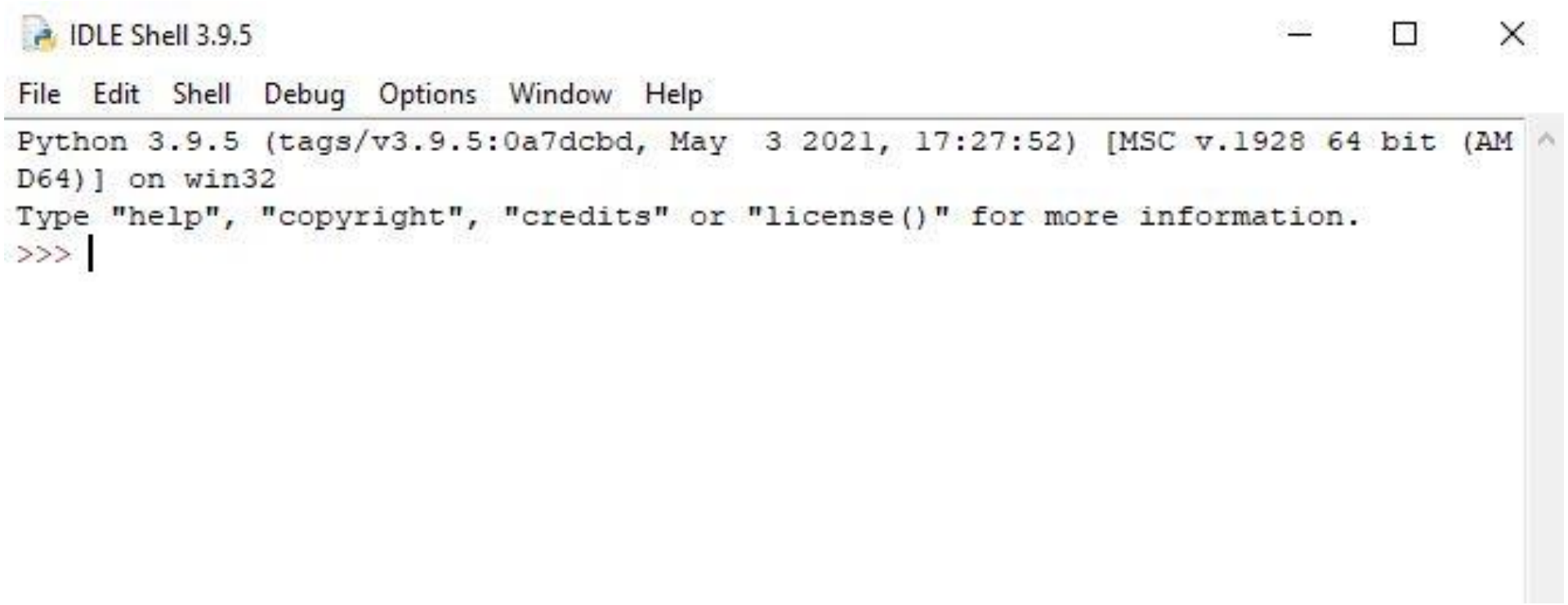
- Python IDLE

To launch IDLE, simply type IDLE into the Start Menu search box and click on the IDLE program.



The Toolbox

- Python IDLE



PREAMBLES

The Toolbox

- Python IDLE

There are two ways by which you could communicate with Python from the IDLE environment:

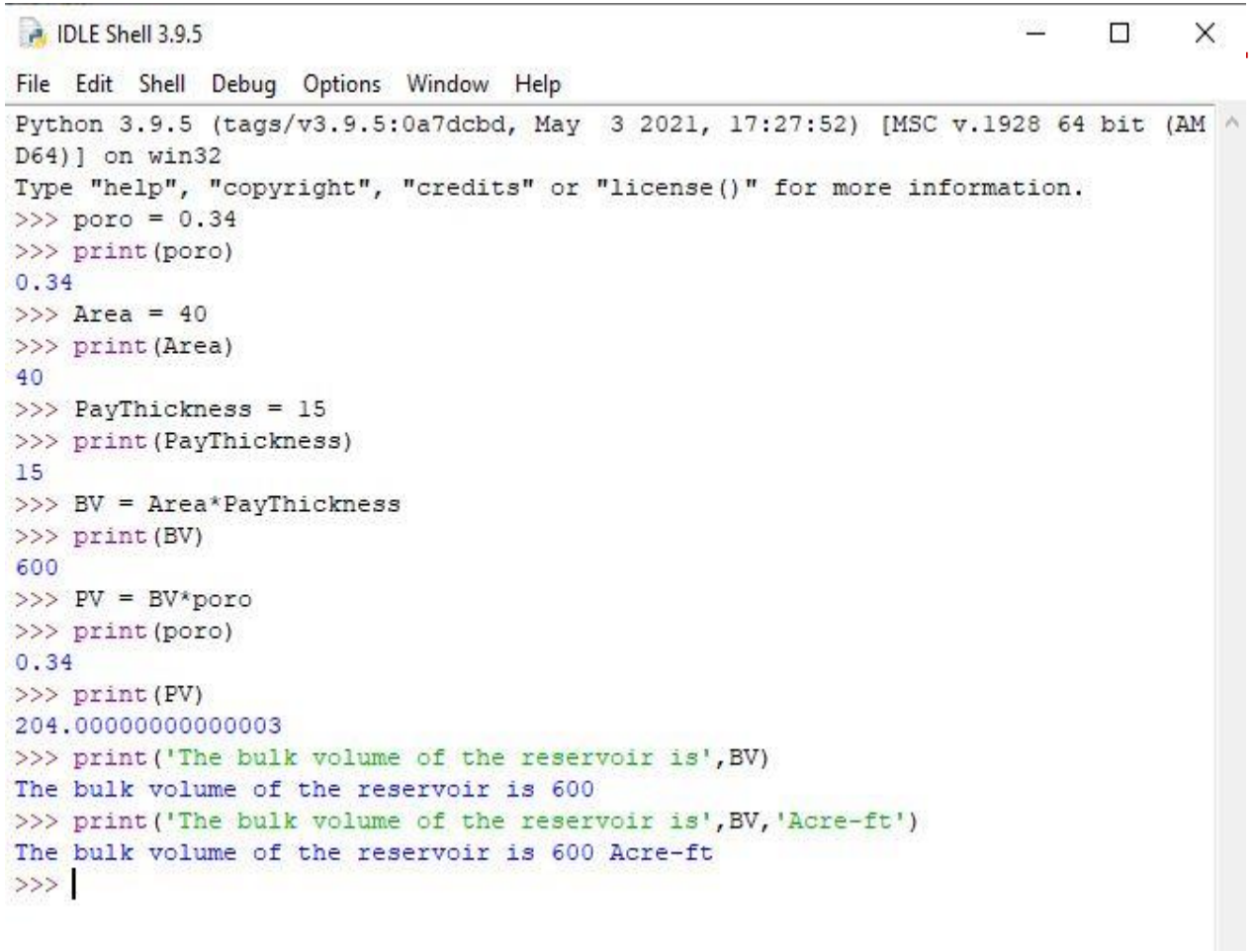
- Interactive
- From a file (script)

PREAMBLES

The Toolbox

- Communicating with Python interactively

In this case, you type in Python command (one at a time) into the console. Each command get executed once the 'Enter' key is pressed. Depending on the command, results may be displayed on the console once the command is executed.

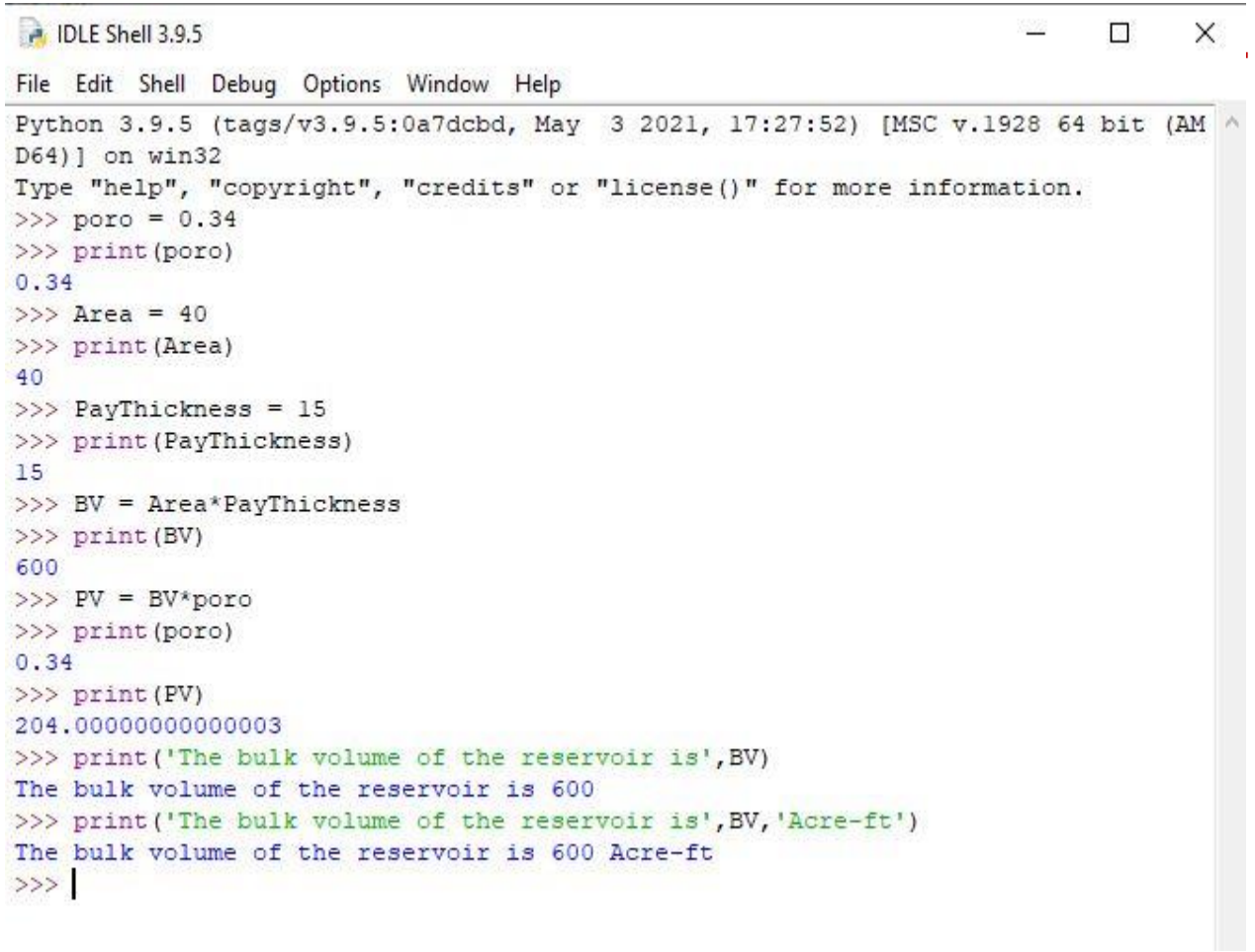


PREAMBLES

The Toolbox

- Communicating with Python interactively

In this case, you type in Python command (one at a time) into the console. Each command get executed once the 'Enter' key is pressed. Depending on the command, results may be displayed on the console once the command is executed.



PREAMBLES

The Toolbox

- Communicating with Python from a file

In this case, you type in Python commands (all at a time) into a text file editor (code editor). The commands don't get executed as they are being typed. Rather, they get executed (sequentially) when submitted as a whole to the Python interpreter.

PREAMBLES

The Toolbox

- Communicating with Python from a file

Any text editor program could be used for this purpose, as long as the file is saved as a .py file.

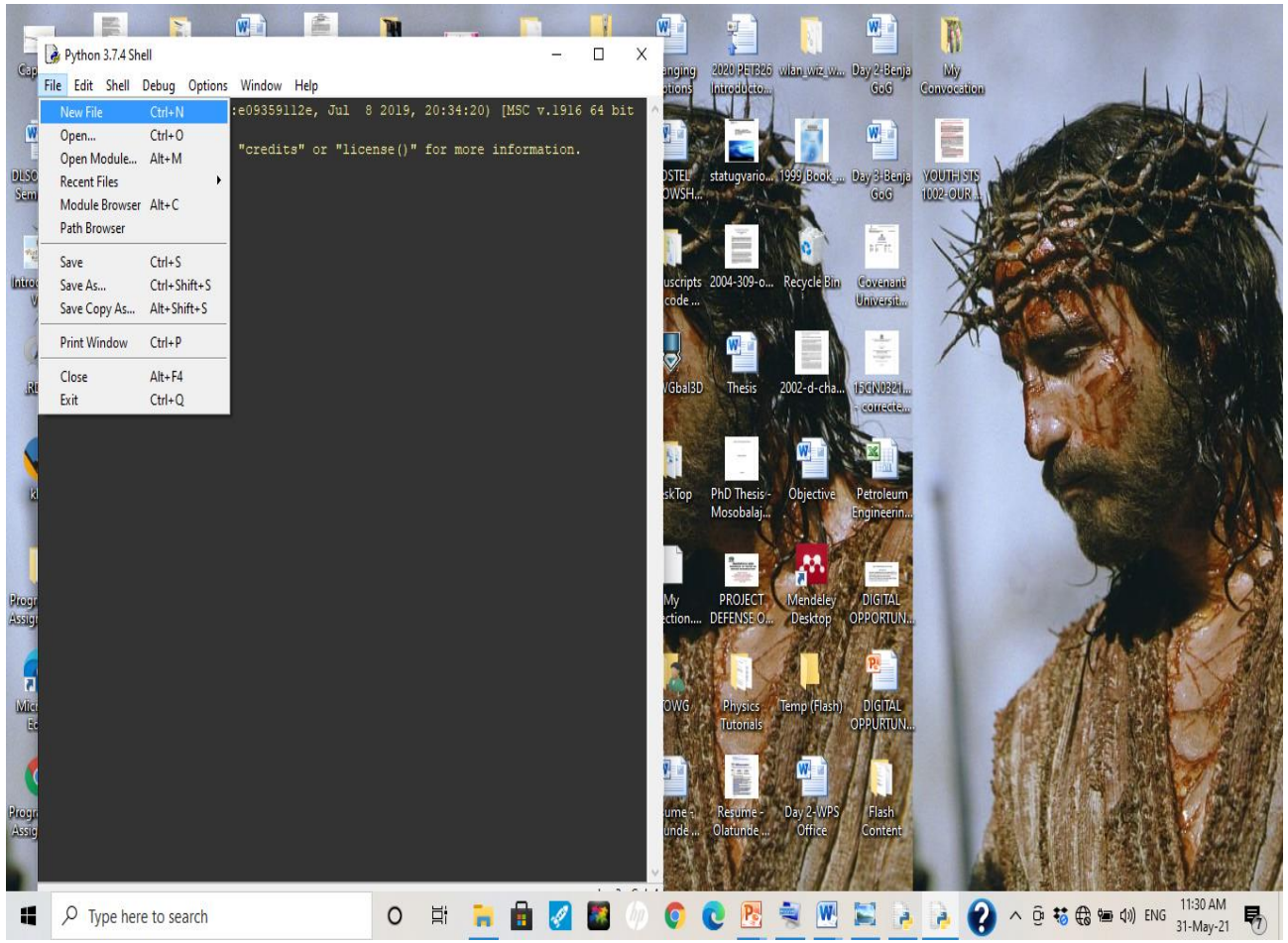
Good, Python has an in-built text editor for this purpose.

PREAMBLES

The Toolbox

- Communicating with Python from a file

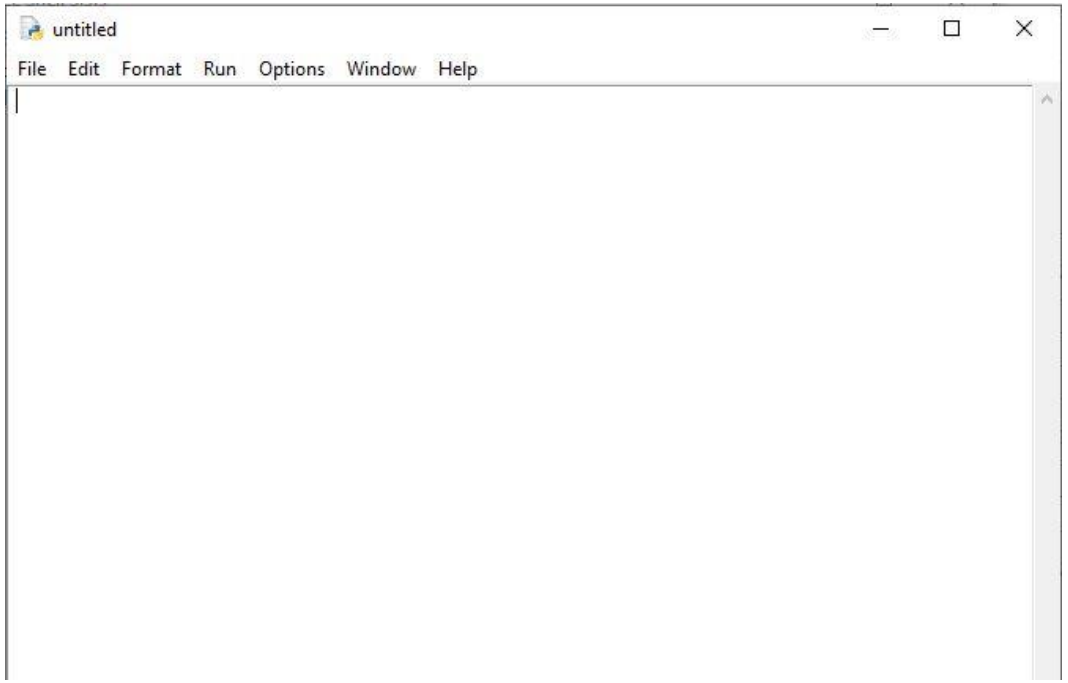
To launch Python's in-built code editor, just click on the File menu and choose New File.



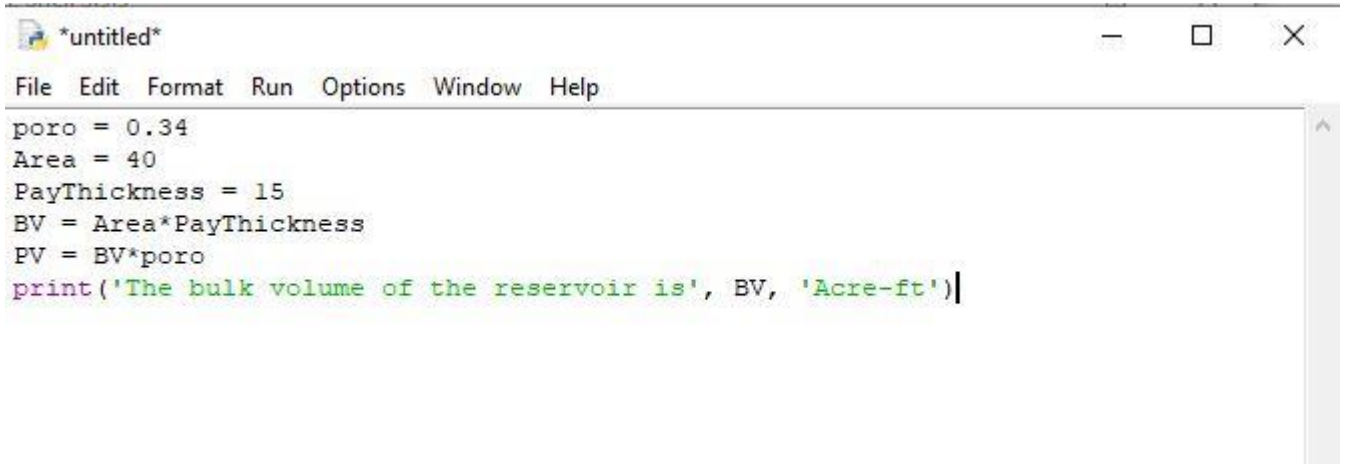
PREAMBLES

The Toolbox

- Communicating with Python from a file



Once the editor is opened, you can type in your lines of codes.



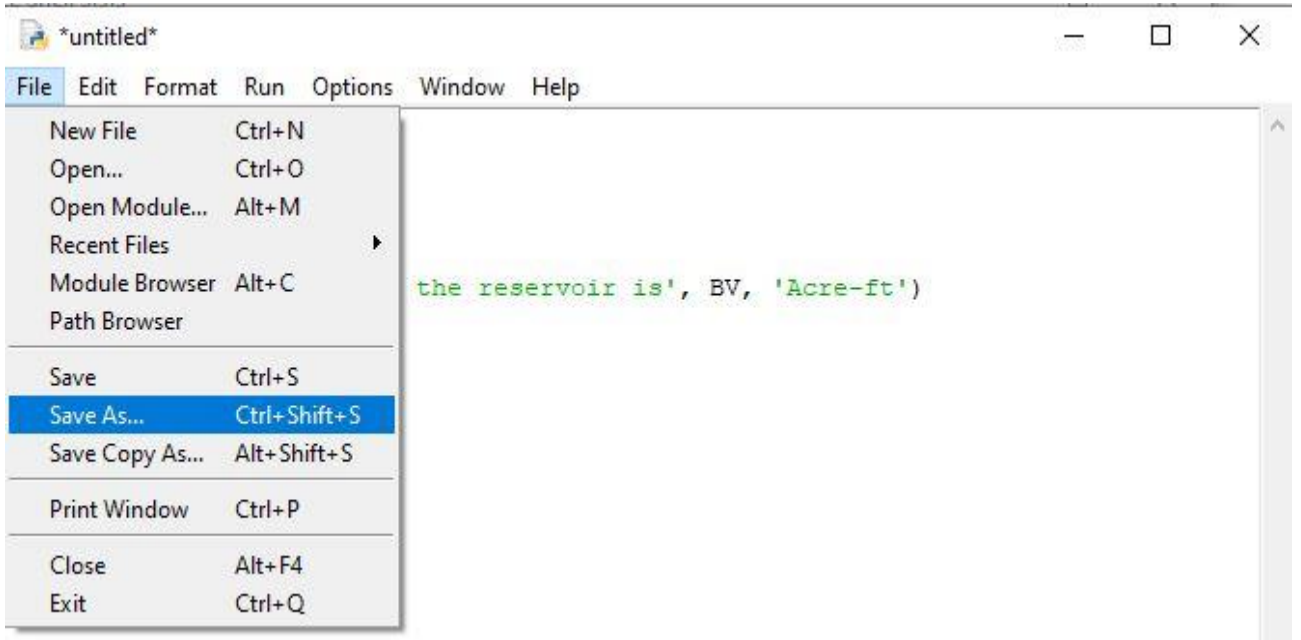
PREAMBLES

The Toolbox

- Communicating with Python from a file

Before submitting the lines of codes in the code editor to the Python interpreter, you need to save the editor file.

To save, simply go the File menu and choose Save As



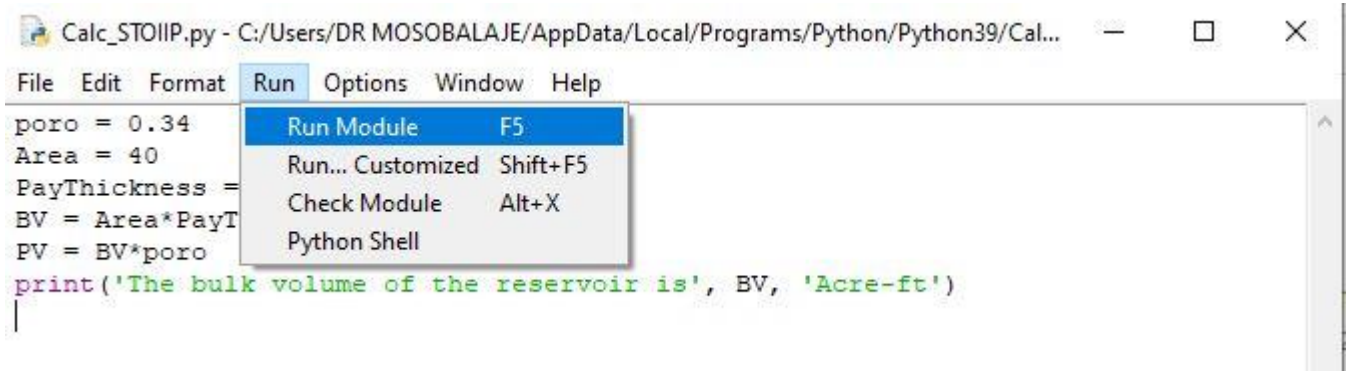
PREAMBLES

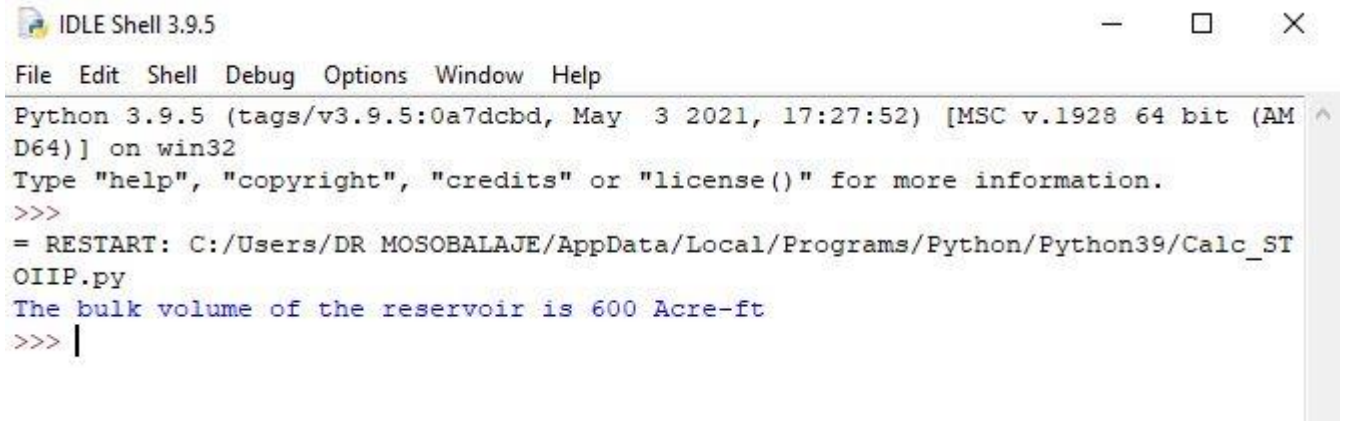
The Toolbox

- Communicating with Python from a file

Once the file (script) is saved, the code lines can be submitted to the Python interpreter by choosing item 'Run Module' in the Run menu.

The output of the code execution (if any) is subsequently displayed on the Python console.

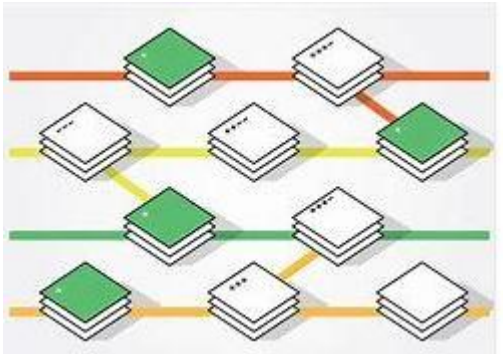




PREAMBLES

The Toolbox

- Git and GitHub



Git is an open source version control software.

What is Version Control?

Version control (VC) is a system used for keeping track of changes made to a file over time. As the changes are made, the system records and save the state of the file at instances indicated by the user. Such user can revert back to a previous version of the file when necessary.

Essentially, the VC system keeps the latest version of the file but also keeps a record of all changes between all versions.

PREAMBLES

The Toolbox

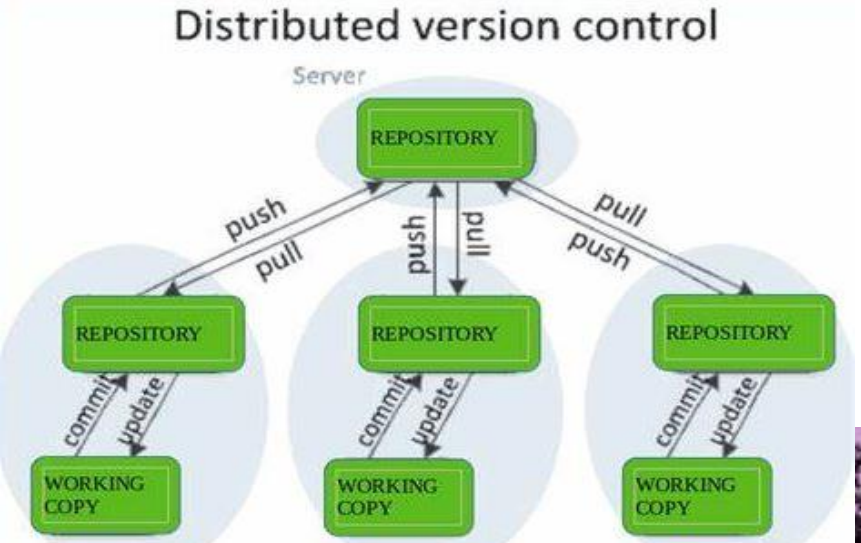
- Git and GitHub

And, there is something called Distributed Version Control (DVC)

What is Distributed Version Control?

Typically, real life projects (including oilfield digital projects) are done by teams whose members need to collaborate – work together on same files. Individual members of the team can make changes to such shared files. There is therefore a need to make such file available on a central server and to keep track of the following:

- who made what change?
- When was the change made?
- Why was the change made?



The Toolbox

Git and GitHub

And, there is something called Distributed Version Control (DVC)

What is Distributed Version Control?

A version control system that also comes with the capabilities for collaboration among several people is known as Distributed Version Control system.

Git is a version control system – locally hosted on your system.

GitHub is an online platform that interfaces with Git, hosting your files on remote servers thereby making them available for collaboration with others.

The Toolbox

Git and GitHub

In this course, we shall be working as a team, therefore, both Git and GitHub are part of tools we shall be using. Essentially, submissions to some assignments shall be in the form of code file editing and sharing between students and the Course Instructor.

Assignment 1

Get the following tools ready on your PC:

- Git - install
- A user account on github.com
- GitHub desktop - install

PREAMBLES

The Embedded Course

A Coursera course is embedded into this course (PET328). It is compulsory that all students completes the Coursera course as it is part of the assessment items in PET328.

PREAMBLES

The Embedded Course

The embedded course is titled 'Programming for Everybody (Getting Started with Python).

The course is offered by University of Michigan.

Programming for Everybody (Getting Started with Python)

Offered by

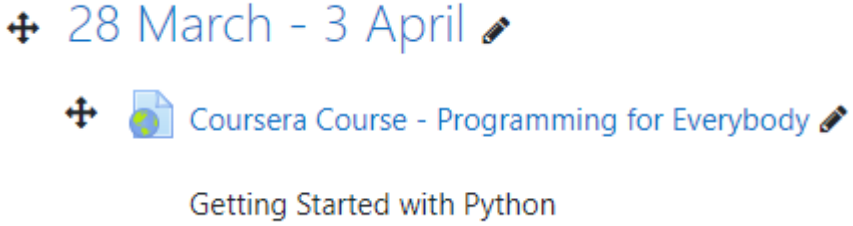
UNIVERSITY OF MICHIGAN

PREAMBLES

The Embedded Course

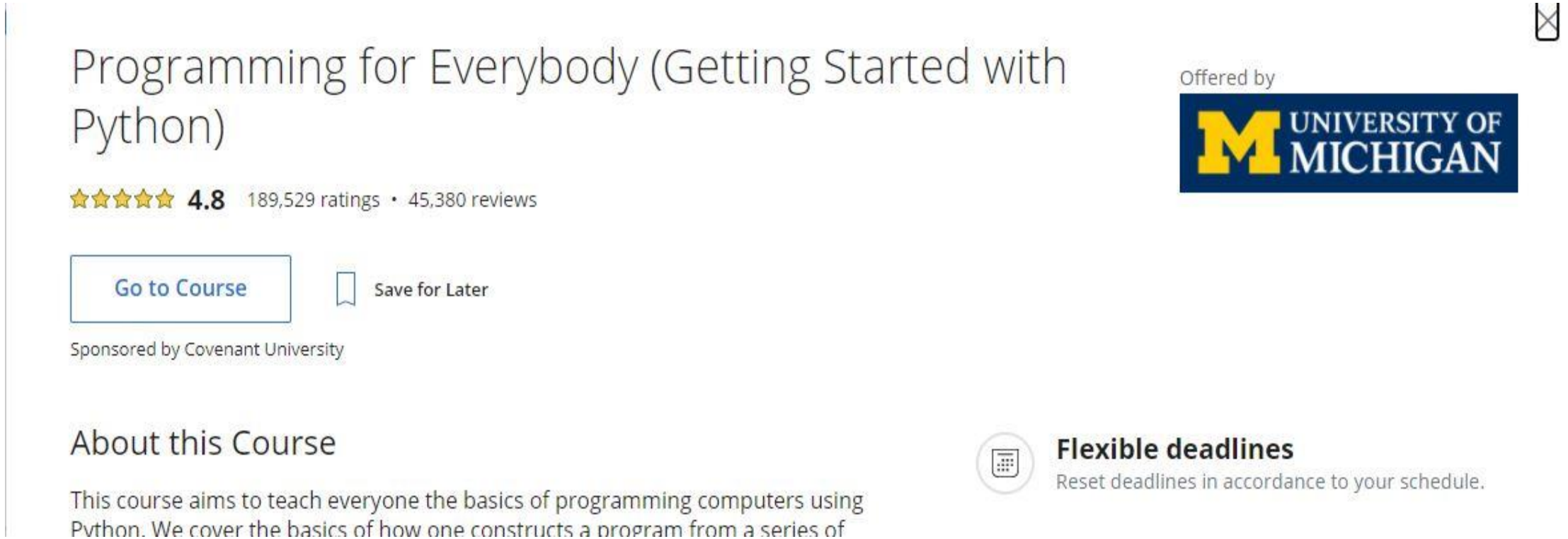
The link to the embedded course has been added to the PET328 course site on Moodle.

To enroll for the Coursera course, simply click on the link.



PREAMBLES

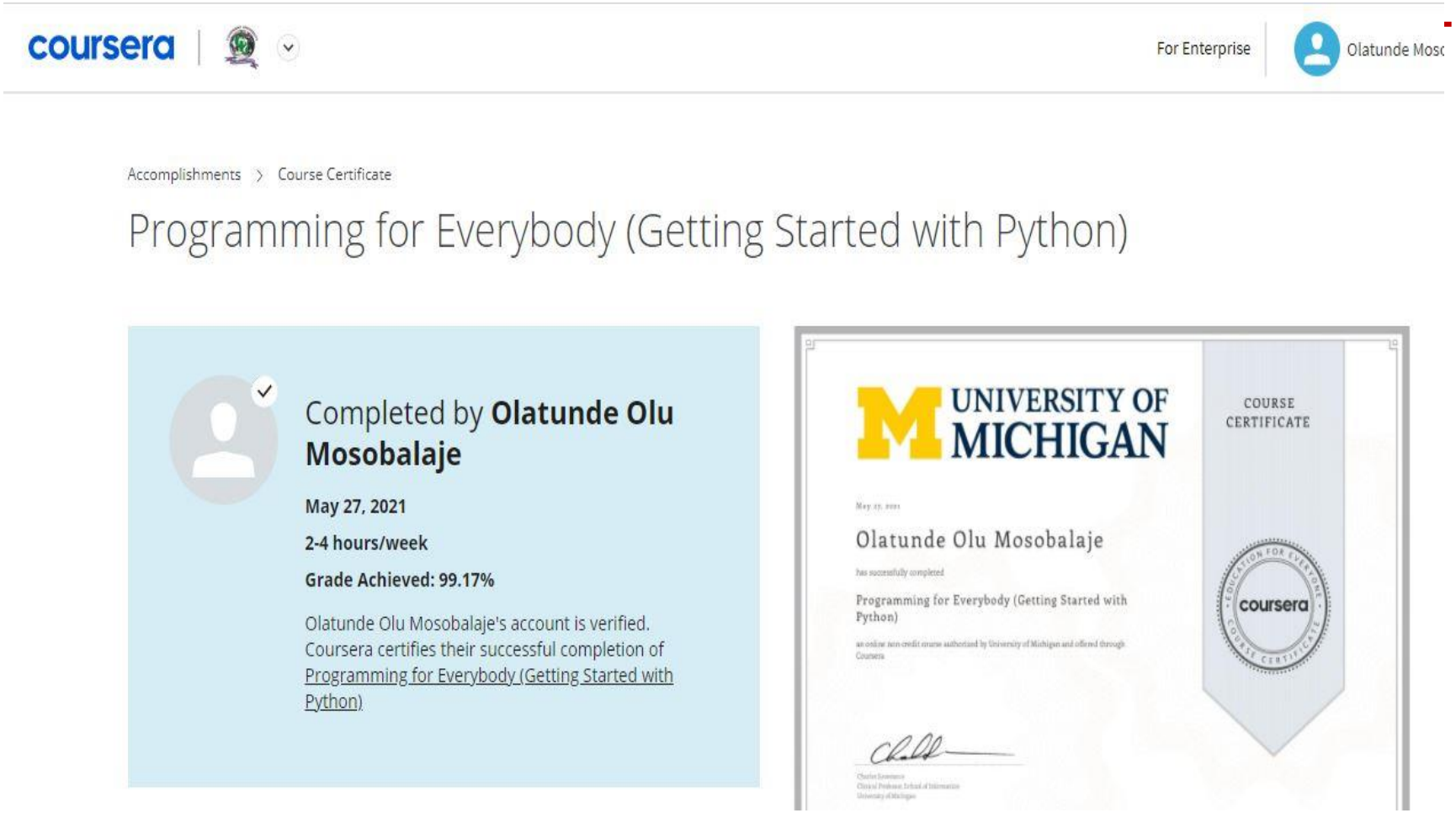
The Embedded Course



PREAMBLES

The Embedded Course

- When you complete the course, you are awarded a certificate of completetion!!!



Introduction to Computer Programming

- Analogy: Programming language vs. Natural language
  - There is a striking similarity between learning programming language and learning natural language. In both cases, the process is thus:
    - learn the vocabulary and the grammar – spell words, construct sentences etc.
    - Communicate
      - natural language: use words, sentences, paragraphs to communicate an idea
      - Programming language: use keywords, variables, functions, expressions, statements to communicate steps to computer.

PREAMBLES

Introduction to Computer Programming

- A program is simply a collection of sequential Python statements written to perform a specific task

```python
def bubble_sort(list):
    sorted_list = list[:]
    is_sorted = False
    while is_sorted == False:
        swaps = 0
        for i in range(len(list) - 1):
            if sorted_list[i] > sorted_list[i + 1]: # swap
                temp = sorted_list[i]
                sorted_list[i] = sorted_list[i + 1]
                sorted_list[i + 1] = temp
                swaps += 1
        print(swaps)
        if swaps == 0:
                is_sorted = True
    return sorted_list

print(bubble_sort([2, 1, 3]))
```

Introduction to Computer Programming

Fundamental patterns (concepts) in a program

- The following are typical patterns (statement(s)) you see in a program:
  - Input statements
  - Output statements
  - Sequential execution
  - Conditional statements
  - Repeated execution (loops)
  - Reuse of statements (functions)

PREAMBLES

Introduction to Computer Programming

Fundamental patterns (concepts) in a program

- Input statements: - used to request and accept data from users
- Example: the input function.

Input_Output_demo.py - C:\Users\TTOWG\645\1 karia def\2. CU\CU Courses\PET328 - Computer Applications in Petroleum

File Edit Format Run Options Window Help

```
#...TTOWG!

# input statements
poro = input('Enter the value of porosity: ')
area = input('Enter the value of area: ')
paythickness = input('Enter the value of pay zone thickness: ')

area = float(area)
paythickness = float(paythickness)

BV = area*paythickness

# output statement
print('The bulk volume of the reservoir is', BV, 'Acre-ft')
```

This script is available here

https://github.com/TTOWG/PET328_2021_Class/blob/main/demo_Input_Output.py

PREAMBLES

Introduction to Computer Programming

Fundamental patterns (concepts) in a program

- Output statements: - used to display the results of execution on the screen.
- Example: the print function

Input_Output_demo.py - C:\Users\TTOWG\645\1 karia def\2. CU\CU Courses\PET328 - Computer Applications in Petroleum

File Edit Format Run Options Window Help

```
#...TTOWG!

# input statements
poro = input('Enter the value of porosity: ')
area = input('Enter the value of area: ')
paythickness = input('Enter the value of pay zone thickness: ')

area = float(area)
paythickness = float(paythickness)

BV = area*paythickness

# output statement
print('The bulk volume of the reservoir is', BV, 'Acre-ft')
```

PREAMBLES

Introduction to Computer Programming

Fundamental patterns (concepts) in a program

- Sequential execution: - typically, a program would entail multiple statements.
- Statements are executed in the order (sequence) in which they are encountered.
- Latter statements can make use of results of former statements; not vice-versa

Input_Output_demo.py - C:\Users\TTOWG\645\1 karia def\2. CU\CU Courses\PET328 - Computer Applications in Petroleum

File Edit Format Run Options Window Help

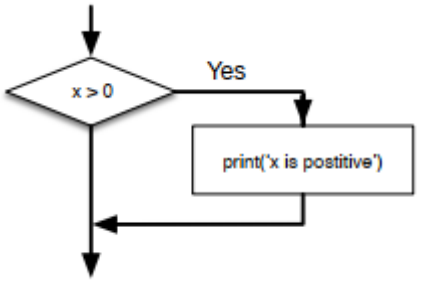
```
#...TTOWG!

# input statements
poro = input('Enter the value of porosity: ')
area = input('Enter the value of area: ')
paythickness = input('Enter the value of pay zone thickness: ')

area = float(area)
paythickness = float(paythickness)

BV = area*paythickness

# output statement
print('The bulk volume of the reservoir is', BV, 'Acre-ft')
```

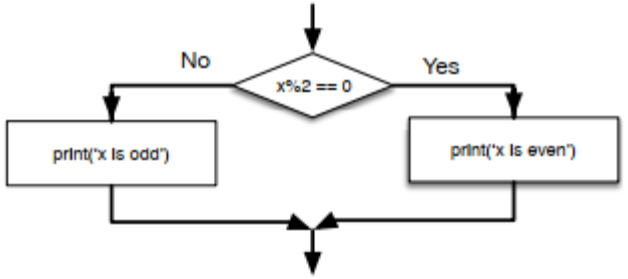
PREAMBLES

Introduction to Computer Programming

Fundamental patterns (concepts) in a program

- Conditional Statements: patterns that make it possible for the program to check for some conditions and decide to:
  - perform a statement(s) or skip the statement(s)
  - Choose between alternative statements.





conditional_statement_demo.py - C:\Users\TTOWG\645\1 karia def\2. CU\CU Courses\PET328 - Computer Applications in Petroleum Enginee

File Edit Format Run Options Window Help

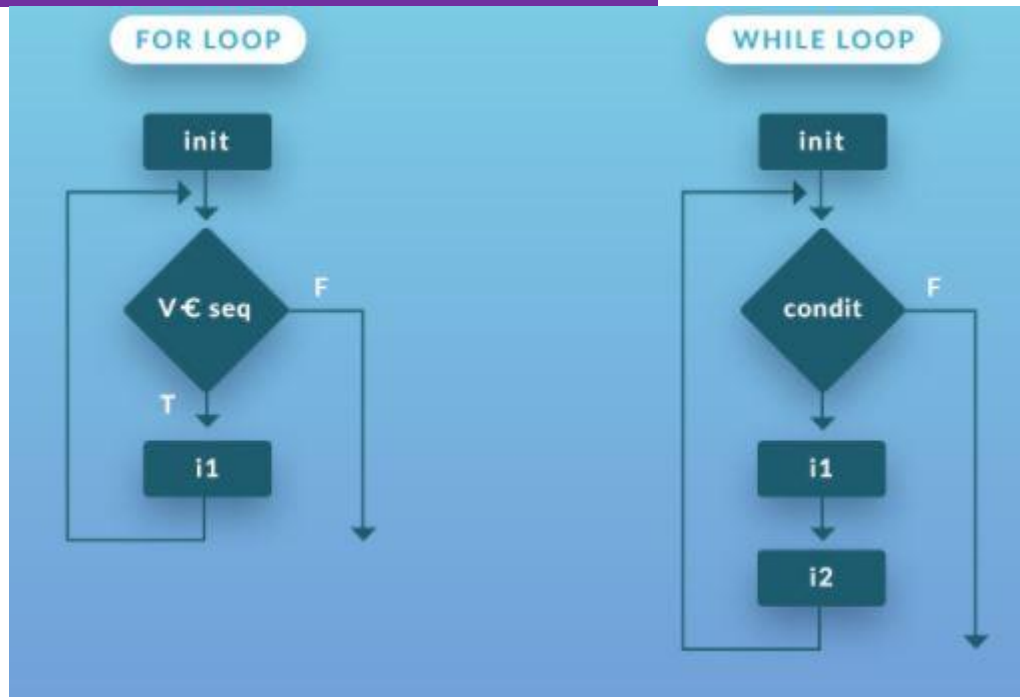
```
#...TTOWG!

initial_pressure = input('Enter the value of initial pressure: ')
bubble_pressure = input('Enter the value of bubble-point pressure: ')

initial_pressure = float(initial_pressure)
bubble_pressure = float(bubble_pressure)

if initial_pressure > bubble_pressure:
    print('The reservoir is undersaturated!!!')
else:
    print('The reservoir is saturated!!!')
```

PREAMBLES

Introduction to Computer Programming

Fundamental patterns (concepts) in a program

- Repeated Execution: patterns that instructs the program to perform (iterate) a given statement(s) repeatedly, for each item in a set of items, varying values of parameter(s) from item to item.



repeated_execution_demo.py - C:/Users/TTOWG/645/1 karia def/

File Edit Format Run Options Window Help

```
#...TTOWG!

blocks = [1,2,3,4,5]
for block in blocks:
    print('This is Block', block)
```

PREAMBLES

Introduction to Computer Programming

Fundamental patterns (concepts) in a program

- Re-use of statements: the task performed by some statement(s) might be routine and needed at various points in your program.
- Such statement(s) may be written once, saved with a name and re-used at various points in your program by referring to the name.

statement_reuse_demo.py - C:\Users\TTOWG\645\1 karia def\2. CU\CU Courses\PET328 - Computer Applications in Petroleum Engineering\Demos\statement_reuse_de...

File Edit Format Run Options Window Help

```
#...TTOWG!

# function definition
def stoiip_calc(area, thickness, poro, sw, boi):
    STOIIP = (7758*area*thickness*poro*(1-sw))/boi
    return STOIIP

# function call for Reservoir TTOWG_1 (re-use)
oil_inplace_TTOWG_1 = stoiip_calc(40, 15, 0.3, 0.28, 1.2)
print('The amount of oil in place in Reservoir TTOWG_1 is', oil_inplace_TTOWG_1, 'STB')


# function call for Reservoir TTOWG_2 (re-use)
oil_inplace_TTOWG_2 = stoiip_calc(80, 10, 0.23, 0.35, 1.1)
print('The amount of oil in place in Reservoir TTOWG_2 is', oil_inplace_TTOWG_2, 'STB')
```

```
>>>#TTOWG!
>>>print('…to the only wise God')
```

Coming Soon…

Season 3 Episode 2 –

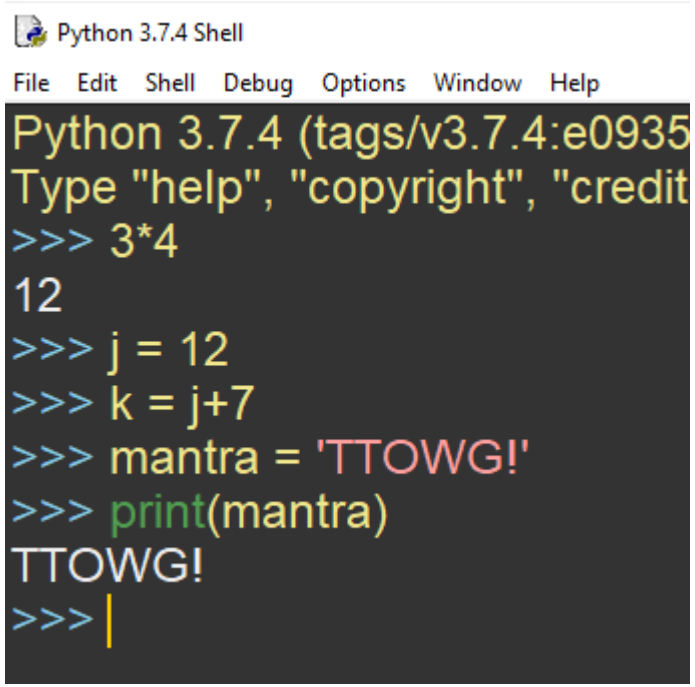
Getting Started with Python

GETTING STARTED WITH PYTHON

Basic Python Objects

- Crudely speaking, Python objects are stuffs upon which actions (specified in python commands) are performed.

- Example: in the code screenshot shown, 3, 4, j, k, 7, mantra, 'TTOWG' are all objects acted upon.



GETTING STARTED WITH PYTHON

Basic Python Objects

- The basic Python objects considered here are Values and Variables.
- Later, some sets of sophisticated objects known as data structure shall be considered.

GETTING STARTED WITH PYTHON

Basic Python Objects

Values

- Values are simply the representation of data entities.

Types of Values

- Values in Python belong to various types such as type *integer*, type *float*, and type *string*.
- Use the function *type* to find out the type to which a value belong.

```
>>> type(2)
<class 'int'>
>>> type('TTOWG!')
<class 'str'>
>>> type(2.0)
<class 'float'>
>>> type('2')
<class 'str'>
```

GETTING STARTED WITH PYTHON

Basic Python Objects

Types of Values

- 2 is of type (class) integer
- 'TTOWG' and '2' are of type string; just like any set of characters (alphanumeric and non-alphanumeric) enclosed in quotes
- 2.0 is of type float; just as are all numbers expressed in decimals.

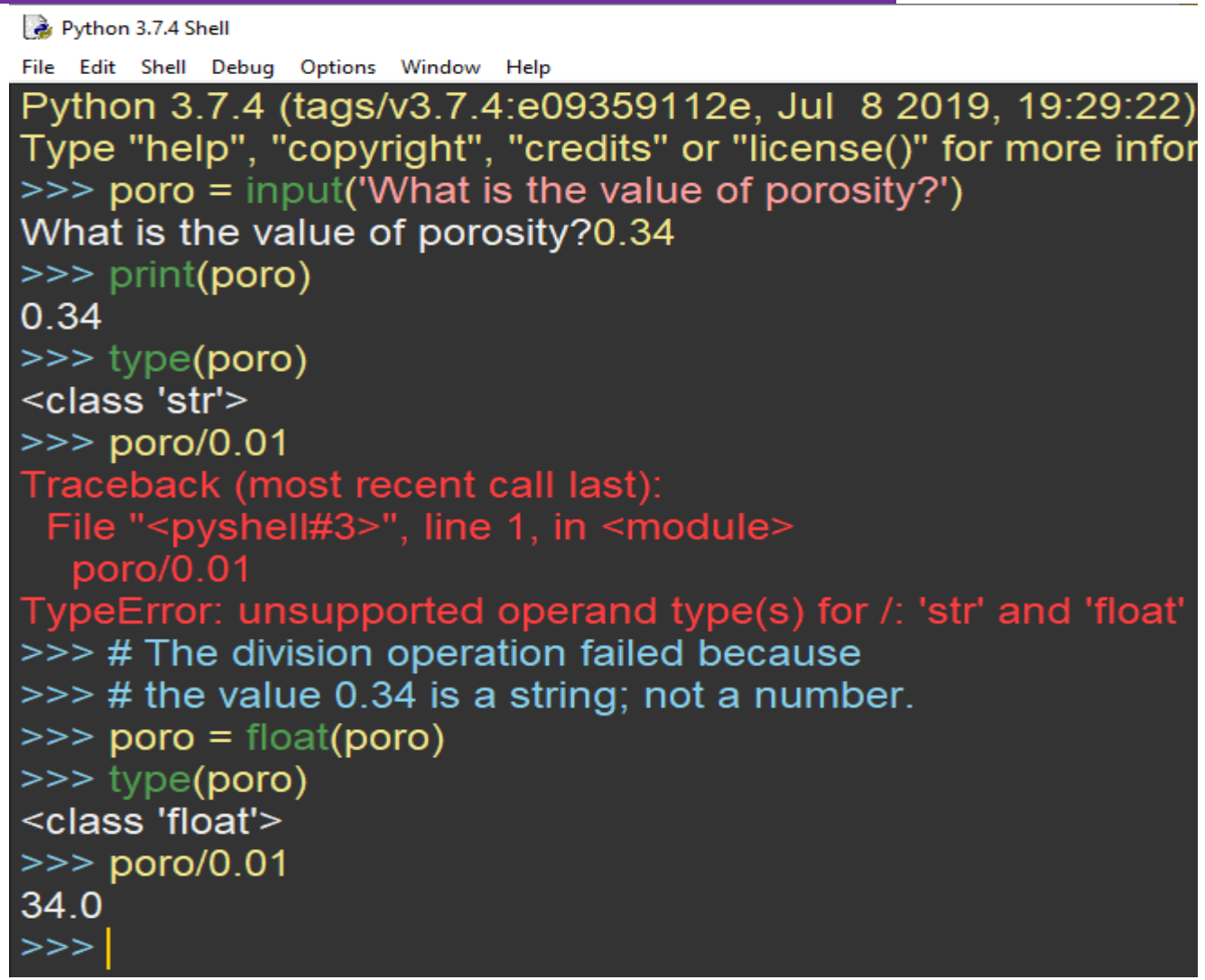
```
>>> type(2)
<class 'int'>
>>> type('TTOWG!')
<class 'str'>
>>> type(2.0)
<class 'float'>
>>> type('2')
<class 'str'>
```

GETTING STARTED WITH PYTHON

Basic Python Objects

Types of Values

- Please, take note that users' response to the input function prompt is stored as a string.
- Before using such *input* in a mathematical operation, they should be converted to a numerical type using *float* or *int* functions.



Basic Python Objects

Variables

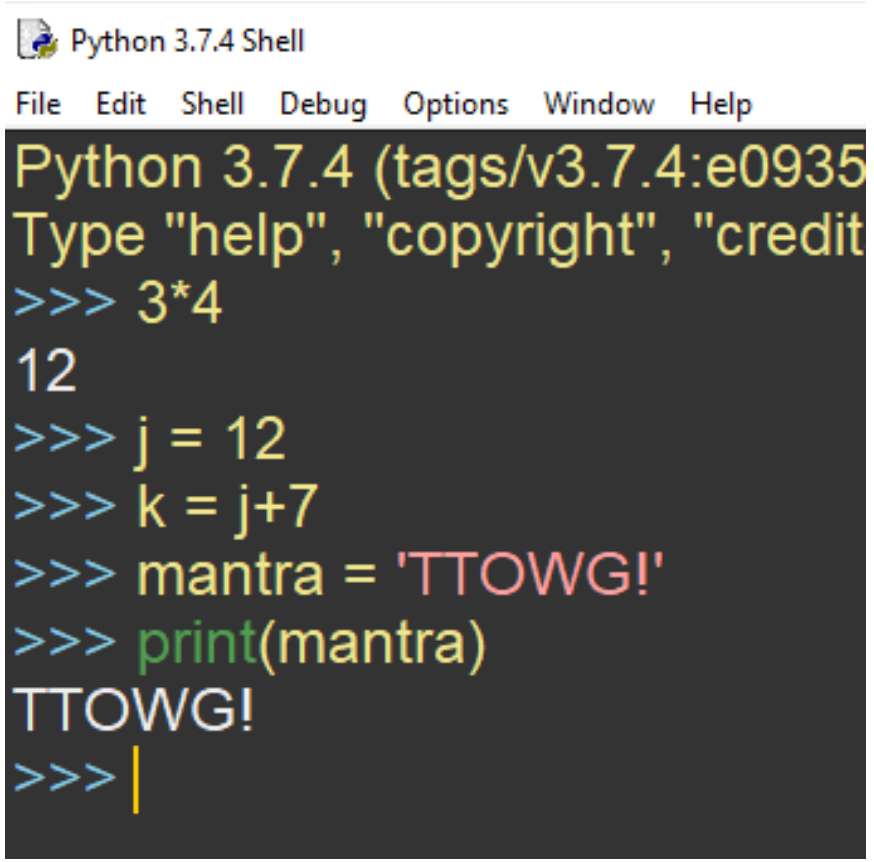
- A variable is a value stored in memory and referred to with a chosen name.
- In other words, values are assigned to variables.
- When the name of a variable is called, the value assigned therein answers.

GETTING STARTED WITH PYTHON

Basic Python Objects

Variables

- A raw value can be assigned to a variable.
  - Example: j is a variable; the value 12 is assigned to it.
- Also, the output of an expression (involving a variable) may be stored in another variable.
  - Example: k is a variable, the value obtained when j+7 is executed is subsequently assigned to variable k.
- Not only numeric values are assigned to variables, strings are also assigned.
  - Example, mantra is a variable with string 'TTOWG!' assigned to it.



Basic Python Objects

Choosing Variable Names

- In naming variable, the following rules are recommended:
  - Variable names should be descriptive, as much as possible. That is, the name should somewhat tell us something about the variable. Example: a variable to hold the value of reservoir permeability is better named 'perm' than named 'x'
  - The name must be a single word. Where multiple words are necessary for descriptive purposes, they can be joined with the underscore character; e. g.: init_pressure.
  - Names should not be too long.
  - Names may contain both alphabets and numbers; but must not start with numbers.
  - Names are case sensitive. If you named a variable as 'poro', do not refer to it as 'Poro'.
  - Avoid using special characters like '@', '$' in names.

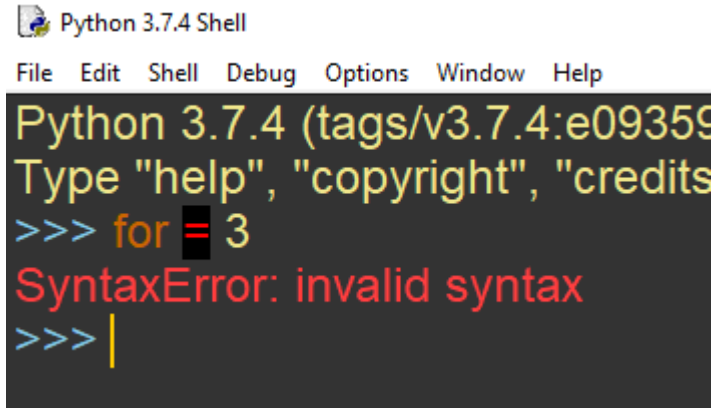
GETTING STARTED WITH PYTHON

Basic Python Objects

Keywords

- Keywords are words that are reserved for Python's in-built structure.
- Here is the list of Python's keywords.
- Keywords cannot be used as variable names; doing so would cause error.

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise     async
def       for       lambda    return    await
```



GETTING STARTED WITH PYTHON

Basic Python Objects

Statements

- A statement is simply unit of code (commands) that is interpretable and executable by Python; just like a sentence in natural language.
- Two common types of Python statements are Assignment statements and Expressions.
- Assignment statements simply assigns values to a variable.
- An expression is a statement that combines variables, values, functions and operators.
- A statement could combine both types such that the result of an expression (RHS) is assigned to a variable (LHS).

```
poro = 0.27 # This is an assignment statement.

area = 40 # This is an assignment statement.

thickness = 15 # This is an assignment statement.

area*thickness # This is an expression.

PV = area*thickness*poro # A combination of expression (RHS) and assignment
```

GETTING STARTED WITH PYTHON

Basic Python Objects

Multi-line statements

- Typically, a Python statement is written in a single line.
- However, if the statement is too long, it could be continued in the next line; but the current line should end with the line continuation character i.e. \

```
>>> 17+2+9 \
        +3+23
54
>>>
```

Multiple statements in a line

- Writing multiple statements in same line is not encouraged; however, if that has to be done, the statements should be separated by semicolon.

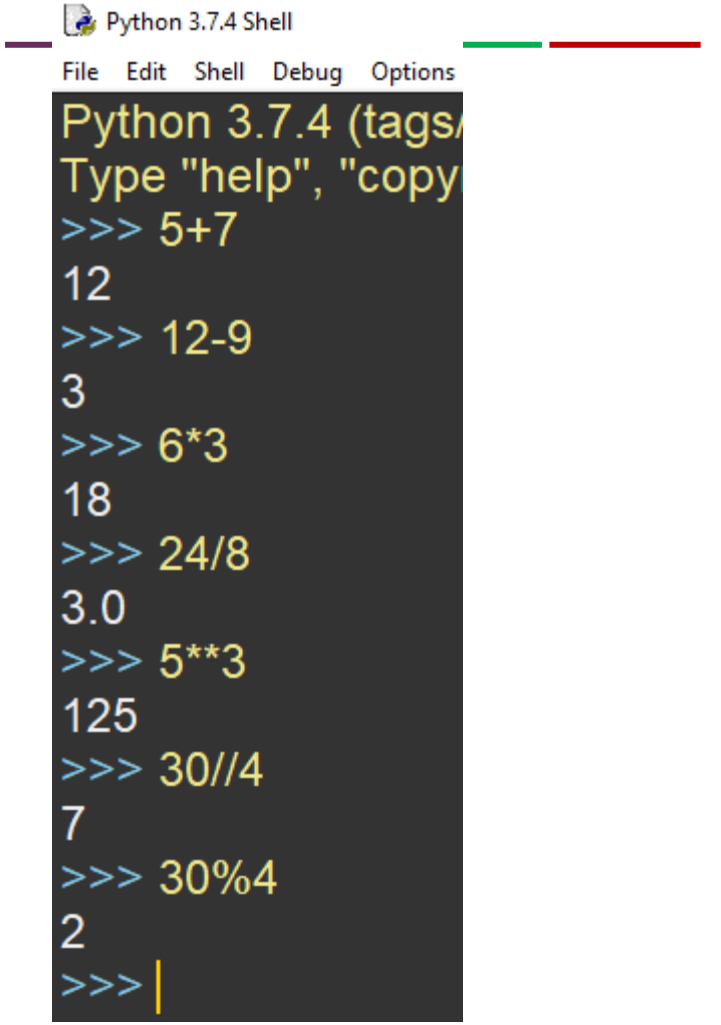
```
>>> poro = 0.18; area = 40; thickness = 15
>>> print(area)
40
>>>
```

GETTING STARTED WITH PYTHON

Basic Python Objects

Operators

- Operators are symbols of mathematical operations.
  - + for addition
  - - for subtraction
  - * for multiplication
  - / for division
  - ** for exponentiation (raise to power)
  - // integer division (truncates the result of division to its integer part.
  - % modulus (gives the remainder of an integer division).
- The objects acted upon by operators are called operands.



Basic Python Objects

Order of Operations

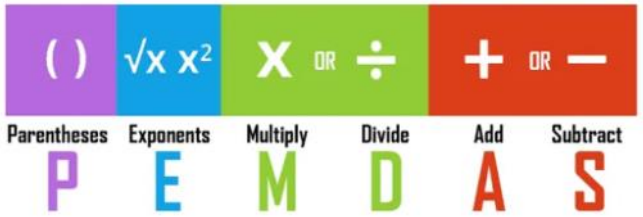
- When multiple operations are featured in a statement, Python executes them in the order specified by the accronym: PE-MD-AS (Parenthesis, Exponentiation, Multiplication, Division, and Subtraction).
- You can used parenthesis to dictate the order you desire.
- Multiplication and Division has equal precedence; hence are executed left to right.
- Addition and Subtraction has equal precedence; hence are executed left to right.
- You may also use parenthesis to make an expression more readable and less confusing.
- Nested parenthesis are executed from inside to outside.

GETTING STARTED WITH PYTHON

Basic Python Objects

Order of Operations

- Consider the following operations to convince yourself of the PEMDAS order.



$$3 \times (3+7) - 4^2 \div 2$$
$$= 3 \times 10 - 4^2 \div 2$$
$$= 3 \times 10 - 16 \div 2$$
$$= 30 - 8$$
$$= 22$$

Python 3.7.4 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.19
Type "help", "copyright", "credits" or "license()" for more information.
>>> (3+5)**(9-6)
512
>>> 14+(3+5)**(9-6)
526
>>> # No, I mean the result of 14+(3+5) should be raised to power 9-6
>>> # Oh! Use parenthesis to dictate that order:
>>> (14+(3+5))**(9-6)
10648
>>> 14+(3+5)**(9-6)/10
65.2
>>> (14+(3+5)**(9-6))/10
52.6
>>> (14+(3+5))**(9-6)/10
1064.8
>>>
```

GETTING STARTED WITH PYTHON

Basic Python Objects

String Operations

- Strings can be joined end-to-end by using the + operator. If you want a space between the strings, then include it in one of the strings.
- Also, a string can be repeated multiple times using the * operator (with an integer, of course).

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19
Type "help", "copyright", "credits" or "license()" for mo
>>> 'TTOWG!' + 'to the only wise God'
'TTOWG!to the only wise God'
>>> # Oh, I need space
>>> 'TTOWG! ' + 'to the only wise God'
'TTOWG! to the only wise God'
>>>
>>> 'TTOWG!'*3
'TTOWG!TTOWG!TTOWG!'
>>> 'TTOWG! '*3
'TTOWG! TTOWG! TTOWG! '
>>> 3*'TTOWG! '
'TTOWG! TTOWG! TTOWG! '
>>>
```

GETTING STARTED WITH PYTHON

Conditional Statements

- Conditional statements are written to make it possible for a program to check for some condition(s) and decide to either:
  - perform a statement(s) or skip the statement(s)
    - or
  - choose between alternative (branches of) statements.
- So, the concept of condition is central to this kind of statements.
- These conditions are crafted using the concept of Boolean expressions.

GETTING STARTED WITH PYTHON

Conditional Statements

Boolean Expressions

- A Boolean is a type of value; it can only be either True or False
- Just like the integer type can take values 1, 2, 3 e.t.c; the Boolean type can take one of just two values: True or False.
- For this reason, 'True' and 'False' are Python keywords reserved for Boolean values; a variable must not be named using these words.
- Now, a Boolean expression is essentially a comparison expression that evaluates to either True or False.

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>>
>>> 2<7
True
>>> 2>7
False
>>>
```

GETTING STARTED WITH PYTHON

Conditional Statements

Boolean Expressions

- Boolean expressions are constructed using comparison operators listed here.

- Take note that = is an assignment operator while == is a comparison operator.

```
>>> init_press = 4000
>>> bubble_press = 2800
>>>
>>> init_press == bubble_press  # == denotes equal to
False
>>> init_press != bubble_press  # != denotes not equal to
True
>>> init_press > bubble_press  # > denotes greater than
True
>>> init_press < bubble_press  # < denotes greater than
False
>>> init_press >= 4200  # >= denotes greater than or equal to
False
>>> init_press <= 4200  # <= denotes less than or equal to
True
>>> init_press is bubble_press  # is denotes the same as
False
>>> init_press is 4000  # is denotes the same as
False
>>> init_press is not bubble_press  # is not denotes not the same as
True
```

GETTING STARTED WITH PYTHON

Conditional Statements

Logical Operators

- Sometimes, multiple conditions needed to be checked in a conditional statement.

```
>>> 2<3 and 7>5
True
>>> 2<3 and 7<5
False
>>> 2<3 or 7<5
True
>>> not(7<5)
True
>>>
```

- Logical operators are used to combine Boolean expressions
  - *and* returns True if all conditions are true, otherwise, False is returned.
  - *or* returns True if at least one of the conditions is true, otherwise, False is returned.

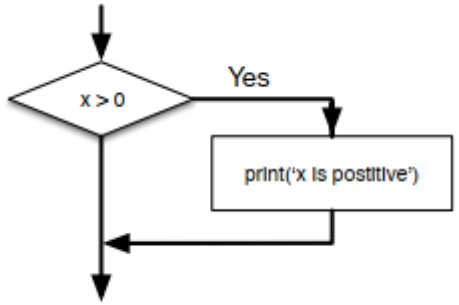
- Logical operators are also used to negate Boolean expressions
  - *not* returns True for a false condition and vice-versa

GETTING STARTED WITH PYTHON

Conditional Statements

if… Statement (Conditional Execution)

- if statements evaluates the given condition; performs the given statement(s) if condition is true and skips the given statement(s) if condition is false.
- The condition(s) is written after the *if* keyword and ended with a colon i.e. (:)
- The statements to be performed or skipped are written as an indented block in subsequent line(s).
- Remove the indentation in lines after the if block.




```
>>> if perm > 50:
        print('Good permeability')


Good permeability
```

Conditional Statements
if… Statement (Conditional Execution)

Petroleum engineering application

- Computing pseudo-critical gas properties using Sutton's correlation
  - Sutton developed a correlation for estimating for estimating P_{pc} and T_{pc} as functions of gas gravity.
  - Here is the first step in Sutton's procedure:
    - If the gas mixture contains <12 mol% of CO_2, < 3% of Nitrogen and no H_2S, then the parameter γ_h takes the same value as the given gas gravity; no need for correction.
    - However, if gas mixture contains >12 mol% of CO_2 OR >3% of Nitrogen OR any H_2S, then the parameter γ_h is determined thus:

$$\gamma_h = \frac{\gamma_w - 1.1767 y_{H_2S} - 1.5196 y_{CO_2} - 0.9672 y_{N_2} - 0.622 y_{H_2O}}{1 - y_{H_2S} - y_{CO_2} - y_{N_2} - y_{H_2O}}.$$

Conditional Statements
if… Statement (Conditional Execution)

Petroleum engineering application

- Computing pseudo-critical gas properties using Sutton's correlation
  - The first step in Sutton's can be executed with an if … statement.
  - Observe that the procedure implies that if any of the impurities in the gas exceeds the stated threshold value, then, the given gas gravity (γ_w) need to be corrected for the effects of the impurities, using the given equation.
  - However, the correction task should be neglected if none of the impurities exceeds its threshold value.

Conditional Statements
if… Statement (Conditional Execution)

Petroleum engineering application

- Computing pseudo-critical gas properties using Sutton's correlation
  - To execute this procedure, we simply construct a Boolean condition to test if any threshold is violated.
  - If the condition is evaluated as True, then a block of statement to perform the gas gravity correction is executed.
  - If the condition is evaluated to be False, there is no need for the correction, hence, the block of statement is skipped.

Conditional Statements
if… Statement (Conditional Execution)

Petroleum engineering application

- Computing pseudo-critical gas properties using Sutton's correlation

```
if co2_comp > 0.12 or n2_comp > 0.03 or h2s_comp > 0:
    gas_gravity = (gas_gravity - (1.1767*h2s_comp) - \
                  (1.5196*co2_comp) - (0.9672*n2_comp) - \
                   (0.622*h2o_comp))/(1- h2s_comp - co2_comp - n2_comp - h2o_comp)
    print('The corrected gas gravity is', gas_gravity)
```

The full script for this computation is available here

GETTING STARTED WITH PYTHON

Conditional Statements

if… Statement (Conditional Execution)

Assignment 2

Upgrade the *demo_gas_grav_corr.py* script (hosted on *TTOWG/ PET328_2021_Class* GitHub repository) to perform the entire Sutton's procedure. Save the upgraded script as *sutton_correlation.py*, commit and push it to your GitHub repository. Submit the URL to your copy of *PET328_2021_Class* repository. Furthermore, send a pull request to the original *TTOWG/ PET328_2021_Class* repository.

- The complete Sutton's algorithm is available here.

Conditional Statements

if… Statement (Conditional Execution)

Assignment 2

You may test run your script with the following data:

- **Inputs**
  - y_{CO2} = 0.0164
  - y_{N2} = 0.0236
  - y_{H2S} = 0.1841
  - Gas gravity = 0.6992

Outputs

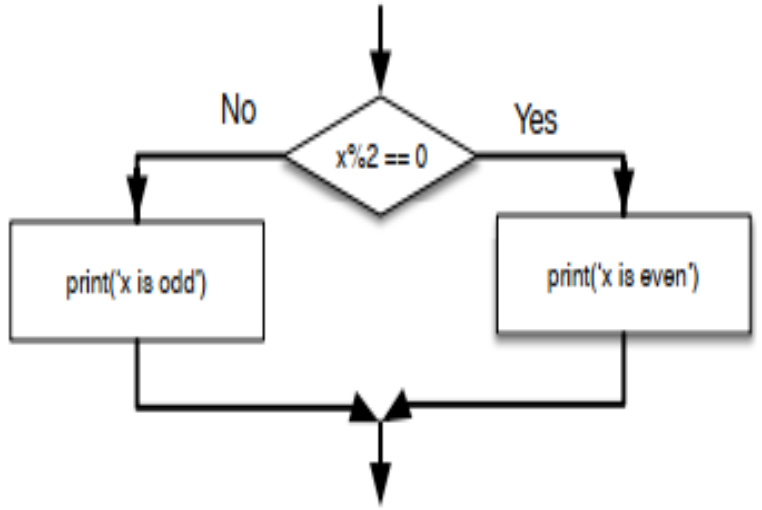
- Corrected gas gravity = 0.5604
- Ppch = 682.3 psia.
- Tpch = 341.8 deg Rankine
- Ppc = 799.0 psia.
- Tpc = 403.3 deg Rankine

GETTING STARTED WITH PYTHON

Conditional Statements

if…then…else Statement (Alternative Execution)

- The if…then…else structure is deployed when there are two alternative tasks and a condition that determines which of the two alternatives should be executed.
- Essentially, there will be a Boolean condition, and two blocks (branches) of statements.
- The first branch (after the condition) is to be executed if the condition evaluates to True while the second branch (after the keyword 'else') is executed if the condition evaluates to False.

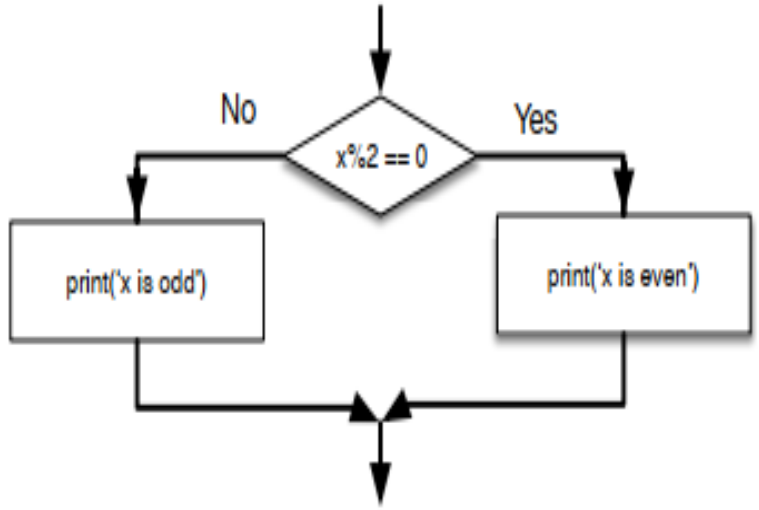


GETTING STARTED WITH PYTHON

Conditional Statements

if…then…else Statement (Alternative Execution)

- The condition(s) is written after the *if* keyword and ended with a colon i.e. (:)
- Then, the statement(s) to be performed if condition is True (i.e., Branch True) are written as an indented block in subsequent line(s).
- Thereafter, the keyword 'else' is written on the next line just after the Branch True. The 'else' keyword should be indented to the same level as the 'if' keyword.
- Finally, the statement(s) to be performed if condition is False (i.e., Branch False) are written as an indented block in subsequent line(s).



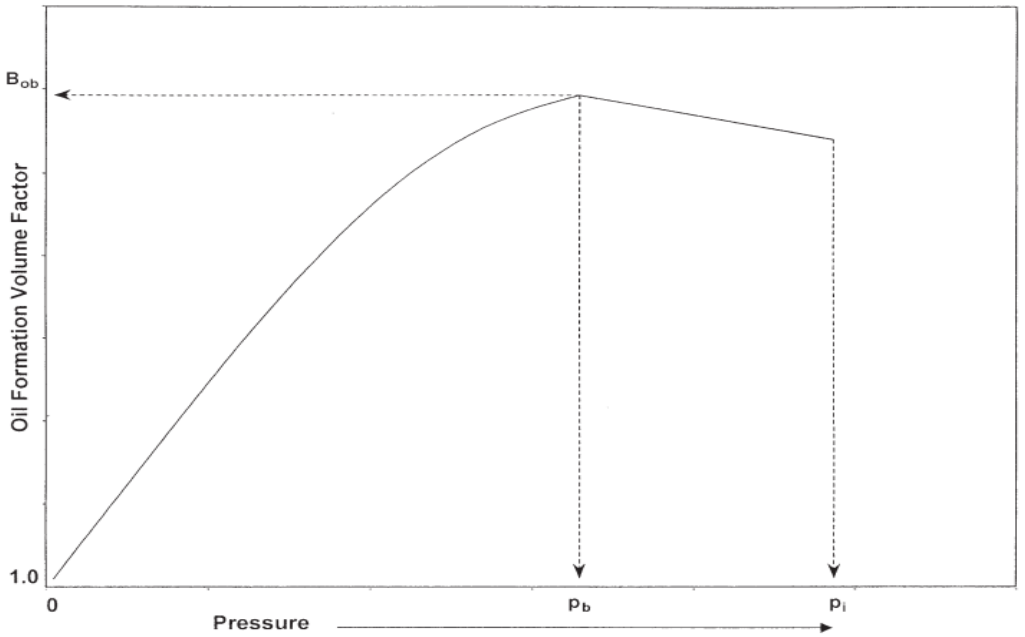

```
>>> perm = 90
>>> if perm < 50:
        print('Fair!')
else:
        print('Good!')


Good!
>>>
```

GETTING STARTED WITH PYTHON

Conditional Statements
if…then…else Statement (Alternative Execution)

Petroleum engineering application

- Computing oil formation volume factor, Bo.
  - The variation of oil formation volume factor, Bo, with pressure is divided into two pressure regimes, as shown.



Conditional Statements

if…then…else Statement (Alternative Execution)

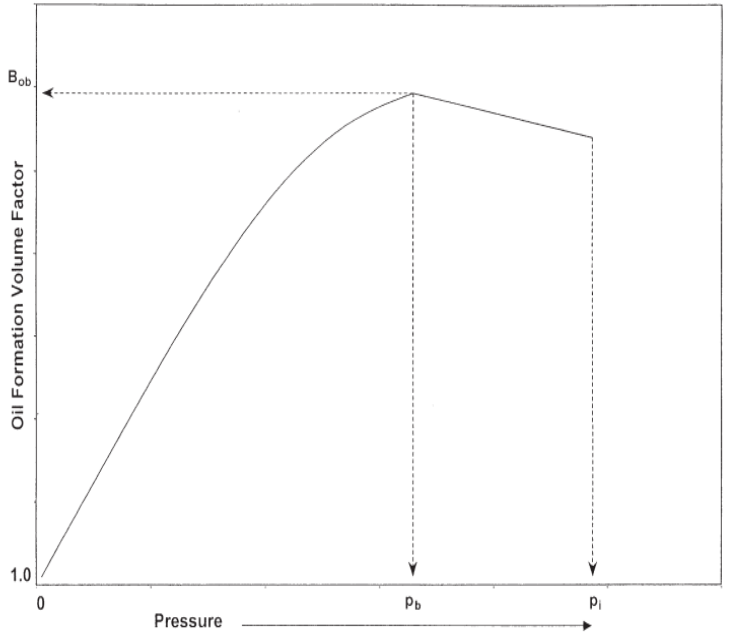
Petroleum engineering application

- Computing oil formation volume factor, Bo.

For pressures below or equal to bubble point, Standing's correlation for calculating Bo is herein presented:

$$B_o = 0.9759 + 0.00012F^{1.2} \text{ ----- } -2.35$$

$$Where\ F = R_s \left(\frac{\gamma_g}{\gamma_o}\right)^{0.5} + 1.25T_F \text{ -- } -2.36$$



Note: T_F is temperature in degree Fahrenheit.

Conditional Statements
if…then…else Statement (Alternative Execution)

Petroleum engineering application

- Computing oil formation volume factor, Bo.

For pressure above bubble point, the analytical equation for computing Bo is given as:

$$B_o = B_{ob} e^{[c_o(P_b - P)]} \text{ ---} -2.37$$

Bob is the Bo at bubble point and can be computed using Equations 2.35 and 2.36

Conditional Statements

if…then…else Statement (Alternative Execution)

Petroleum engineering application

- Computing oil formation volume factor, Bo.
  - To execute this procedure, we simply construct a Boolean condition to test if the current reservoir pressure, p, is greater than the bubble-point pressure of the reservoir.
  - If the condition is evaluated as True, then a block of statement to implement Equation 2.37 is executed.
  - Else, if the condition is evaluated to be False, then a block of statement to implement Equation 2.35 is executed.
  - Note that Equation 2.36 need to be implemented for either of the alternatives, hence, the line to execute it is written before the if…then…else statement.

Conditional Statements

if…then…else Statement (Alternative Execution)

Petroleum engineering application

- Computing oil formation volume factor, Bo.

```
# calculating F parameter
F = (rs*((gas_gravity/oil_gravity)**0.5))+(1.25*tf)

# the if-then-else statement

if p > pb:
    bob = 0.9759+(0.00012*(F**1.2))
    bo = bob*(math.exp(co*(pb-p)))
else:
    bo = 0.9759+(0.00012*(F**1.2))
```

The full script for this computation is available here.

Conditional Statements
Reading Assignment

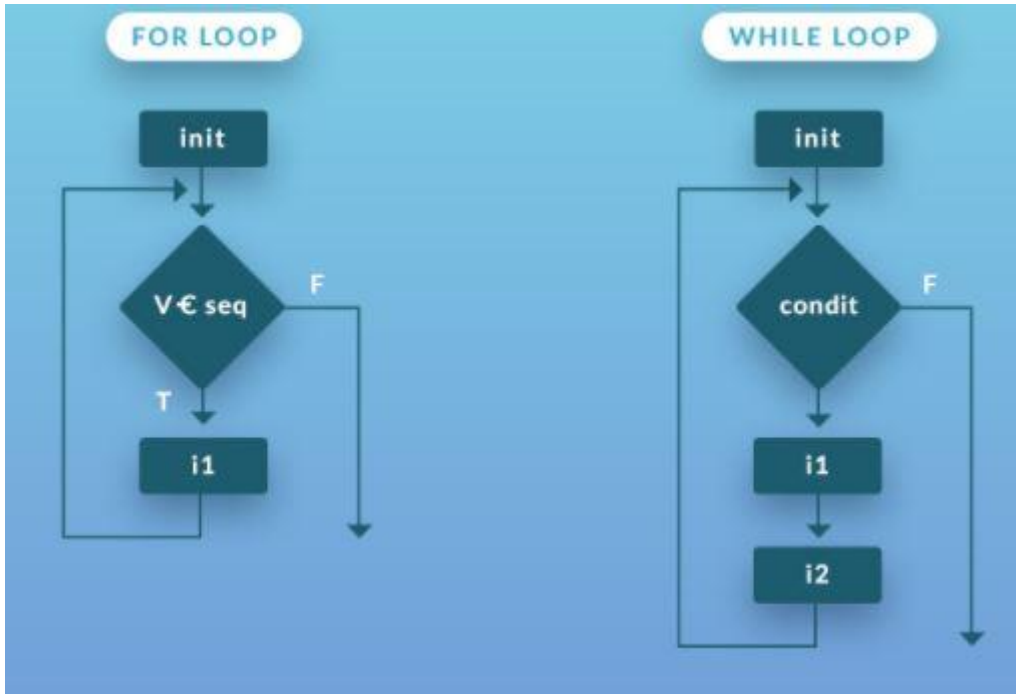
- Read on Chained Conditionals (if…elif… statements) and Nested Conditionals, in the recommended textbook for this course (pages 34 – 36)

Recommended Textbook: *Python for Everybody: Exploring Data using Python 3*, by Charles Severance.

GETTING STARTED WITH PYTHON

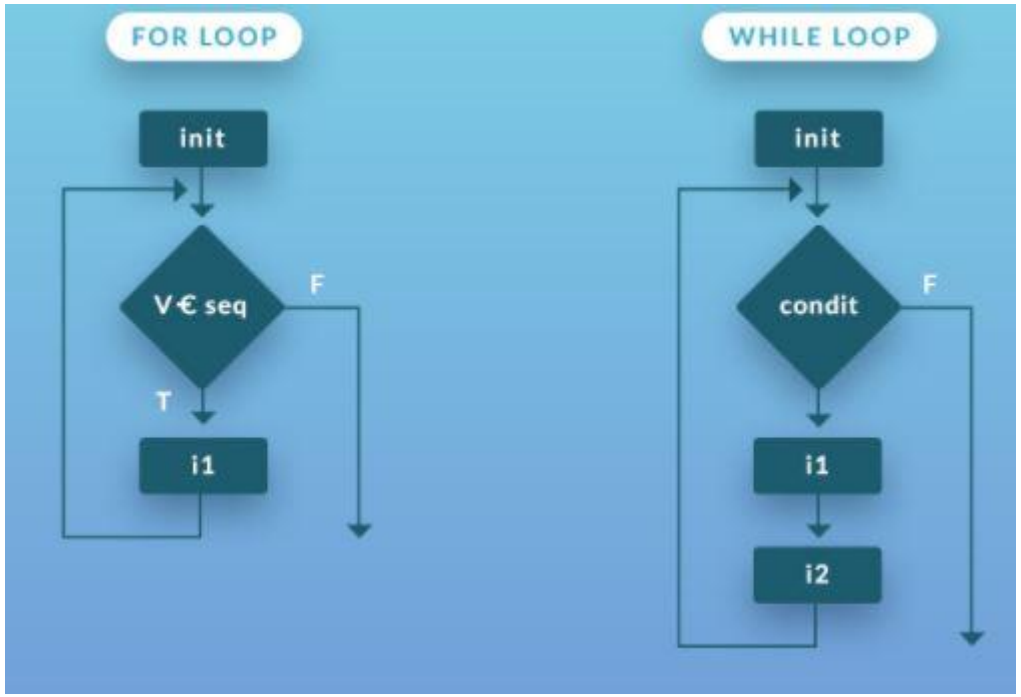
Repeated Execution

- One major reason for writing computer programs is to automate repetitive workflows.
- When a given task is to be performed repeatedly for each member of a set of entities, that is a repetitive workflow.
- In implementing such workflows, moving from one entity to the next entity is called looping or iterating.



Repeated Execution

- Sometimes, the list of entities in the set is known explicitly and presented to the computer – that is a definite loop.
- At other times, the list is not known explicitly, rather the computer is asked to perform the task repeatedly until a given condition becomes False – that is an indefinite loop.
- Definite loops are implemented with 'for' loops and indefinite loops are implemented with 'while' loops.



GETTING STARTED WITH PYTHON

Repeated Execution
for… loops

- In practice, 'for' loops are used in counting the number of items/elements in a list or in summing up (aggregating) the results of a computation for every item/element in a list.
- In such cases, a variable to hold the count or sum is normally created and set to a dummy initial value before the 'for' loop, and is updated inside the loop.
- The first line (header) of a 'for' loop begins with the keyword 'for', followed by an iterator variable, followed by the keyword 'in' and lastly, the list of items on which the task is to be performed.
- Statement(s) to implement the task are written as an indented block on subsequent lines after the header.

```
>>> blocks = [1,2,3,4,5]
>>> for block in blocks:
        print('This is Block', block)


This is Block 1
This is Block 2
This is Block 3
This is Block 4
This is Block 5
>>>
```

GETTING STARTED WITH PYTHON

Repeated Execution
for… loops

Iterator variable

- The iterator variable is a sort of temporary variable that represent the item/element of the set being treated in the current loop iteration/cycle.
- So, the iterator variable changes value as the program loops through the list of items.

- Sometimes, the in-built function *range* is used to generate the list of items.

```
>>> blocks = [1,2,3,4,5]
>>> for block in blocks:
        print('This is Block', block)


This is Block 1
This is Block 2
This is Block 3
This is Block 4
This is Block 5
>>>
```

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(1,5))
[1, 2, 3, 4]
>>> list(range(1,6))
[1, 2, 3, 4, 5]
>>>
```

```
>>> for point in range(1,6):
        print('This is Point', point)


This is Point 1
This is Point 2
This is Point 3
This is Point 4
This is Point 5
```

Repeated Execution
for… loops

Petroleum engineering application

- Reservoir discretization
  - Input parameters required to solve reservoir engineering models are essentially rock and fluid properties.
  - These rock and fluid properties are known to vary across the reservoir – heterogeneity.
  - The challenge: which value of a property is to be used in solving the model for a given reservoir???
  - Average??? Nay!
  - Locally-acceptable values??? Yes!

GETTING STARTED WITH PYTHON

Repeated Execution

for… loops

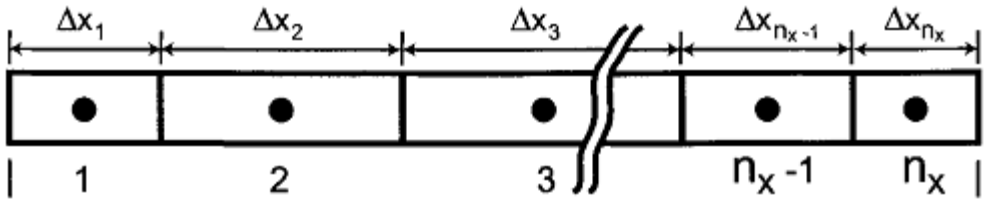
Petroleum engineering application

Reservoir discretization

- Discretization is the means by which locally-acceptable values of reservoir rock and fluid properties are honored in reservoir modelling.

- Loosely speaking, reservoir discretization is the division of the reservoirs into grid-blocks whose properties, dimensions and locations are well defined and uniform.

- Upon discretizing the reservoir into blocks; the flow model would then need be written and solved, repeatedly, for each block.



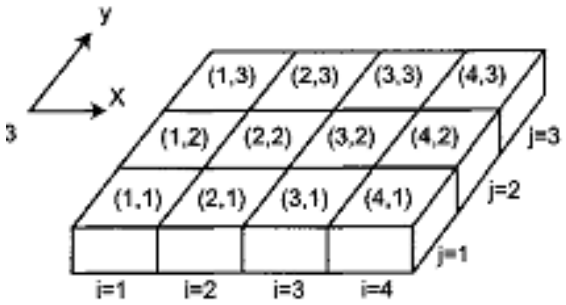
Repeated Execution
for… loops
Petroleum engineering application

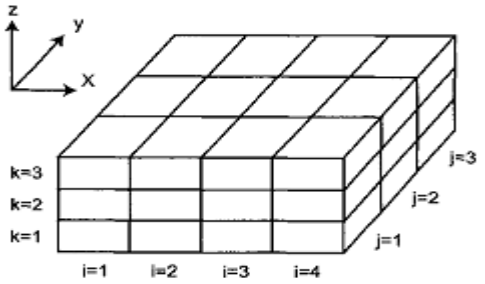
- Reservoir discretization
- Blocks Ordering (Numbering) Schemes –
  - a way to identify each block in 1D, 2D or 3D discretized model.
  - Two types of ordering:
    - Engineering ordering.
    - Natural ordering

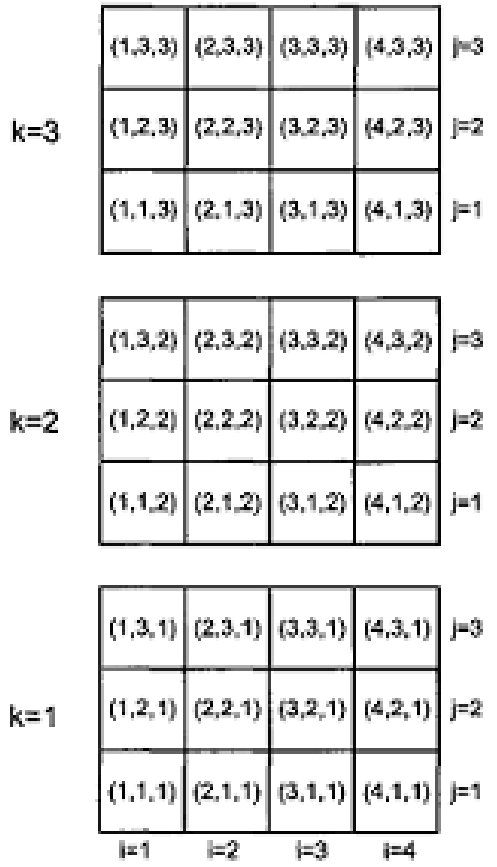
GETTING STARTED WITH PYTHON

Repeated Execution
for… loops
Petroleum engineering application

- Reservoir discretization
- Engineering Ordering –
  - uses i, j, k, notations to order blocks in the x, y, z directions respectively.
    - i – counts columns along a certain row;
    - j – counts rows along a certain column;
    - k refers to layers.



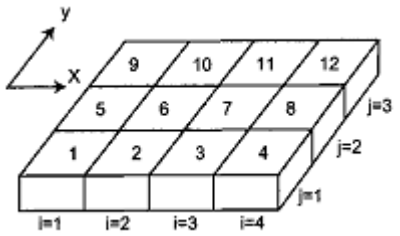


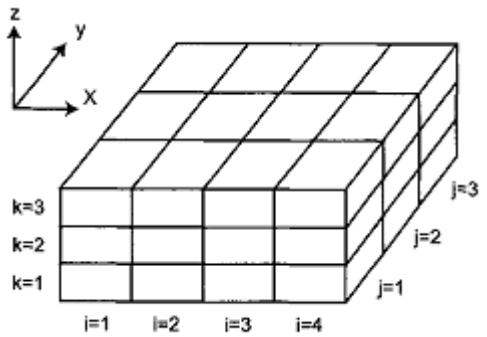


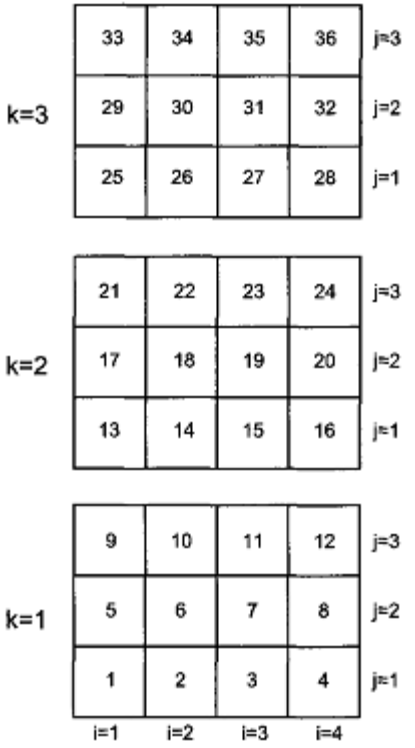
GETTING STARTED WITH PYTHON

Repeated Execution
for… loops
Petroleum engineering application

- Reservoir discretization
- Natural Ordering –
  - uses natural counting scheme.
  - Columns are counted fastest, followed by rows and then layers.







(c) Natural ordering of blocks.

Repeated Execution
for… loops

Petroleum engineering application

- Reservoir discretization
- The engineering ordering fits perfectly well into the 'for' loop scheme.
  - Looping through columns in a given row would be implemented by a column 'for' loop whose iterator variable would be *i* (the column counter).
  - Looping through rows in a given layer would be implemented by a row 'for' loop whose iterator variable would be *j* (the row counter).
  - Looping through layers would be implemented by a layer 'for' loop whose iterator variable would be *k* (the layer counter).

Repeated Execution
for… loops

Petroleum engineering application

- Reservoir discretization
- In a one-dimensional discretized model (i.e. a model discretized in only one direction), only one 'for' loop is needed; it may be column, row or layer loop.
- In a multi-dimensional discretized model (i.e. a model discretized in multiple directions), multiple nested 'for' loops are needed; with the column loop cycling faster than the row loop and the row loop cycling faster than the layer loop.
- In nested loop, the innermost loop cycles fastest while the outermost loop cycles the slowest.

Repeated Execution
for… loops

Petroleum engineering application

Reservoir discretization

- While the engineering ordering is used in computations, communicating outputs of such computations is better done with natural ordering.
- The following equation can be used to obtain the natural ordering index of a block from its engineering ordering indices (i, j, k)

$$n_{order} = [(k-1)n_x.n_y] + [(j-1)n_x] + i$$

  - n_{order} = natural order index of the block
  - i is the column index of the block
  - j is the row index of the block
  - k is the layer index of the block
  - n_x is the number of blocks in x-direction (i.e. number of columns)
  - n_y is the number of blocks in y-direction
- A constant value of 1 is used for any of the i, j, k indices not relevant to a case at hand.

GETTING STARTED WITH PYTHON

Repeated Execution

for… loops

Petroleum engineering application

- Computing STOIIP for a discretized reservoir model
- The amount of stock tank oil in-place in a given reservoir is given thus:

$$STOIIP = \frac{7758Ah\phi(1 - S_{wi})}{B_{oi}}$$

- Typically, values of porosity (ϕ)and initial water saturation (Swi) varies across the reservoir; hence, the given reservoir is discretized and the STOIIP equation is implemented for each block of the discretized model. The STOIIP for each is block aggregated in a running total to yield the reservoir STOIIP ultimately.

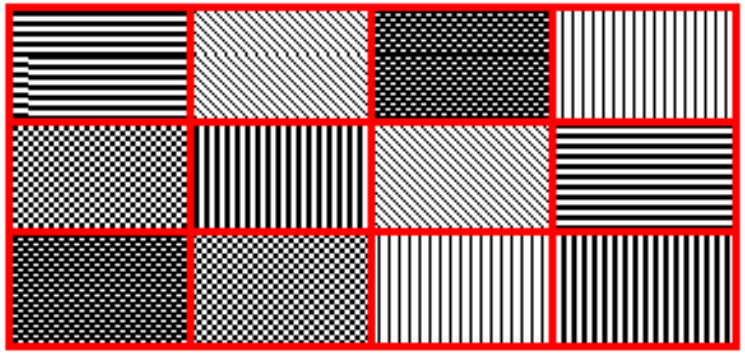
GETTING STARTED WITH PYTHON

Repeated Execution

for… loops

Petroleum engineering application

- Computing STOIIP for a discretized reservoir model
- Here is a simple (trivial) 2D example of the grid of such discretized reservoir models.



Legend	poro	swi
	0.1	0.23
	0.25	0.29
	0.29	0.31
	0.33	0.37
	0.23	0.20
	0.27	0.28

GETTING STARTED WITH PYTHON

Repeated Execution
for… loops

Petroleum engineering application

- Computing STOIIP for a discretized reservoir model

```
# the 'for' loop

for j in range(1,ny+1):
    for i in range(1,nx+1):
        block_n_order = (nx*(j-1))+i
        poro = float(input('What is the value of porosity for Block {0}?'.format(block_n_order)))
        sw = float(input('What is the value of water saturation for Block {0}?'.format(block_n_order)))
        block_stoiip = (7758*area*h*poro*(1-sw))/boi
        total_stoiip = total_stoiip + block_stoiip
        print('The amount of oil in Block {0} is {1:.2f} STB'.format(block_n_order, block_stoiip))
```

The full script for this computation is available here.

Disclaimer: the implementation of this task as presented in the script is only for pedagogical purposes; in reality, a more efficient implementation would be done.

GETTING STARTED WITH PYTHON

Repeated Execution
for… loops

Assignment 3

An undersaturated oil reservoir material balance simulator computes the cumulative oil produced from a block, in a discretized model, thus:

$$N_p = \frac{NB_{oi}c_e(P_i - P_{now})}{B_o} \quad --------- -P5$$

- N_p is the cumulative oil produced from a given block.
- N is the initial oil in-place (STOIIP) for each block (assumed constant for all blocks)
- B_{oi} is the initial oil formation volume factor (assumed constant for all blocks)
- c_e is the effective compressibility (assumed constant for all blocks)
- P_i is the initial reservoir pressure (assumed constant for all blocks)
- P_{now} is the current reservoir pressure (varies across blocks depending on proximity to producer well)
- B_o is the current value of the oil formation volume factor (depends on current pressure in a block)

GETTING STARTED WITH PYTHON

Repeated Execution
for… loops

Assignment 3

- Below is the expression to calculate the B_o value corresponding to a given current pressure.

$$B_o = B_{ob}[1 - c_o(P_{now} - P_b)] \text{ ---------- } -P6$$

- Given a set of parameter (N_p, B_{oi}, B_{ob}, c_e, c_o and P_i) values and a grid of current pressure values, write a Python script to implement Equations P5 and P6 for each block. Also include statements to sum up and present the total cumulative oil produced from the entire reservoir.

- Save the script as *mat_bal.py*, commit and push it to your GitHub repository. Submit the URL to your copy of *PET328_2021_Class* repository. Furthermore, send a pull request to the original *TTOWG/ PET328_2021_Class* repository.

Repeated Execution
for… loops

Assignment 3

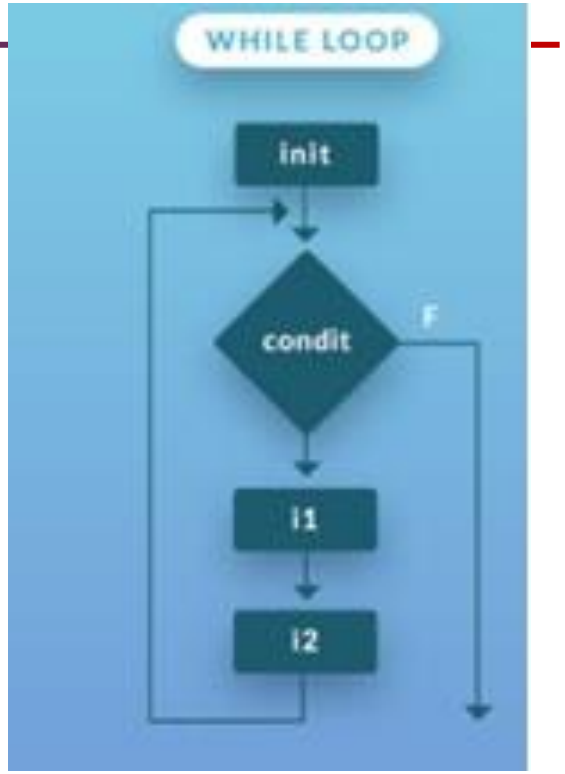
- You may test your script with the following data:
  - **Parameters**
    - N = 200,779.157 STB
    - P_i = 4025 psi
    - P_b = 3330 psi
    - B_{oi} = 1.2417 RB/STB
    - B_{ob} = 1.2511RB/STB
    - c_e = co = 0.0000113 psi^{-1}

Current pressure values in grid:

4018.913	4018.875	4018.802	4018.699
4018.905	4018.866	4018.79	4018.682
4018.89	4018.848	4018.765	4018.648

GETTING STARTED WITH PYTHON

Repeated Execution
while... loops

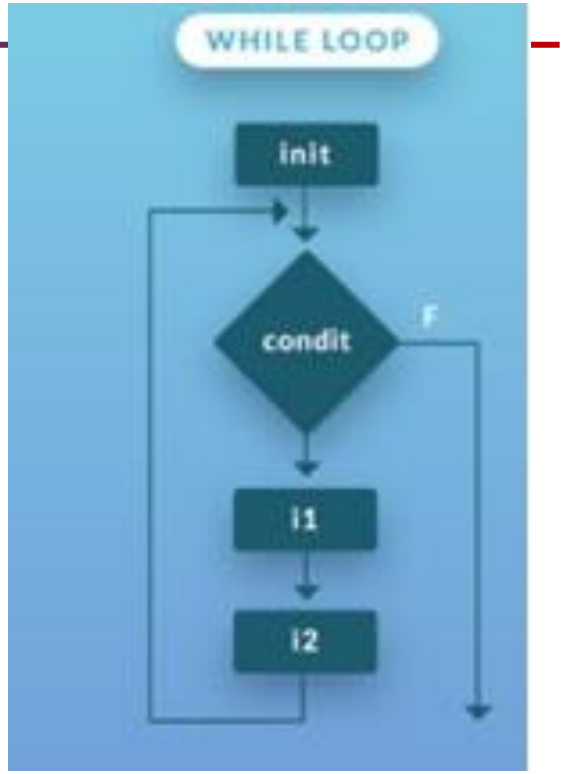


- Indefinite loops are implemented with 'while' loops.
- Unlike 'for' loops, a definite list of items to be acted upon is not presented to 'while' loops.
- Rather, 'while' loops performs a given block of statement(s) until a specified condition is evaluated to be False; then it stops.
- The loop runs if the specified condition is True, just like an 'if' statement.
- However, unlike an 'if' statement, the 'while' loop repeats the block of statement(s).

GETTING STARTED WITH PYTHON

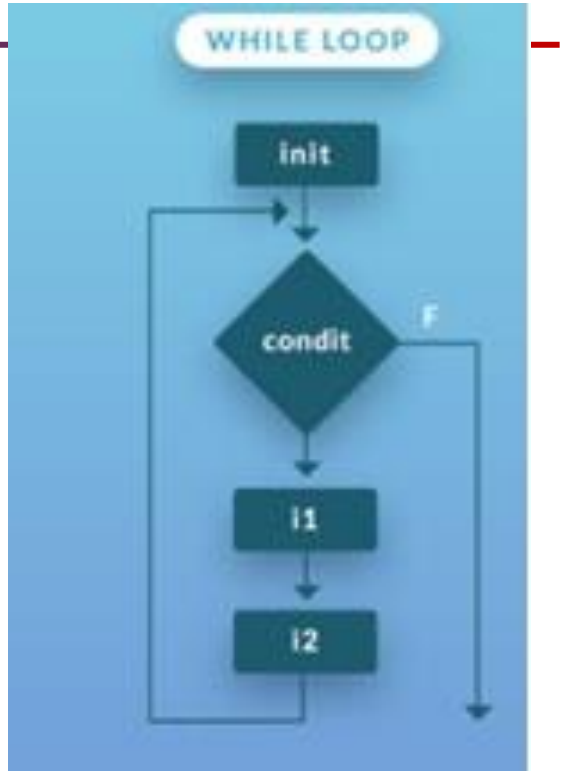
Repeated Execution
while... loops

- The execution of 'while' loops goes thus:
  - Evaluate the specified condition
  - If condition is False, exit the loop and continue with statements(s) outside (below) the loop.
  - Execute the block of statement(s) if condition is true and return to evaluation step



GETTING STARTED WITH PYTHON

Repeated Execution
while... loops

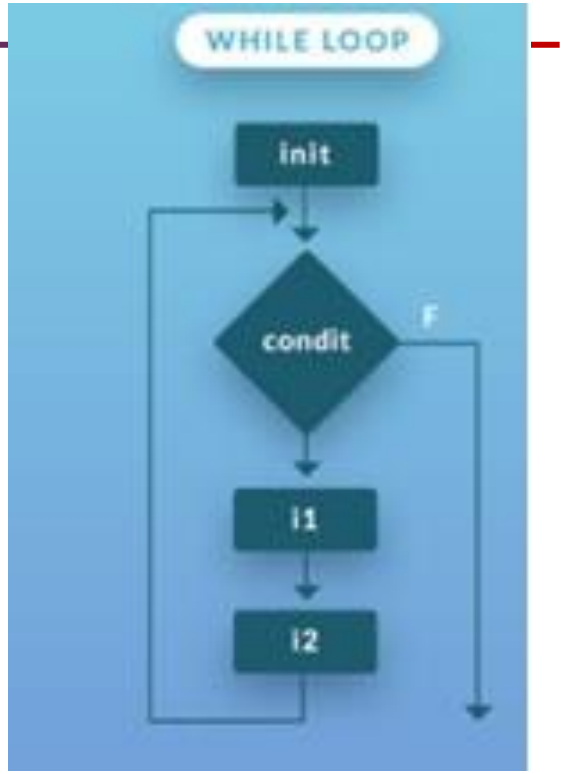


- The 'while' loop condition is constructed as Boolean expression.
- Typically, the condition involves an iterator variable that is normally initialized outside (before) the loop and gets updated in the loop (during each iteration).
- It is by such changes (updates) to the value of the iterator variable that the condition would eventually turn False and the loop gets terminated.

GETTING STARTED WITH PYTHON

Repeated Execution
while... loops

- When the iteration variable is not specified or not updated, the loop runs forever – infinite loops.
-
- Reading Assignment:
  - Read on Infinite Loops, 'break' and 'continue' statements in the textbook recommended for this course.



Repeated Execution
while... Loops

Petroleum engineering applications

- The 'while' loops finds applications in cases where a task need to be performed for a series of dynamically-changing values of a variable (like pressure) until that variable violates a specified threshold value.

- For instance, in scripting an undersaturated oil reservoir simulator, computations are to be performed for series of decreasing pressure values until pressure reduces to a value below bubble-point pressure.

Repeated Execution
while... Loops

Petroleum engineering applications

- See a trivial example here.

```
>>> current_pressure = 4000
>>> bubble_pressure = 2800
>>> while current_pressure > bubble_pressure:
        print('Pressure {0} is still undersaturated, perform computations'.format(current_pressure))
        current_pressure = current_pressure - 200


Pressure 4000 is still undersaturated, perform computations
Pressure 3800 is still undersaturated, perform computations
Pressure 3600 is still undersaturated, perform computations
Pressure 3400 is still undersaturated, perform computations
Pressure 3200 is still undersaturated, perform computations
Pressure 3000 is still undersaturated, perform computations
```

GETTING STARTED WITH PYTHON

Repeated Execution

while... Loops

Petroleum engineering applications

- Updating oil formation volume factor (Bo), for undersaturated reservoir simulation.

Recall that, the analytical equation for computing Bo, at pressures above bubble point is given as: (see Slide 75 – 76)

$$B_o = B_{ob} e^{[c_o(P_b - P)]} \quad ---- 2.37$$

- There is a need to write statement that would not just compute Bo at a single pressure value, but at series of decreasing pressure values until pressure decreases to bubble point pressure, at which point, the simulation must be terminated.

GETTING STARTED WITH PYTHON

Repeated Execution
while... Loops

Petroleum engineering applications

- Updating oil formation volume factor (Bo), for undersaturated reservoir simulation.

```
while p > pb:
    bo = bob*(math.exp(co*(pb-p)))
    print('The value of oil FVF at {0:.2f} psi is {1:.4f}'.format(p, bo))
    p = p - pressure_step

# continuing after the 'while' block
print('Bubble-point pressure reached! End of simulation')
```

The full script for this computation is available here.

GETTING STARTED WITH PYTHON

Repeated Execution
while... Loops

Petroleum engineering applications

- Updating oil formation volume factor (Bo), for undersaturated reservoir simulation.

```
 RESTART: C:/Users/TTOWG/645/1 karia def/2. CU/CU Cours
PET328_2021_Class/demo_oil_fvf_updator.py
What is the value of reservoir initial pressure?4000
What is the value of reservoir bubble-point pressure?3330
What is the value of oil FVF at bubble-point pressure?1.2511
What is the value of oil compressibility?0.0000113
By how much should pressure be decremented?50
The value of oil FVF at 4000.00 psi is 1.2417
The value of oil FVF at 3950.00 psi is 1.2424
The value of oil FVF at 3900.00 psi is 1.2431
The value of oil FVF at 3850.00 psi is 1.2438
The value of oil FVF at 3800.00 psi is 1.2445
The value of oil FVF at 3750.00 psi is 1.2452
The value of oil FVF at 3700.00 psi is 1.2459
The value of oil FVF at 3650.00 psi is 1.2466
The value of oil FVF at 3600.00 psi is 1.2473
The value of oil FVF at 3550.00 psi is 1.2480
The value of oil FVF at 3500.00 psi is 1.2487
The value of oil FVF at 3450.00 psi is 1.2494
The value of oil FVF at 3400.00 psi is 1.2501
The value of oil FVF at 3350.00 psi is 1.2508
Bubble-point pressure reached! End of simulation
```

Functions (Reusable Code)

- Sometimes, you write a code (sequence of statements) to perform a task, and you realize you often perform that task; may be in the same program, or in other programs.

- You don't want to keep writing (or copying) same code anytime you need to perform such task.
- You may give the code (sequence of statements) a cute name; so that anytime that task is to be performed, you simply call that name and the entire sequence of statements in the code get executed!
  - Just like you assign a value to a name to make a variable; you assign a sequence of statements to make a function.

- Such a named sequence of code is known as function.

Functions (Reusable Code)

- Benefits of writing functions:
  - Saves typing efforts
  - Reduces the risk of mistakes
  - Help to organize your codes
  - Makes your code more readable

- Take note, functions (named sequence of statements) are objects just as variables (named values) are objects.
- The installed (base)Python itself thrives on a lot of in-built functions such as *print*, *input*, *range*, *min*, *max*, *len*, *float*, *int*, *str* etc.
- Functions created (defined) by users (i.e. not built into Python during installation) are known as user-defined functions.

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Function Definition:
- The first line (header) of a function definition starts with the keyword def, followed by the name of the function (as chosen by you), followed by a comma-separated list of function argument(s) enclosed in parenthesis, and finally, a colon.

- If the function requires no arguments, an empty parenthesis () is typed.

```
#...TTOWG!

# function definition
def stoiip_calc(area, thickness, poro, sw, boi):
    STOIIP = (7758*area*thickness*poro*(1-sw))/boi
    return STOIIP
```

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Function Definition:
- The sequence of statements (to be executed whenever the function is called) is known as the body of the function, and is written in subsequent indented lines after the function's header.

- A statement specifying the value(s) to be returned by the function must be included with the keyword return

```
#...TTOWG!

# function definition
def stoiip_calc(area, thickness, poro, sw, boi):
    STOIIP = (7758*area*thickness*poro*(1-sw))/boi
    return STOIIP
```

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Function Arguments:
- The arguments of a function are the values that that it would need to execute its sequence of statements when called.
- When defining a function, placeholders (variables) corresponding to these arguments are named, listed and used in expressions.
- When the function is called, actual values for the arguments are specified (more on this, soon).

```
#...TTOWG!

# function definition
def stoiip_calc(area, thickness, poro, sw, boi):
    STOIIP = (7758*area*thickness*poro*(1-sw))/boi
    return STOIIP
```

Functions (Reusable Code)

- Function Arguments:
- Default arguments
- In some cases, some arguments may have default values – a standard value to take if value is not provided in the function call.

- Such default value is to be assigned to the concerned argument name at the point of defining the function.

- All such default arguments should only be listed after all un-defaulted arguments have been listed.

Functions (Reusable Code)

- Function Arguments:
- Default arguments
- Example: the density of a real gas (in lb/ft^3) is computed thus:
  - $\rho_g = \frac{2.7P\gamma_g}{zT}$

- While the equation above works for gas density at any given values of pressure and temperature; often, engineers want to compute gas density specifically at standard values of pressure (14.7 psia) and temperature (520^0R). At these values, z takes a standard value of 1.0.

Functions (Reusable Code)

- Function Arguments:
- Default arguments
- In this example, the function might be defined to accommodate these default values.
- However, the user may still specify other values for these default arguments when calling the function (more on this, soon).

```
#...TTOWG!

# Real gas density function definition
# Note: pressure must be in psia and temperature in degree Rankine

def density(gravity, pressure = 14.7, temperature = 520, z = 1):
    gas_density = (2.70*pressure*gravity)/(z*temperature)
    return gas_density
```

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Return value:
- A function should have a return statement specifying an output to be returned when called.
- The value to be returned is typically an output generated by one of the statements in the function; not necessarily the last output in the sequence.
- A collection of more than one value may be returned.

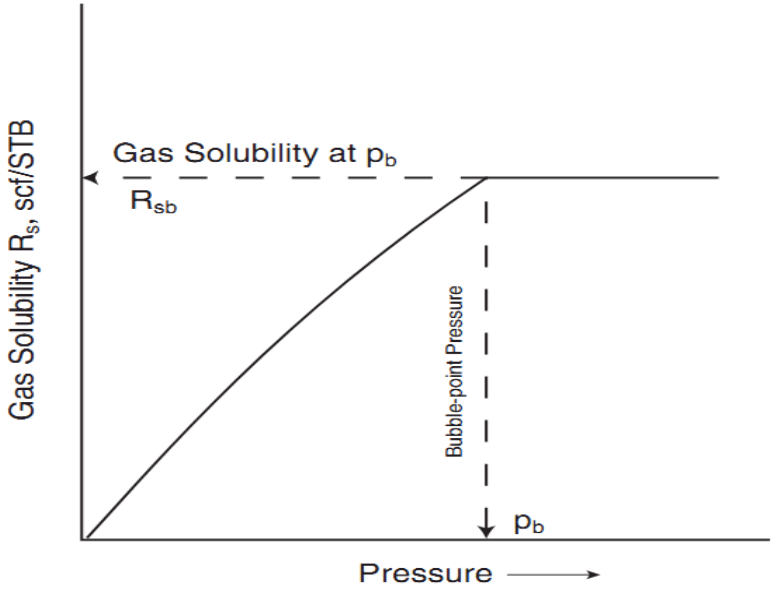
```
#...TTOWG!

# function definition
def stoiip_calc(area, thickness, poro, sw, boi):
    STOIIP = (7758*area*thickness*poro*(1-sw))/boi
    return STOIIP
```

```
def stoiip_calc2(area, thickness, poro, sw, boi):
    BV = area*thickness
    PV = BV*poro
    STOIIP = (7758*PV*(1-sw))/boi
    return PV, STOIIP # returning multiple values
```

Functions (Reusable Code)

- Return value:
- Alternative return values may exist, if the task involved a conditional execution.

- When no return value is specified, the function simply return a special value called None.




```
# TTOWG!

# A function to compute solution gas-oil ratio, Rs

def rs_calc(t, p, gravity, api, pb):
    y = (0.00091*t)-(0.0125*api)
    if p<pb:
        rs = gravity*(((p)/(18*(10**y)))**1.204)
        return rs
    else:
        rsb = gravity*(((pb)/(18*(10**y)))**1.204)
        return rsb
```

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Calling a Function
- A function (already defined) is called simply by typing its name and a list of its argument values enclosed in parenthesis.

```
# function call for Reservoir TTOWG_1 (re-use)
stoiip_calc(40, 15, 0.3, 0.28, 1.2)
```

- If the function requires no arguments, an empty parenthesis () is typed.

```
# function call for Reservoir TTOWG_2 (re-use)
stoiip_calc(80, 10, 0.23, 0.35, 1.1)
```

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Calling a Function

- If a function call is assigned to a variable on the LHS of the call statement, upon execution, the return value of the function is stored in that variable.

```
# function call for Reservoir TTOWG_1 (re-use)
oil_inplace_TTOWG_1 = stoiip_calc(40, 15, 0.3, 0.28, 1.2)
print('The amount of oil in place in Reservoir TTOWG_1 is', oil_inplace_TTOWG_1, 'STB')
```

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Calling a Function

- For functions that returns multiple values, the function call may be assigned to multiple variables on the LHS of the call.

- In that case, the return values are unbundled into respective LHS variables.

- Otherwise all the return values are stored into the single LHS variable as a collection.

```
>>> Pore_Vol, N = stoiip_calc2(40,15,0.3,0.28,1.2)
>>> print(Pore_Vol)
180.0
>>> print(N)
837864.0
```

```
>>> Volumetrics = stoiip_calc2(40,15,0.3,0.28,1.2)
>>> print(Volumetrics)
(180.0, 837864.0)
```

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Calling a Function
- Specifying arguments
- In specifying values of arguments, users don't have to necessarily indicate names of the arguments as stated during the function definition.

- Users only have to specify the argument values in the order that they were listed during definition – this is known as positional argument specification.

- Sometimes, the values to be passed to a function call might have been previously assigned to a variable; such variable name may be passed in place of the value.

```
# function call for Reservoir TTOWG_1 (re-use)
stoiip_calc(40, 15, 0.3, 0.28, 1.2)
```

```
drainage_area = 40
payzone_thickness = 15

print(stoiip_calc(drainage_area, payzone_thickness, 0.3, 0.28, 1.2))
```

Functions (Reusable Code)

- Calling a Function
- Specifying arguments
- Alternatively, argument values may be specified alongside the names listed during the function definition.
- In this case, specifying the arguments don't have to be in a specific order – this is known as keyworded argument specification.
- Caution: the names must be typed exactly, as typed during definition.

```
# function call with keyworded argument specification (ordered)
print(stoiip_calc(area = 40, thickness = 15, poro = 0.3, sw = 0.28, boi = 1.2)) # positional argument specification


# function call with keyworded argument specification (unordered)
print(stoiip_calc(poro = 0.3, area = 40, boi = 1.2, thickness = 15, sw = 0.28)) # positional argument specification
```

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Calling a Function
- Specifying arguments
- Positional argument specification versus Keyworded argument specification
- Positional argument specification saves typing efforts and reduces the risk of error in naming; requires you know the order of the argument.

- Keyworded argument specification makes no demand on the order, but requires you know the exact name of the arguments.

Functions (Reusable Code)

- Calling a Function with Defaulted Arguments
- When default value argument are present, both specification approach may be mixed in a function call.
- In such case, the un-defaulted (compulsory) arguments must first be specified positionally while the defaulted (optional) arguments may be specified with keywords.
- Take note that the fact that an argument is already defaulted does not imply that a user cannot specify another value.
- In cases where a user need to specify all compulsory arguments and some (not all) optional arguments; it is advisable that keywords be used.
- In general, positional arguments specification can only be used for arguments preceding the first skipped optional arguments.

Functions (Reusable Code)

- Calling a Function with Defaulted Arguments
- Given that function *gas_density* has been defined thus:

```
def gas_density(gravity, pressure = 14.7, temperature = 520, z = 1):
    density = (2.70*pressure*gravity)/(z*temperature)
    return round(density, 4)
```

Functions (Reusable Code)

- Calling a Function with Defaulted Arguments
- The following calls (except one) are equivalent:

```
>>> print(gas_density(0.786, 14.7, 520, 1)) # all positionally specified
0.06
>>> print(gas_density(gravity = 0.786, pressure =  14.7, temperature = 520, z = 1)) # all keyworded
0.06
>>> print(gas_density(0.786)) # all optional omitted
0.06
>>> # Below, temperature specified with keyword because pressure has been skipped
>>> print(gas_density(0.786, temperature = 520))
0.06
>>> # Below, pressure specified positionally because no argument has been skipped yet.
>>> print(gas_density(0.786, 14.7))
0.06
>>> # Below, temperature wrongly specified positionally, got interpreted as pressure
>>> print(gas_density(0.786, 520))
2.1222
```

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Calling a Function
- Function Availability
- Before a function is called in a script (program), it must be made available in that script.
- A function could be made directly available in a script by defining it in the same script where it is called.

```
#defining function stoiip
def stoiip(area, thickness, poro, sw, boi):
    N = (7758*area*thickness*poro*(1-sw))/boi
    return round(N, 2)

# the 'for' loop
for j in range(1,ny+1):
    for i in range(1,nx+1):
        block_n_order = (nx*(j-1))+i
        poro = float(input('What is the value of porosity for Block {0}?'.format(block_n_order)))
        sw = float(input('What is the value of water saturation for Block {0}?'.format(block_n_order)))
        block_stoiip = stoiip(area, h, poro, sw, boi)
        total_stoiip = total_stoiip + block_stoiip
        print('The amount of oil in Block {0} is {1:.2f} STB'.format(block_n_order, block_stoiip))
```

Functions (Reusable Code)

- Calling a Function
- Function Availability
- Sometimes, a function defined in script (defining script) needed to be called in another script (calling script). To avoid clumsiness, you may not want to copy the function definition to the calling script; in such cases, you simply *import* the function.

Functions (Reusable Code)

- Calling a Function
- Function Availability
- Importing Functions
- You may import function(s) using the following approaches:
- Given that function *stoiip* has been defined in script (file) *peteng.py*
  - View the content of *peteng.py* here.

Functions (Reusable Code)

- Calling a Function
- Function Availability
- Importing Functions
- Approach 1: Import the entire file

```
# importing the file where stoiip is defined
import peteng

# the 'for' loop
for j in range(1,ny+1):
    for i in range(1,nx+1):
        block_n_order = (nx*(j-1))+i
        poro = float(input('What is the value of porosity for Block {0}?'.format(block_n_order)))
        sw = float(input('What is the value of water saturation for Block {0}?'.format(block_n_order)))
        block_stoiip = peteng.stoiip(area, h, poro, sw, boi)
        total_stoiip = total_stoiip + block_stoiip
        print('The amount of oil in Block {0} is {1:.2f} STB'.format(block_n_order, block_stoiip))
```

Functions (Reusable Code)

- Calling a Function
- Function Availability
- Importing Functions
- Approach 1: Import the entire file
  - Notes:
    - In this case, the whole content of the file (function stoiip and others) is imported.
    - Calling any function in the imported file must be preceded with the file name and a period(.); e.g: peteng.stoiip(…)

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Calling a Function
- Function Availability
- Importing Functions
- Approach 2: From the file, import all functions

```
# importing the file where stoiip is defined
from peteng import *

# the 'for' loop
for j in range(1,ny+1):
    for i in range(1,nx+1):
        block_n_order = (nx*(j-1))+i
        poro = float(input('What is the value of porosity for Block {0}?'.format(block_n_order)))
        sw = float(input('What is the value of water saturation for Block {0}?'.format(block_n_order)))
        block_stoiip = stoiip(area, h, poro, sw, boi)
        total_stoiip = total_stoiip + block_stoiip
        print('The amount of oil in Block {0} is {1:.2f} STB'.format(block_n_order, block_stoiip))
```

Functions (Reusable Code)

- Calling a Function
- Function Availability
- Importing Functions
- Approach 2: From the file, import all functions
  - Notes:
    - In this case also, the whole content of the file (function stoiip and others) is imported.
    - The functions should be called without any prefix e.g: stoiip(…)

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

- Calling a Function
- Function Availability
- Importing Functions
- Approach 3: From the file, import only needed function(s)

```
# importing the file where stoiip is defined
from peteng import stoiip

# the 'for' loop
for j in range(1,ny+1):
    for i in range(1,nx+1):
        block_n_order = (nx*(j-1))+i
        poro = float(input('What is the value of porosity for Block {0}?'.format(block_n_order)))
        sw = float(input('What is the value of water saturation for Block {0}?'.format(block_n_order)))
        block_stoiip = stoiip(area, h, poro, sw, boi)
        total_stoiip = total_stoiip + block_stoiip
        print('The amount of oil in Block {0} is {1:.2f} STB'.format(block_n_order, block_stoiip))
```

Functions (Reusable Code)

- Calling a Function
- Function Availability
- Importing Functions
- Approach 3: From the file, import all functions
  - Notes:
    - In this case, only the function(s) indicated after the keyword import is imported.
    - Here too, the function(s) should be called without any prefix e.g: stoiip(…)

GETTING STARTED WITH PYTHON

Functions (Reusable Code)

Calling a Function

```
>>> # Sample calls to functions defined in peteng.py
>>> from peteng import *
>>> gas_density(0.786, 14.7, 520, 1)
0.06
>>> bubble_pressure(220, 4000, 0.786, 0.8217, 743)
2608.51
>>> sol_gor(220, 4000, 0.786, 0.8217, 2608.51)
743.76
>>> fvf(4000, 220, 0.786, 0.8217, pb = 2608.51, co = 0.00001413) # rs skipped
1.4267
>>> fvf(4000, 220, 0.786, 0.8217, rs = 743.76, co = 0.00001413) # pb skipped
1.4267
>>> fvf(2100, 220, 0.786, 0.8217, rs = 570, pb = 2608.51) # co not required
1.3593
>>> fvf(2100, 220, 0.786, 0.8217, pb = 2608.51) # rs skipped, co not required
1.3608
```

The file *peteng.py* is available here

The algorithms for various functions in *peteng.py* are available here

```
>>>#TTOWG!
>>>print('…to the only wise God')
```

Coming Soon…

Season 3 Episode 3 –

Python Data Structures

PYTHON DATA STRUCTURES

- So far in this course, we have only used variables that holds single numerical values as data
  - Examples: poro = 0.23; area = 40; gas_gravity = 0.786.

- In reality, petroleum engineering data may be a collection of more than one numerical value.
  - Example: the porosity data of a discretized reservoir would be a collection of porosity values of all gridblocks in the reservoir.
  - For efficiency purposes, the multi-valued porosity data need to be passed to Python scripts as a single variable.

0.1	0.25	0.29	0.33
0.23	0.27	0.25	0.1
0.29	0.23	0.33	0.27

PYTHON DATA STRUCTURES

- Thankfully, Python has various in-built objects that permit the assignment of such multi-valued data to variables.
- These objects are here referred to as Python data structures.
- Lists, Tuples, Dictionaries and Strings are common examples of Python data structures.

PYTHON DATA STRUCTURES

Lists

- A list is essentially a sequence of values with the following capabilities:
  - It is mutable – the constituent values can be altered/modified in-place (i.e. altered without creating a new list)
    - Alterations such as changing values, sorting etc.
  - It has variable length – values may be added/deleted to/from the list.
  - It can hold any type of values: integers, floats, characters, lists, tuples, dictionaries or mixture of these.

  - Individual values in a list are called elements or items.
  - Note that lists are one-dimensional: i.e. the sequence of values are only in one-dimension (columns) not two dimensions as in rows and columns.

PYTHON DATA STRUCTURES

Lists

- Defining a list
- A list is defined (created) by simply enclosing comma-separated values in square brackets

```
>>> [1,2,3,4,5] # a list of integers
[1, 2, 3, 4, 5]
>>> [0.23, 4.6, 23.6] # a list of floating-point numbers
[0.23, 4.6, 23.6]
>>> ['T', 'T', 'O', 'W', 'G', '!'] # a list of characters
['T', 'T', 'O', 'W', 'G', '!']
>>> ['Today', 'is', 'August', 30, 2021] # a list of strings and numerals
['Today', 'is', 'August', 30, 2021]
>>> [7, 8, [1.2, 0.5], 14, 9] # a list containing another list - nested list
[7, 8, [1.2, 0.5], 14, 9]
```

PYTHON DATA STRUCTURES

Lists

- Defining a list
- A list is defined (created) by simply enclosing comma-separated values in square brackets

```
>>> [1,2,3,4,5] # a list of integers
[1, 2, 3, 4, 5]
>>> [0.23, 4.6, 23.6] # a list of floating-point numbers
[0.23, 4.6, 23.6]
>>> ['T', 'T', 'O', 'W', 'G', '!'] # a list of characters
['T', 'T', 'O', 'W', 'G', '!']
>>> ['Today', 'is', 'August', 30, 2021] # a list of strings and numerals
['Today', 'is', 'August', 30, 2021]
>>> [7, 8, [1.2, 0.5], 14, 9] # a list containing another list - nested list
[7, 8, [1.2, 0.5], 14, 9]
```

- Often, there is a need to initialize a list as an empty list to be populated with values later (may be in a 'for' loop).

```
>>> [] # an empty list!
[]
```

- Of course, a list may be assigned to a variable

```
>>> poro = [0.29, 0.23, 0.33, 0.27, 0.23, 0.27, 0.25, 0.1, 0.1, 0.25, 0.29, 0.33]
>>> print(poro)
[0.29, 0.23, 0.33, 0.27, 0.23, 0.27, 0.25, 0.1, 0.1, 0.25, 0.29, 0.33]
```

PYTHON DATA STRUCTURES

Lists

- Defining a list
- A list may also be defined with the function *list*.
- Note that the entire collection of values must be enclosed in round brackets before being passed as argument to function *list*.
- More specifically, function list is used to coerce (convert) a non-list data to a list.

```
>>> poro = list((0.29, 0.23, 0.33, 0.27, 0.23, 0.27, 0.25, 0.1, 0.1, 0.25, 0.29, 0.33))
>>> print(poro)
[0.29, 0.23, 0.33, 0.27, 0.23, 0.27, 0.25, 0.1, 0.1, 0.25, 0.29, 0.33]
```

```
my_list = list(0.2, 0.1, 0.4)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    my_list = list(0.2, 0.1, 0.4)
TypeError: list expected at most 1 argument, got 3
```

```
>>> print(list('TTOWG'))
['T', 'T', 'O', 'W', 'G']
```

```
list('ttowG')
['t', 't', 'o', 'w', 'G']
list('ttowG',)
['t', 't', 'o', 'w', 'G']
list(('ttowG',))
['ttowG']
list((1,2,3),)
[1, 2, 3]
list(((1,2,3),))
[(1, 2, 3)]
list(([1,2,3],))
[[1, 2, 3]]
```

```
list(1)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    list(1)
TypeError: 'int' object is not iterable
list(1,)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    list(1,)
TypeError: 'int' object is not iterable
list((1))
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    list((1))
TypeError: 'int' object is not iterable
list((1,))
[1]
```

PYTHON DATA STRUCTURES

Lists

- Accessing elements of a list
- Necessarily, there would be need to access individual or group of elements in a list.

- Such need to access might be for the purpose of:
  - extraction:
    - of such element(s) for use in an expression.
    - of such element(s) to be assigned to another variable
  - re-assignment: i.e. changing the value of that element

- In any case, accessing the element(s) of a list is done by referring to the index (or indices) of the concerned element(s).

PYTHON DATA STRUCTURES

Lists

- Accessing elements of a list
- List Indices
- List indices are simply integers that corresponds to the position of the respective elements of lists.

- Note: the position starts counting from 0, not 1.
  - The index of first element is 0;
  - the index of the second element is 1;
  - the index of nth element is n-1

- Negative indices counts backward in a list – learn more

PYTHON DATA STRUCTURES

Lists

- Accessing elements of a list
- Accessing single element in a list
- To access a single element of a list, simply type the name of the list, followed by a pair of square brackets in which the index is specified.
- Here are a few statements accessing elements for extraction purposes.

```
>>> poro = [0.29, 0.23, 0.33, 0.27, 0.23, 0.27, 0.25, 0.1]
>>> print(poro)
[0.29, 0.23, 0.33, 0.27, 0.23, 0.27, 0.25, 0.1]
>>> poro[2] # accessing the third element
0.33
>>> block_poro = poro[4] # extracting and assigning to variable block_poro
>>> print(block_poro)
0.23
>>> block_BV = 100
>>> block_PV = block_BV * poro[3] # extracting for use in an expression
>>> print(block_PV)
27.0
```

PYTHON DATA STRUCTURES

Lists

- Accessing elements of a list
- Accessing single element in a list
- To access a single element of a list, simply type the name of the list, followed by a pair of square brackets in which the index is specified.
- Here are a few statements accessing elements for re-assignment.

```
>>> print(poro)
[0.29, 0.23, 0.33, 0.27, 0.23, 0.27, 0.25, 0.1]
>>> poro[5] = 0.11 # altering (reassigning) the sixth element to 0.11
>>> print(poro)
[0.29, 0.23, 0.33, 0.27, 0.23, 0.11, 0.25, 0.1]
>>> block_BV = 100
>>> block_PV = 15
>>> poro[0] = block_PV/block_BV # # altering the first element to the value of the RHS expression.
>>> print(poro)
[0.15, 0.23, 0.33, 0.27, 0.23, 0.11, 0.25, 0.1]
```

PYTHON DATA STRUCTURES

Lists

- Accessing elements of a list
- Accessing multiple elements in a list – List Slices
- To access multiple elements of a list (a slice of the list), simply type the name of the list, followed by a pair of square brackets in which the range of indices is specified.
- A range of indices is specified by typing the starting index, followed by a colon, and finally, the index before which the slice must stop.

```
>>> print(poro)
[0.15, 0.23, 0.33, 0.27, 0.23, 0.11, 0.25, 0.1]
>>> some_poro = poro[2:6] # extracting a slice
>>> print(some_poro)
[0.33, 0.27, 0.23, 0.11]
>>> poro[1:3] = [0.25, 0.17] # re-assigning a slice
>>> print(poro)
[0.15, 0.25, 0.17, 0.27, 0.23, 0.11, 0.25, 0.1]
>>> poro_values = [0.34, 0.18]
>>> poro[4:6] = poro_values # re-assigning a slice to the content of a variable
>>> print(poro)
[0.15, 0.25, 0.17, 0.27, 0.34, 0.18, 0.25, 0.1]
```

PYTHON DATA STRUCTURES

Lists

- Accessing elements of a list
- Accessing multiple elements in a list – List Slices
- Generally, a range of indices to access a slice containing nth element through to mth element is specified thus:
  - $n-1:m$ (take note that both n and m here refer to natural element counters)
- The index on the left of the colon (i.e. the starting index) is n-1 because indices are counted from 0.
- The index on the right of the colon is m (not m – 1, as expected) because Python excludes the last element of any range, so it stops at (m - 1)th position.
  - "as expected' by reason of indexing from zero

```
>>> print(poro)
[0.15, 0.23, 0.33, 0.27, 0.23, 0.11, 0.25, 0.1]
>>> some_poro = poro[2:6] # extracting a slice
>>> print(some_poro)
[0.33, 0.27, 0.23, 0.11]
>>> poro[1:3] = [0.25, 0.17] # re-assigning a slice
>>> print(poro)
[0.15, 0.25, 0.17, 0.27, 0.23, 0.11, 0.25, 0.1]
>>> poro_values = [0.34, 0.18]
>>> poro[4:6] = poro_values # re-assigning a slice to the content of a variable
>>> print(poro)
[0.15, 0.25, 0.17, 0.27, 0.34, 0.18, 0.25, 0.1]
```

PYTHON DATA STRUCTURES

Lists

- Accessing elements of a list
- Accessing multiple elements in a list – List Slices

- If the left index is not specified, then the slice is taken from the beginning of the list.
- If the right index is not specified, then the slice is taken to the end of the list.
- If both indices are not specified, then the entire list is returned unsliced

```
>>> print(poro)
[0.15, 0.25, 0.17, 0.27, 0.34, 0.18, 0.25, 0.1]
>>> poro[:4] # left index ommitted, slice from beginning
[0.15, 0.25, 0.17, 0.27]
>>> poro[2:] # right index ommitted, slice to end
[0.17, 0.27, 0.34, 0.18, 0.25, 0.1]
>>> poro[:] # both indices ommitted, return entire list.
[0.15, 0.25, 0.17, 0.27, 0.34, 0.18, 0.25, 0.1]
>>>
```

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions

- Concatenation: combining multiple lists (side by side) using the + operator.

- Replication: replicating all elements of a list a given number of times using the * operator.

```
>>> print(poro)
[0.15, 0.25, 0.17, 0.27, 0.34, 0.18, 0.25, 0.1]
>>> more_poro = [0.28, 0.19, 0.20, 0.16]
>>> all_poro = poro + more_poro # concatenating
>>> print(all_poro)
[0.15, 0.25, 0.17, 0.27, 0.34, 0.18, 0.25, 0.1, 0.28, 0.19, 0.2, 0.16]
```

```
>>> perm_left = [120, 114, 150, 170]
>>> all_perm = perm_left*2 # replicating, twice
>>> print(all_perm)
[120, 114, 150, 170, 120, 114, 150, 170]
```

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- An extra element (just 1) could be appended to the end of a list using the *append* method.

```
>>> all_stoiip = [1456, 1738, 1509] # say these are stoiip for Blocks 1, 2, and 3
>>> stoiip_block4 = 1611 # just obtained stoiip for Block 4
>>> all_stoiip.append(stoiip_block4) # appending stoiip_block4 to the end of lis
>>> print(all_stoiip)
[1456, 1738, 1509, 1611]
```

- This *append* method is commonly used to append a newly generated output to the overall list of outputs in a *for* loop.

```
total_stoiip = 0
list_of_stoiip = []

# the 'for' loop
for j in range(1,ny+1):
    for i in range(1,nx+1):
        block_n_order = (nx*(j-1))+i
        poro = float(input('What is the value of p
        sw = float(input('What is the value of wa
        block_stoiip = (7758*area*h*poro*(1-sw)
        list_of_stoiip.append(block_stoiip)
        total_stoiip = total_stoiip + block_stoiip
```

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- An existing list could be extended with multiple elements using the *extend* method.

```
>>> pressures = [4000, 3890, 3503]
>>> more_pressures = [3205, 2956]
>>> pressures.extend(more_pressures) # extending
>>> print(pressures)
[4000, 3890, 3503, 3205, 2956]
```

PYTHON DATA STRUCTURES

Lists

List Operations, Methods and Functions

- Please, note that the append and extend methods do not return values; rather, they modify the object list in-place (without copying). Caution: never do poro = poro.append()
- The argument remains unmodified
- The object list refers to the list on which the method is called while the argument list refers to the list to be appended to (or extended) to the object list.

```
>>> print(all_stoiip)
[1456, 1738, 1509, 1611, 1487]
>>> another_stoiip = 1500
>>> complete_stoiip = all_stoiip.append(another_stoiip) # modifies all_stoiip; returns None to complete_stoiip
>>> print(complete_stoiip)
None
>>> print(all_stoiip)
[1456, 1738, 1509, 1611, 1487, 1500]
>>> print(another_stoiip)
1500
```

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- An element could be inserted into any position in a list (not just the end of the list) using the *insert* method.
- Method *insert* does not return a value, it only modifies the object in-place.

- Find out if Method *insert* could be used to insert multiple elements.

```
>>> poro = [0.16, 0.27, 0.30]
>>> # oops! I missed the second poro value
>>> poro.insert(1, 0.11) # inserting into 2nd position (index 1)
>>> print(poro)
[0.16, 0.11, 0.27, 0.3]
```

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- An element could be removed from any position in a list using the *pop* method, if the position is known, of course.

```
>>> print(poro)
[0.16, 0.11, 0.27, 0.3]
>>> removed_poro = poro.pop(2) # removing the 3rd element
>>> print(poro)
[0.16, 0.11, 0.3]
>>> print(removed_poro)
0.27
```

- Method *pop* returns the removed value, and also modifies the object in-place accordingly.
- Method *pop* could be useful if you need to use a value in a list and like to remove the used value to avoid reusing it – sort of sampling without replacement.

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- However, if the position of the element to be removed is not known, but the value is known; then the *remove* method can be used to remove it.

- However, Method *remove* only removes the first occurrence of the specified value in the list; subsequent occurrences of that specified value are not removed.
- Method *remove* does not return a value, it only modifies the object in-place.

```
>>> print(poro)
[0.16, 0.11, 0.27, 0.3, 0.11, 0.23]
>>> poro.remove(0.11)
>>> print(poro)
[0.16, 0.27, 0.3, 0.11, 0.23]
>>>
```

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- Both Methods *pop* and *remove* accept only one argument; so they can only delete one element of the list at a time.
- To delete multiple elements, use the *del* operator.

```
>>> print(poro)
[0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
>>> del poro[1:3]
>>> print(poro)
[0.16, 0.25, 0.11, 0.23]
```

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- Elements of a list can be sorted either in ascending order of descending order using Method *sort*.
- By default, the method sorts in ascending order. However, to sort in descending order, set argument *reverse* to True.
- Caution: never do poro = poro.sort()

```
>>> print(poro)
[0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
>>> sorted_poro = poro.sort()
>>> print(sorted_poro) # to show that sort does not return value.
None
>>> print(poro) # to show that sort modifies in-place.
[0.11, 0.16, 0.23, 0.25, 0.28, 0.34]
>>> poro.sort(reverse = True) # sorting in descending order.
>>> print(poro)
[0.34, 0.28, 0.25, 0.23, 0.16, 0.11]
```

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- That a value is present (or not present) in a list could be determined using the *in* operator.

```
>>> print(poro)
[0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
>>> 0.34 in poro # checking if 0.34 is present in poro
True
>>> 0.15 in poro # checking if 0.15 is present in poro
False
```

- The *in* operator returns True if the specified value is present in the list, and False, otherwise.
- Such *in* operations might be used in constructing the conditions for *if* and *while* statements.

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- Sometimes, the presence of a value in a list is already ascertained, but there is a need to determine the exact location (index) of the value in the list. Method *index* could be used for such purpose.

```
>>> print(poro)
[0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
>>> poro.index(0.28)
2
>>> poro.index(0.30)
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    poro.index(0.30)
ValueError: 0.3 is not in list
```

- If the specified value occurs more than once, only the index of the first occurrence is returned.
- Assignment: how may the indices of all occurrences be obtained?
- If the specified value is not present, an error is thrown.
- Method *index* may be used to implement a *find* and *replace* functionality.

PYTHON DATA STRUCTURES

Lists

- List Operations, Methods and Functions
- You can use the following functions to get some summary statistics of the elements of a list, without manually looping through the list.
  - *len* – returns number of elements
  - *sum* – returns sum of elements.
  - *min* – returns the smallest element.
  - *max* – returns the largest element

```
>>> print(poro)
[0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
>>> len(poro)
6
>>> sum(poro)
1.37
>>> min(poro)
0.11
>>> max(poro)
0.34
```

PYTHON DATA STRUCTURES

Lists

Looping through Lists

- Often, there is a need to perform an element-wise operation on a list; i.e. same operation performed on each element of the list.

- Typically, this can be done with a 'for' loop.

PYTHON DATA STRUCTURES

Lists

Looping through Lists

- The 'for' loop traverses through the elements of a list sequentially.
- Generally, the structure of the header of such 'for' loop is thus:

```
for iterator_name in list_name:
```

  - where *iterator_name* refers to the name of the iterator variable and *list_name* refers to the name of the list to be traversed.

PYTHON DATA STRUCTURES

Lists

- Looping through Lists
- At the beginning of each cycle of the 'for' loop, the value of the corresponding (i.e. current) element of the list is assigned to the iterator variable.
  - Example, at the 3rd cycle, the iterator variable stores the 3rd element; at the 4th cycle, the iterator variable discards the 3rd element and stores the 4th element.
- That is, the current element is accessed, extracted and assigned to the iterator variable.

```
>>> print(poro)
[0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
>>>
>>> for poro_val in poro:
        print('Currently, poro_val stores {0}'.format(poro_val))


Currently, poro_val stores 0.16
Currently, poro_val stores 0.34
Currently, poro_val stores 0.28
Currently, poro_val stores 0.25
Currently, poro_val stores 0.11
Currently, poro_val stores 0.23
```

PYTHON DATA STRUCTURES

Lists

- Looping through Lists
- With such successive element-value assignment to the iterator variable, the iterator variable only needed to be called in any expression/statement/operation that requires the current element value.

- Example: multiply a given (constant) block bulk volume (BV) by each element of a list of porosity values (poro) in order to obtain pore volumes corresponding to each element of the list poro.

```
>>> BV = 100
>>> poro = [0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
>>>
>>> for poro_val in poro:
        PV = BV*poro_val
        print('Currently, pore volume is {0}'.format(PV))


Currently, pore volume is 16.0
Currently, pore volume is 34.0
Currently, pore volume is 28.000000000000004
Currently, pore volume is 25.0
Currently, pore volume is 11.0
Currently, pore volume is 23.0
```

PYTHON DATA STRUCTURES

Lists

- Looping through Lists
- Notice that so far, the 'for' loop only accessed the current element of the list for the purpose of extraction and usage.
- The structure of the 'for' loop so far presented can not access the elements of the list for the purpose of re-assignment (editing it).
- To achieve this, an alternative structure of the 'for' loop header is needed

```
>>> print(poro)
[0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
>>> # let's say, we want to change every
>>> #  element to the average porosity value
>>> mean_poro = sum(poro)/len(poro)
>>> print(mean_poro)
0.22833333333333336
>>> for poro_val in poro:
        poro_val = mean_poro


>>> print(poro) # to see if it has been edited as intended
[0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
```

PYTHON DATA STRUCTURES

Lists

- Looping through Lists
- The alternative 'for' loop header structure utilizes the indices of the list instead of the list itself.

```
for counter in list_indices:
```

  - where *counter* is the iterator variable and *list_indices* is the object to be traversed.
- With this, the index of the current element is assigned to the iterator variable

- Consequently, the current element can be accessed either for extraction or editing purposes by simply calling the list itself and passing the iterator variable (which now hold the index of the current element) as the index of such call.

PYTHON DATA STRUCTURES

Lists

- Looping through Lists
- The needed indices list can be obtained by passing the length of the original list to the Function *range*.
- By default, Function *range* generates consecutive integers from 0 to the passed argument less 1.
- The needed length may be obtained with Function *len*

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>>
>>> len(poro)
6
>>> list(range(len(poro)))
[0, 1, 2, 3, 4, 5]
```

```
>>> print(poro)
[0.16, 0.34, 0.28, 0.25, 0.11, 0.23]
>>> # let's say, we want to change every
>>> # element to the average porosity value
>>> mean_poro = sum(poro)/len(poro)
>>> print(mean_poro)
0.2283333333333336
>>> for i in range(len(poro)):
        poro[i] = round(mean_poro, 3)


>>> print(poro) # to see if it has been edited as intended
[0.228, 0.228, 0.228, 0.228, 0.228, 0.228]
```

PYTHON DATA STRUCTURES

Tuples

- Just like a list, a tuple is simply a sequence/collection of values.
- However, unlike lists, tuples are NOT mutable.
  - The constituent values cannot be altered/modified in-place (i.e. altered without creating a new list)
    - Elements cannot be reassigned
    - New element(s) cannot be appended or inserted
    - Tuples cannot be sorted in-place nor be extended.

- Like lists, tuples can hold any type of values: integers, floats, characters, strings, lists, tuples, dictionaries or mixture of these.
- Individual values in a tuples are also called elements or items.
- Note that tuples are one-dimensional: i.e. the sequence of values are only in one-dimension (columns) not two dimensions as in rows and columns.

PYTHON DATA STRUCTURES

Tuples

- Tuples are immutable as shown in the following mutation attempts that worked with lists but not with tuples.

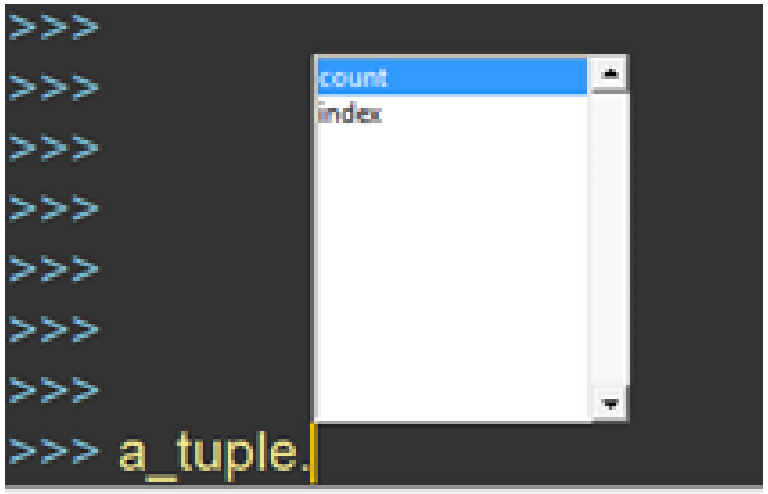
```
>>> a_list = [0.3, 0.24, 0.18]
>>> a_tuple = (0.3, 0.24, 0.18)
>>> print(a_list)
[0.3, 0.24, 0.18]
>>> print(a_tuple)
(0.3, 0.24, 0.18)
>>> a_list[1] = 0.34 # reassigning the 2nd elements: possible with lists
>>> print(a_list)
[0.3, 0.34, 0.18]
>>> a_tuple[1] = 0.34 # reassigning the 2nd elements: impossible with tuples

Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    a_tuple[1] = 0.34 # reassigning the 2nd elements: impossible with tuples
TypeError: 'tuple' object does not support item assignment
```

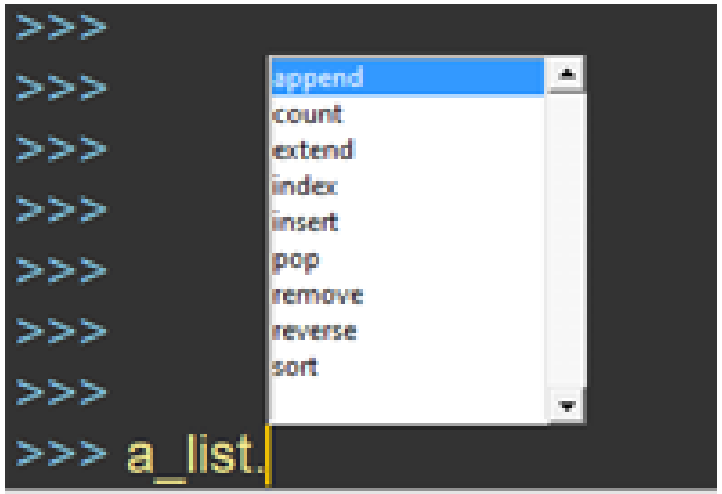
```
>>> a_list.append(0.29) # appending 0.29 to a_list: possible
>>> print(a_list)
[0.3, 0.34, 0.18, 0.29]
>>> a_tuple.append(0.29) # appending 0.29 to a_tuple: impossible

Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    a_tuple.append(0.29) # appending 0.29 to a_tuple: impossible
AttributeError: 'tuple' object has no attribute 'append'
```

PYTHON DATA STRUCTURES

Tuples

- As a matter of facts, many in-place modification methods (that do not return values) available for lists are NOT available for tuples





PYTHON DATA STRUCTURES

Tuples

- Only operations/functions/methods that return values are usable with tuples.
- Such return values may be assigned to a new tuple on the LHS of the statement.

```
>>> print(a_tuple)
(0.3, 0.24, 0.18)
>>> another_tuple = a_tuple + (0.29,) # instead of a_tuple.append
>>> print(another_tuple)
(0.3, 0.24, 0.18, 0.29)
>>> another_tuple = a_tuple + (0.29, 0.31)
>>> print(another_tuple)
(0.3, 0.24, 0.18, 0.29, 0.31)
```

PYTHON DATA STRUCTURES

Tuples

- Further still, if you really want to 'modify' a tuple; you might invoke an operation that return a value, and then assign the return values to the name of the tuple; that way, the return value overwrites the original contents of the tuple.

```
>>> print(a_tuple)
(0.3, 0.24, 0.18)
>>> a_tuple = a_tuple + (0.29, 0.31) # instead of a_tuple.extend
>>> print(a_tuple)
(0.3, 0.24, 0.18, 0.29, 0.31)
```

PYTHON DATA STRUCTURES

Tuples

- Defining a tuple
- A tuple is defined (created) by simply enclosing comma-separated values in curved brackets.
- The enclosing curved brackets are optional, a comma-separated sequence of values is by default taken as a tuple.
- If a tuple contains only one element, the comma should still be added to the right side of the element, for Python to recognize it as a tuple.

```
>>> a_tuple = (0.30, 0.24, 0.18)
>>> print(a_tuple)
(0.3, 0.24, 0.18)
>>> another_tuple = 0.30, 0.24, 0.18
>>> print(another_tuple)
(0.3, 0.24, 0.18)
>>> type(another_tuple)
<class 'tuple'>
>>> one_tuple = (0.30, ) # Note the comma
>>> print(one_tuple)
(0.3,)
>>> type(one_tuple)
<class 'tuple'>
```

PYTHON DATA STRUCTURES

Tuples

- Defining a tuple
- Often, there is a need to initialize a tuple as an empty tuple to be populated with values later (may be in a 'for' loop).

```
>>> zero_tuple = ()
>>> print(zero_tuple)
()
>>> type(zero_tuple)
<class 'tuple'>
```

PYTHON DATA STRUCTURES

Tuples

- Defining a tuple
- Often, there is a need to initialize a tuple as an empty tuple to be populated with values later (may be in a 'for' loop).

```
>>> zero_tuple = ()
>>> print(zero_tuple)
()
>>> type(zero_tuple)
<class 'tuple'>
```

PYTHON DATA STRUCTURES

Tuples

- Defining a tuple
- A tuple may also be defined with the function *tuple*.

- More specifically, function *tuple* is used to coerce (convert) a non-tuple data to a tuple.

```
>>> tuple([5, 9])
(5, 9)
```

PYTHON DATA STRUCTURES

Tuples

- Accessing elements of a tuple
- Accessing element(s) of a tuple is done in the exact same way as in accessing element(s) of a list.

```
>>> print(a_tuple)
(0.3, 0.24, 0.18)
>>> a_tuple[0]
0.3
>>> a_tuple[0:2]
(0.3, 0.24)
>>> a_tuple[1:]
(0.24, 0.18)
```

PYTHON DATA STRUCTURES

Tuples

Tuples Operations, Methods and Functions

- While many in-place list modification operations are not available for tuples, operations/methods/functions that return values work on tuples in the same way they work on lists.
  - Examples: +, *, *.index, .count, in, len, sum, min and max* all work on tuples.

PYTHON DATA STRUCTURES

Tuples

- Multiple Assignments with tuples

- You could use a tuple (a list, too) to assign values to multiple variables with a single statement.

- Python simply unpacks the values at the right and assigns them to the variables at the left, respectively.

- This is useful in returning multiple outputs from a function

```
>>> (perm, poro) = (100, 0.37)
>>> print(perm)
100
>>> print(poro)
0.37
```

PYTHON DATA STRUCTURES

Lists and Tuples

- Choosing between lists and tuples

- Admittedly, lists and tuples may be used interchangeably.

- However, there are instances when one is preferred than the other.
  - When mutability is involved, use a list. Examples:
    - a sequence initialized outside a 'for' loop to be updated in the loop
    - A sequence that needed to be sorted.
  - When mutability is NOT desired, use a tuple. Examples:
    - arguments to functions
    - return values of functions

Lists and Tuples

Petroleum engineering applications

- Being multi-valued data objects, lists and tuples fits well into reservoir modelling and simulation wherein reservoirs are discretized into multiple gridblocks.
- The input parameters in such simulations are typically multi-valued grid of rock and fluid properties.
- Such grid of values may be packaged as lists/tuples before being passed to functions involved in the simulation.
- Also, the output of such simulations might be grid values; such outputs may be packaged as list/tuples.

PYTHON DATA STRUCTURES

Lists and Tuples

Petroleum engineering applications

- Recall *demo_stoiip_calc_discretized_model*? Check it here.
- It is a script to compute STOIIP for a discretized reservoir model.
- However, it is clumsy to use as it is; as the input poro and swi had to be supplied for each block one after the other.
- For ease of use, it has been converted to a function (*stoiip_discretized*); with the poro and swi grid of values packaged as tuples and passed to the function. Function *stoiip_discretized* is listed in peteng.py here
- Also, the output block_stoiip that get printed per block can be returned as a single list (stoiip_list) for all blocks as a whole.

PYTHON DATA STRUCTURES

Lists and Tuples

Petroleum engineering applications

- The header of the function is thus:

```
######### A function to compute STOIIP for all blocks in a discretized reservoir #########
def stoiip_discretized(Lx, Ly, h, nx, ny, boi, poro_list, swi_list):
```

- The initialization and updating of the stoiip_list is thus:

```
# initializing output variables
total_stoiip = 0
stoiip_list =[]
```

```
stoiip_list.append(block_stoiip)
total_stoiip = total_stoiip + block_stoiip
```

- The return statement, featuring a tuple is thus:

```
return (total_stoiip, stoiip_list)
```

PYTHON DATA STRUCTURES

Lists and Tuples

Petroleum engineering applications

- Examples of calls to stoiip_discretized:

```
>>> # Sample call 1
>>> volumetrics = stoiip_discretized(100, 100, 40, 2, 1, 1.2, [0.2, 0.2], [0.3, 0.3])
>>> print(volumetrics)
(362040000.0, [181020000.0, 181020000.0])
```

```
>>> # Sample call 2: with the output tuple unpacked into two separate variables
>>> (my_STOIIP, my_STOIIP_list) = stoiip_discretized(100, 100, 40, 2, 1, 1.2, [0.2, 0.2], [0.3, 0.3])
>>> print(my_STOIIP)
362040000.0
>>> print(my_STOIIP_list)
[181020000.0, 181020000.0]
```

PYTHON DATA STRUCTURES

Dictionaries

- A dictionary is a mutable sequence of values; just like a list.
- The elements in a list are indexed with integers (indices) corresponding to their respective positions in the sequence.
- However, the indices (known as *keys*) of a dictionary object are specified alongside the elements (*values*) in the sequence.
  - i.e. the elements (*values*) of a dictionary object are indexed with their respective *keys* as specified along with the values.
- This makes the items in a dictionary to be pairs of *key-value*; while the items in a list/tuples are only values.

PYTHON DATA STRUCTURES

Dictionaries

- Example:
- A collection of values corresponding to the drainage area (A), payzone thickness (h), porosity (poro), initial water saturation (swi) and oil formation volume factor (boi) for a certain reservoir may be presented as a list or as dictionary thus:

```
>>> #As a list
>>> data_list = [40, 15, 0.2, 0.31, 1.2417]
>>> print(data_list)
[40, 15, 0.2, 0.31, 1.2417]
>>>
>>> #As a dictionary
>>> data_dict = {'area': 40, 'h': 15, 'poro': 0.2, 'swi': 0.31, 'boi': 1.2417}
>>> print(data_dict)
{'area': 40, 'h': 15, 'poro': 0.2, 'swi': 0.31, 'boi': 1.2417}
```

PYTHON DATA STRUCTURES

Dictionaries

- Example:
- The main difference between these presentation is the way elements can be accessed.
- For instance, in data_list, poro can only be accessed with index 2 whereas, in data_dict, poro can be accessed with the string 'poro'

```
>>> print(data_list[2])
0.2
>>> print(data_dict['poro'])
0.2
```

PYTHON DATA STRUCTURES

Dictionaries

- Defining a dictionary
- A dictionary is simply defined by enclosing key-value pairs in curly brackets.
  - Note: the keys must be strings
- An empty dictionary, (to be updated later) may be defined as {}
- Function dict may also be used to define a dictionary or to convert an object to a dictionary.

```
>>> data_dict = {'area': 40, 'h': 15, 'poro': 0.2, 'swi': 0.31, 'boi': 1.2417}
>>> print(data_dict)
{'area': 40, 'h': 15, 'poro': 0.2, 'swi': 0.31, 'boi': 1.2417}
>>> init_dict = {}
>>> print(init_dict)
{}
>>> init_dict2 = dict()
>>> print(init_dict2)
{}
```

```
>>> print(dict([('poro', 0.2), ('perm', 100)]))
{'poro': 0.2, 'perm': 100}
>>> print(dict((('poro', 0.2), ('perm', 100))))
{'poro': 0.2, 'perm': 100}
```

PYTHON DATA STRUCTURES

Dictionaries

- Accessing a dictionary
- Elements of a dictionary are accessed (both for extraction and for re-assignment) simply by indexing with the keys.
- Unlike lists, dictionaries permit out-of-range indexing (for assignment purposes only)
  - That is, indexing with a key not yet present in the dictionary.
  - In such case, a new item (corresponding to the specified key and the assigned value) is created.
  - It is with such out-of-range indexing that a dictionary can be updated in a loop; dictionaries do not have the insert and append methods.

```
>>> print(data_dict)
{'area': 40, 'h': 15, 'poro': 0.2, 'swi': 0.31, 'boi': 1.2417}
>>> porosity = data_dict['poro'] #accessing for extraction
>>> print(porosity)
0.2
>>> data_dict['swi'] = 0.28 #accessing for re-assignment
>>> print(data_dict)
{'area': 40, 'h': 15, 'poro': 0.2, 'swi': 0.28, 'boi': 1.2417}
>>> data_dict['perm'] = 100 #out of range index, creating a new item
>>> print(data_dict)
{'area': 40, 'h': 15, 'poro': 0.2, 'swi': 0.28, 'boi': 1.2417, 'perm': 100}
```

PYTHON DATA STRUCTURES

Dictionaries

- Applications in petroleum engineering.
- We may re-write the function *stoiip* such that the five arguments may be passed as a single argument; being a dictionary object.

```
#########  A function to compute Stock Tank Oil Initially In-Place (STOIIP), N #########
### This function accepts a single argument; being a
### a dictionary
def stoiip_2(data):
    N = (7758*data['area']*data['thickness']*data['poro']*(1-data['swi']))/data['boi']
    return round(N, 2)
```

```
>>> # Sample call:
>>> data_dict = {'area': 40, 'thickness': 15, 'poro': 0.28, 'swi': 0.3, 'boi': 1.2}
>>> print(stoiip_2(data_dict))
760284.0
```

Dictionaries

- Applications in petroleum engineering.
- Sometimes, we want the output of a function to be a labelled collection of values; with each value corresponding to different entities acted upon by the function.
- Example:
  - Recall the function *stoiip_discretized*
  - In it current form; it returns a list containing the STOIIP values: one for each gridblock in the discretized reservoir model.
  - Now, we want it return a dictionary containing STOIIP values along side the gridblock labels as keys.
  - Such labeled output format becomes useful when there are several blocks and it is difficult to visually/mentally attach natural ordering to the returned collection.

PYTHON DATA STRUCTURES

Dictionaries

- Applications in petroleum engineering.
- Here is a segment of the modified function (*stoiip_dicretized_2*). Note the asterisked portions as the modifications done to the original function.

```
# initializing output variables
total_stoiip = 0
stoiip_dict ={} *

# the 'for' loop
for j in range(1,ny+1):
    for i in range(1,nx+1):
        block_n_order = (nx*(j-1))+i
    *   block_label = 'Block'+str(block_n_order) # to be used as key in stoiip_dict
        poro = poro_list[(block_n_order - 1)]
        sw = swi_list[(block_n_order - 1)]
        block_stoiip = (7758*area*h*poro*(1-sw))/boi
    *   stoiip_dict[block_label] = block_stoiip
        total_stoiip = total_stoiip + block_stoiip
return (total_stoiip, stoiip_dict)
```

Dictionaries

- Applications in petroleum engineering.
- Notice that in *stoiip_dicretized_2*, the label used as *key* for the dictionary is dynamically generated for each run of the 'for' loop.
- The label should be the string 'Block' concatenated with the block number (encoded as *block_n_order*); using the '+' operator.
- But *block_n_order* changes dynamically with each run of the 'for' loop.
- Therefore, the concatenation is done dynamically for each run of the loop; the label is stored as variable *block_label*.
- Ultimately, variable *block_label* is passed as *key* to the dictionary.

PYTHON DATA STRUCTURES

Dictionaries

- Applications in petroleum engineering.

```
>>> poro_vals = [0.18, 0.21, 0.32, 0.28]
>>> swi_vals = [0.23, 0.25, 0.29, 0.31]
>>> # Sample call: for stoiip_discretized -  the original
>>> print(stoiip_discretized(100, 100, 15, 2, 2, 1.2, poro_vals , swi_vals))
(173706468.75, [33601837.5, 38183906.25, 55081800.0, 46838925.00000001])
```

```
>>> poro_vals = [0.18, 0.21, 0.32, 0.28]
>>> swi_vals = [0.23, 0.25, 0.29, 0.31]
>>> # Sample call: for stoiip_discretized_2 -  the modified
>>> print(stoiip_discretized_2(100, 100, 15, 2, 2, 1.2, poro_vals , swi_vals))
(173706468.75, {'Block1': 33601837.5, 'Block2': 38183906.25, 'Block3': 55081800.0, 'Block4': 46838925.00000001})
```

Dictionaries

- Looping through a dictionary
- Just like lists and tuples; the items of a dictionary can be looped through.
- In such loops, the dictionary *keys* are the default iterator variables (not the *values*)
- If the dictionary *values* are needed in the loop, they are simply accessed by indexing with the *keys* (which are the respective iterator variable)
- In essence, a loop through a dictionary is a loop through its *keys*; except otherwise constructed.

PYTHON DATA STRUCTURES

Dictionaries

- Looping through a dictionary
- Just like lists and tuples; the items of a dictionary can be looped through.

```
>>> data_dict = {'area': 40, 'thickness': 15, 'poro': 0.28, 'swi': 0.3, 'boi': 1.2}
>>> for data_item in data_dict:
        print('The value of {0} is {1}'.format(data_item, data_dict[data_item]))


The value of area is 40
The value of thickness is 15
The value of poro is 0.28
The value of swi is 0.3
The value of boi is 1.2
```

```
if you == 'A top performer in the exam':
    print('See you in PDA Cluster')
else:
    print('Bye, for now')
```

```
>>>#TTOWG!
>>>print('…to the only wise God')
```