**Instructions**: You must read the material and create an outline of the topics in your OWN words.  Do not copy the text from the tutorials into your notes. Make sure your outline contains notes for each subsection of the reading assignment. Thoroughly cover each topic to show you have a firm understanding of the programming concept or construct.

| **Ques** | **NOTES:** |
| --- | --- |
| Functions | Decomposition:  This is the process of breaking down a very complex problem into smaller, more  manageable pieces. In programming, it is used to break down complex programs  into smaller chunks and helps reduce the time it takes to find and repair bugs within  such chunks.  These smaller chunks make up a larger program called functions in Python and may be  designed to solve common problems such as performing a login, presenting a menu,  collecting user input, formatting output, etc. Functions provide an opportunity to re-use  code blocks in different programs, and recycling existing code components speeds up  the application development process.  Based on the concept of abstraction, these functions are named, publish a required  set of inputs, produce a stated return value, and make it easy to use functions  effectively without knowing the details of their implementation. This is the very  definition of “abstraction”, another computational thinking practice. The use of  abstraction in programming is the process of assembling blocks of functionality  by name, and strategically arranging those blocks, based on an understanding of their  functionality, into a complex system. These individual pieces (functions) are commonly  connected by programmers via their inputs and outputs.  In summary, functions are game-changers because they 1) Break large tasks into smaller  chunks, 2) Provide recyclable solutions to common and/or complex tasks,  3) Ease the burden of code creation and maintenance (editing, updating, and debugging),  and 4) Facilitate abstraction.  Functions:  There are two types of functions: Bult-in functions and user-defined functions.  They all must be called; their names must always be followed by a set of  parentheses, which sometimes may contain values, and other times they may not.  Programmers eventually learn the requirements and purpose of built-in functions by  consulting categorized, online, and hard-copy references. This helps them to  understand what a function does, what information the function needs when it is called,  and what information it will return when it is finished executing. Here we see another  description of abstraction: understanding what something does without concerning  oneself with the intricate details. |
| Decomposition in Computational Thinking: Solving Problems More Effectively  Python Functions  Modular Programming and Modules | The above image illustrates the parts of a function.  Using (Calling) Functions:  To use a function, it must be called. It might be part of an algorithmic sequence and  will be called when its turn comes.  It may be called based on the following:   1. The value of a Boolean expression, or 2. Some integer values reach a certain threshold.   It may also be called by a particular user making a particular selection, as might  happen in a paint program. There are myriad ways to determine how and when a function  is called.  The fundamental responsibility of a programmer is to know how to call a function properly.  Let’s see an example:  The above program calculates the user data, computes gross and net pay, and reports  its results to the user.  Here, we have three functions defined: calcPay(), withholding(), and main(),  and the main() function serves as the conductor for an orchestra of functions.  From the sample program, if you look at the *main*() function to determine  how it works (lines 8-14), you will see that it first asks for user input (wage and time),  saving each in their respective variables. It then calls the function *calcPay*(),  sending that collected information to the function.  Information sent when calling a function is called its arguments, and in this case,  there are two arguments, user\_wage and user\_time, in that order. In this case,  the order is important.  The passage discusses the distinction between positional and keyword argument calls in  Python functions. It emphasizes that while positional arguments are assigned to function  parameters based on their order, keyword arguments are assigned based on their  names. For instance, in a function call like calcPay(hours=25, wage=16), the arguments  are explicitly associated with the parameters they correspond to. This method, termed  keyword or named argument calls, ensures clarity and flexibility in function calls,  as the order of arguments can vary without affecting their assignment to parameters.  Let’s see an example of this:  Looking at line 16 in the original code, we see that this line is responsible for initiating  the program's execution. It acts as a domino, and without this initial call, nothing  would have happened.  Also note that in lines 11 and 12, variables are assigned to receive the return values  from each function call, in this case calcPay() and withholding().  The returned values are saved or cached in the variables gross\_pay and taxes.  This completes the call-receive-process-return sequence programmers use all the time. Large complex programs consist largely of strategically placed sequences like this.  The following terms, though used interchangeably, are considered different by purists:   1. ***Procedures***are defined as independent code modules that fulfill some concrete 2. purpose or task. 3. ***Functions***are defined as a sequence of program instructions that perform a specific 4. task AND return a value, including None or the value of nothing. 5. ***Methods***are functions that exist within objects, which we haven't covered yet.   For simplicity's sake, this course will refer to all three as ***functions***.  Modularity:  Talented programmers put useful, related functions together in something called a module, and several modules can be collected together to form a library. Python has several built-in modules: os(operating system), random, math, time, sys(system), collections, and statistics. These contain collections of functions related to the module name (For instance, random number functions in the case of the random module). This speeds up and eases development through abstraction.  Computational thinking is described as the process of identifying a clear, defined, step-by-step solution to a complex problem. It involves breaking down a problem into smaller pieces, recognizing patterns, and eliminating extraneous details so that a step-by-step solution can be defined and replicated.  What Does Decomposition Mean in Computational Thinking?  This is the process of breaking down a problem into several smaller problems that can more easily be addressed. It is considered an effective method of solving complex problems as it helps identify patterns, eliminate extraneous details, and solve the problem step by step instead of trying to do so all at once.  Examples of Decomposition in Everyday Life  Hosting a Holiday Dinner:  Here, we use decomposition to select the menu, enlist support from others in the kitchen, task people with what to bring, determine the process of cooking what and when and set the time for the event. If you went to the grocery store, you would use decomposition to build your grocery list, guide your direction as you shopped each aisle, and even drive to and from the store.  Change Initiative at Work:  If you implemented a new program at work, then you would mostly use decomposition to build your strategic plan. This would include defining the program’s vision, strategizing how to gain buy-in, developing goals for the initiative, and creating a step-by-step process for achieving those goals.  Examples of Decomposition in Curriculum:  Decomposition is a powerful tool that guides how we approach everyday projects and tasks.  It is also something employed by students. Some examples of accentuating these in the curriculum include:   * **English Language Arts:**Students analyze themes in a text by first answering: Who is the protagonist and antagonist? Where is the setting? What is the conflict? What is the resolution? * **Mathematics:** Students find the area of different shapes by decomposing them into triangles. * **Science:**Students research the different organs in order to understand how the human body digests food. * **Social Studies:**Students explore a different culture by studying the traditions, history, and norms that comprise it. * **Languages:**Students learn about sentence structure in a foreign language by breaking it down into different parts like subject, verb, and object. * **Arts:**Students work to build the set for a play by reviewing the scenes to determine their setting and prop needs.   Example of Decomposition in Computer Science:  From a computer science and coding perspective, decomposition can come into play when students are programming a new game where they have to consider the characters, setting, and plot, as well as consider how different actions will take place, how they will be deployed, and so much more.  Final Thoughts:  Decomposition is deeply ingrained in how we function daily and address both big and small problems, and it exists with students, but students need to learn to use it effectively to tackle their projects and tasks. It teaches students to embrace ambiguity and equips them with the confidence to learn new things.  Functions are block of code that perform a specific task, e.g. suppose we need to create a program that makes a circle and also colors it. We would create two functions to the solve the problem:   1. Function to create a circle, and 2. Function to color the shape.   Create a Function:  def greet():  print(“Hello World!”)  Here are the different parts of the program:  Here, we create a simple function named greet() that prints “Hello World!” and it should be noted that when creating a function, attention should be given to indentation, which are paces  at the start of a code line.  Calling a Function:  To call a function after creating it we use the syntax:  greet()  Example: Python Function Call  def greet():  print(“Hello World!”)  # Call the function  Greet()  print(‘Outside function’)  Output:  Hello World!  Outside function  Here’s how the control of the program flows:  Here,   1. When the function greet() is called, the program's control transfers to the function definition. 2. All the code inside the function is executed. 3. The control of the program jumps to the next statement after the function call.   Python Function Arguments:  def greet(name):  print(“Hello”, name)  # pass argument  greet(“John”)  Out put:  Hello John  Here, we passed ‘John’ as an argument to the greet() function. We can also pass a different  argument say “David” and we would call the function like so:  greet(David)  Output:  Hello David  Example: Function to Add Two Numbers:    # function with two arguments  def add\_numbers(num1, num2):  sum = num1 + num2  print("Sum: ", sum)  # function call with two values  add\_numbers(5, 4)  Output:  Sum: 9  Above we created a function named add\_numbers() with arguments: num1 and num2  Here is the flow:  The return Statement:  We use the return statement to return a value from a function.  # function definition  def find\_square(num):  result = num \* num  return result  # function call  square = find\_square(3)  print('Square:', square)  Output:  Square: 9  Above we create a function name find\_square() that accepts a number and returns the square  of the number:  Note that the return statement also denotes that the function has ended. Any code after the  return statement is not executed.  The pass Statement:  This is used as a placeholder for future code, preventing errors from empty code blocks.  Example:  def future\_function():  pass  # this will execute without any action or error  future\_function()  Python Library Functions:  Python provides some built-in functions that can be directly used in a program. There’s no need to create these functions we just call them.  Some Python library functions are:   1. [print()](https://www.programiz.com/python-programming/methods/built-in/print) - prints the string inside the quotation marks 2. sqrt() - returns the square root of a number 3. [pow()](https://www.programiz.com/python-programming/methods/built-in/pow) - returns the power of a number   These are defined inside the module and to use them, we must include the module inside our  Program.  Example:  import math  # sqrt computes the square root  square\_root = math.sqrt(4)  print("Square Root of 4 is",square\_root)  # pow() comptes the power  power = pow(2, 3)  print("2 to the power 3 is",power)  output  Square Root of 4 is 2.0  2 to the power 3 is 8  Modular Programming:  This software design technique is based on the general principle of modular design. Modular design is an approach that has been proven indispensable in engineering and is a standard in building efficient software.  Modular design means that a complex system is broken down into smaller parts of components, i.e., modules. These can be independently created and tested.  If you want to develop readable, reliable, and maintainable programs with little effort, then you have to use some modular software design. When using modules, each should be dependent on a few other modules.  Import Modules:  Every Python file with the extension “.py” and consisting of proper Python code can be seen as a module. A module can contain arbitrary objects, for example, files, classes, or attributes, accessed after importing the said module.  Example module import:  import math  Usage:  math.pi  Output:  3.14192653589793  math.sin(math.pi/2)  To import several module in one line:  import math, random  If we only need certain objects of a module then, we can import only those:  from math import sin, pi  with the above import style we can access the sin and pi functions directly without prefixing them with math.  We can also use the “\*” to import object in the any namespace:  from math import \*  Note: it is not recommended to use the asterisk notation unless when working in interactive Python shell. A good reason not to use “\*” is the fact that the origin of a name can be quite obscure, not knowing from which module is was imported. Example:  from numpy import \*  from math import \*  print(sin(3))  Output:  0.1411200080598672  sin(3)  Output:  0.1411200080598672  Swapping the import statement from the above example also give us the same result, but we are unable to tell which module is being used.  Since using the “\*” helps with having to type a lot we could also use renaming of namespace to help with this. Example  import numpy 🡺 import numpy as np  Now to use it:  np.diag([3,11,7,9])  Designing and Writing Modules  Any Python file with proper code containing definitions and statements is a module. The module name is molded from the file name by removing the suffix .py. For example, if the file name is Fibonacci.py, the module is Fibonacci.  Let’s save the following code to a file named fibonacci.py:  def fib(n):  if n == 0:  return 0  elif n == 1:  return 1  else:  return fib(n-1) + fib(n-2)  def ifib(n):  a, b = 0, 1  for i in range(n):  a, b = b, a + b  return a  Usage:  import fibonacci  fibonacci.fib(7)  Output:  13  We can make it’s simpler by assigning a local name to a module function:  fib = fibonacci.ifig  fib(10)  More on Modules:  Modules are imported once, though in Python 2.x, it was possible to use the reload() function to re-import a module; it isn’t available in Python 3.x and has to be now accessed from the imp module like so:  from imp import reload  but in 3.4, we now use the “importlib” module as the imp.reload is marked as deprecated.  A module has a private symbol table, which is used as the global symbol table by all functions defined in the module. This is a way to prevent a global variable of a module from accidentally clashing with a user's global variable with the same name. Global variables of a module can be accessed with the same notation as functions, i.e., modname.name A module can import other modules. It is customary to place all import statements at the beginning of a module or a script.  Importing Names from a Module Directly:  Names from a module can directly be imported into the importing module’s symbol table:  from fibonacci import fib, ifib  ifib(500)  This does not introduce the module name from which the imports are taken in the local symbol table. It's possible but not recommended to import all names defined in a module, except those beginning with an underscore "\_":  from fibonacci import \*  fib(500)  This shouldn't be done in scripts but it's possible to use it in interactive sessions to save typing.  Executing Modules as Scripts  To do this we do:  python fibo.py  The module that has been started as a script will be executed as if it had been imported, but with one exception: The system variable **name** is set to "**main**." So, it's possible to program different behaviors into a module for the two cases. With the following conditional statement, the file can be used as a module or as a script, but only if it is run as a script the method fib will be started with a command line argument:  if \_\_name\_\_ == "\_\_main\_\_":  import sys  fib(int(sys.argv[1])) |
|  | If it is run as a script, we get the following output:  $ python fibo.py 50  1 1 2 3 5 8 13 21 34  If it is imported, the code in the if block will not be executed:  import fibo  Renaming a Namespace:  We can change the name of a namespace while importing a module:  import math as mathematics  print(mathematics.cos(mathematics.pi))  After this import there exists a namespace mathematics but no namespace math. It's possible to import just a few methods from a module:  **from** **math** **import** pi,**pow** **as** power, sin **as** sinus  power(2,3)  Kinds of Modules:  There are different kinds of modules:  There are different kinds of modules:  • Those written in Python have the suffix .py  • Dynamically linked C modules with suffixes of .dll, .pyd, .so, .sl, ...  • C-Modules linked with the Interpreter. It's possible to get a complete list of these modules:  **import** **sys**  **print**(sys.builtin\_module\_names)  **OUTPUT:**  ('\_abc', '\_ast', '\_bisect', '\_blake2', '\_codecs', '\_codecs\_cn', '\_codecs\_hk', '\_codecs\_iso2022', '\_codecs\_jp', '\_codecs\_kr', '\_codecs\_tw', '\_collections', '\_contextvars', '\_csv', '\_datetime', '\_functools', '\_heapq', '\_imp', '\_io', '\_json', '\_locale', '\_lsprof', '\_md5', '\_multibytecodec', '\_opcode', '\_operator', '\_pickle', '\_random', '\_sha1', '\_sha256', '\_sha3', '\_sha512', '\_signal', '\_sre', '\_stat', '\_string', '\_struct', '\_symtable', '\_thread', '\_tracemalloc', '\_warnings', '\_weakref', '\_winapi', 'array', 'atexit', 'audioop', 'binascii', 'builtins', 'cmath', 'errno', 'faulthandler', 'gc', 'itertools', 'marshal', 'math', 'mmap', 'msvcrt', 'nt', 'parser', 'sys', 'time', 'winreg', 'xxsubtype', 'zipimport', 'zlib')  Module Search Path:  Paths to modules are searched in this order:   1. The directory of the top-level file, i.e. the file being executed. 2. The directories of PYTHONPATH, if this global environment variable of your operating system is set. 3. standard installation path Linux/Unix e.g. in /usr/lib/python3.5.   It's possible to find out where a module is located after it has been imported:  import numpy  numpy.**file**  '/usr/lib/python3/dist-packages/numpy/**init**.py'  import random  random.**file**   '/usr/lib/python3.5/random.py'  Note: The **file** attribute doesn't always exist. This is the case with modules which are statically linked C libraries.  **import** **math**  math.\_\_file\_\_  **OUTPUT:**  **---------------------------------------------------------------------------**  **AttributeError** Traceback (most recent call last)  **<ipython-input-4-bb98ec32d2a8>** in <module>  1 **import** math  **----> 2** math.\_\_file\_\_  **AttributeError**: module 'math' has no attribute '\_\_file\_\_'  Content of a Module:  With the built-in function dir() and the name of the module as an argument, you can list all valid attributes and methods for that module.  import math  dir(math)  OUTPUT:  ['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']  But calling dir() without an argument, a list with the names in the current local scope is returned:  import math  cities = ["New York", "Toronto", "Berlin", "Washington", "Amsterdam", "Hamburg"]  dir()  OUTPUT:  ['In, 'Out, '\_, '\_1, '\_\_, '\_\_\_, '\_\_builtin\_\_, '\_\_builtins\_\_, '\_\_doc\_\_, '\_\_loader\_\_, '\_\_name\_\_, '\_\_package\_\_, '\_\_spec\_\_, '\_dh, '\_i, '\_i1, '\_i2, '\_ih, '\_ii, '\_iii, '\_oh, 'builtins, 'cities, 'exit, 'get\_ipython, 'math, 'quit']  It is also possible to get a list of the Built-in functions, exceptions, and other objects by importing the builtins module:  **import** **builtins**  **dir**(builtins)  **OUTPUT:**  ['ArithmeticError, 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '\_\_IPYTHON\_\_', '\_\_build\_class\_\_', '\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec', 'filter', 'float', 'format', 'frozenset', 'get\_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']  Packages:  Modules can be organized into packages to group them based on common functionality. |
|  |  |