**Python Part 1 - Introduction**

|  |  |
| --- | --- |
| **Python Overview**   * We will treat Python as an **interpreted** language. * Python has *tons* of applications. We will use it for scripting/automation. * **Comments** use **#.** * **Syntax** is different than most languages you may be used to. Notably, semicolons and curly brackets are *generally* not utilized. * Often, **.py** is used as the file extension for Python files. * The Python shebang (first line and must begin in the first column) can be used to tell the shell to invoke Python.   #!/usr/bin/env python3   * **Note:** major and minor versions can be specified in the shebang #!/usr/bin/env python**2.4** * By pointing the shebang to **env**, it will check the user's $PATH, thus making the script more portable. Alternatively, you can use #!/usr/bin/python3 as you have seen before. * Available on Unix implementations and Microsoft Windows   **Versions**   * There are two active versions of Python: 2 (2.7) and 3. You will often find both versions installed on a system (be careful to use the correct one) * Python 2 is well-established with many libraries, but it has many design flaws that Python 3 has improved/fixed * Python 3 is **not** backwards compatible with Python 2 * Development of Python 2 will stop in 2020 * **These notes will use Python 3** * python -V will show you which version the "python" command points to. Just use python3 if you aren’t sure.   The examples in these notes can be found in /usr/local/rslavin/cs3423/python | **Example 1**: This example is meant to give you an idea of a basic python script. You will learn more about the details later. The script lists '.py' files within a specified directory are larger than a specified size.  $ vi example1.py  #!/usr/bin/env python3  import os  import sys  import glob  count = 0  # check for number of arguments  if len(sys.argv) != 3:  print("Usage: " + sys.argv[0] + " <dirname> <size>")  sys.exit(1)  # check that the first arg is a directory  if not os.path.isdir(sys.argv[1]):  print(sys.argv[1] + " is not a directory")  sys.exit(1)  # get the .py files  py\_files = glob.glob(sys.argv[1] + "/\*.py")  # check their sizes  for pf in py\_files:  stat = os.stat(pf)  if stat.st\_size > int(sys.argv[2]):  count += 1  print(pf + " - " + str(stat.st\_size))  print(str(count) + " .py files found with size greater than "  + sys.argv[2])  $ chmod u+x example1.py  $ ./example1.py . 500  ./example1.py - 656  ./example1.py - 656  ./func.py - 553  ./lists.py - 657  ./oddlines.py - 639  ./oddlines2.py - 556  5 .py files found with size greater than 500 |
| **Data**   * **No** explicit **declaration** of variables (Python is dynamically typed). Variables receive their data type, structure, size, and value from assignment. * **Variables** are assigned and referenced by their name: *variable* * **Lists** are a type of variable similar to an array. They are zero-indexed. * Associative arrays are called **dictionaries**or **hashes** * Various list/dictionary tools are provided (e.g., slicing) * **Arguments** can be accessed with sys.argv which is the list of command-line arguments. The getopt and argparse modules are also available for easy argument/option parsing (you are encouraged to look into these modules on your own).   **Sequence Control and Code Blocks**   * The standard flow control statements (**if**, **for**, **while**) exist but use different syntax than you are probably used to. Instead of curly braces, blocks of code are signified by indentation. * **Functions** are easily specified as def statements. |  |
| **Invocation**  To invoke a script named example1.py:   * $ python3 example1.py *arguments* * If it has executable permission and contains the Python shebang: $ example1.py *arguments*   **Interactive Shell**  Python includes an interactive shell which, like other shells, will parse Python code as it is entered. In these notes we will use **>>>** to signify the Python interactive shell prompt (much like $ for shell). You can start the interactive shell by simply running the corresponding interpreter (e.g., python3).  Note that the interactive shell will print the result of any expressions you execute *without calling print()*. This is useful for learning but is not the same behavior as when running a Python program. | **Example 2**: Interactive shell  $ python3  >>> a = "hello"  >>> print(a)  hello  >>> a  hello |
| **Tabs and Spaces**  Python can be extremely finicky with whitespace. Instead of using curly braces for code blocks as you may be used to, Python code blocks are specified by indentation and newlines. The Python interpreter recognizes either tabs or spaces for indentation. You can even mix tabs and spaces in your code **AS LONG AS YOU ARE CONSISTENT WITHIN BLOCKS** (this is mostly so it’s easy to copy and paste code)**.** Obviously, it is good practice to simply be consistent throughout your code.It’s a great idea to check out the **PEP8 Style Guide for Python Code** to learn the coding standards (<https://python.org/dev/peps/pep-0008/>).  Most Python programmers use four spaces in place of a tab for indentation. You can adjust your editor to do this automatically for you. I won’t get into the tabs vs space debate. Just be consistent.  The following lines can be useful in your ~/.vimrc file if you want to use spaces instead of tabs.  " set tabs to be equal to four spaces  set tabstop=4  " set auto-tabbing (for blocks) to be four spaces  set shiftwidth=4  " make the tab key automatically expand to spaces  set expandtab  You can use the :retab command to convert existing tabs to your rules.  If you want to enable these rules only for Python files, do the following instead.  **autocmd Filetype python** set**l** expandtab tabstop=4 shiftwidth=4 | |
| **Variables**  Variables (actually ***Object References***) do not require declaration nor specification of their type. Python is **dynamically typed** meaning variables can change their type at runtime.  Assignment (or *binding*) is done with the = operator. Most basic types are **immutable**, meaning they cannot be changed once in memory. In practice, this mostly isn’t noticeable, but it’s important to know for some operations. We’ll see more later.  An assignment of a variable to a variable will create a **reference** to the samethe **object** in memory.  1  a = 1  "hello"  b = a  a = "hello"  In Python, it is convention to use snake\_case for variable names. | **Example 3**: dynamic types and assignment  $ vi vars.py  a = 1  b = a  print("a: " + str(a))  print("b: " + str(b))  print ("Assigning 'hello' to a")  a = "hello"  print("a: " + a) # a is a string now  print("b: " + str(b)) # b still points to a's old value  $ python3 vars.py  a: 1  b: 1  Assigning 'hello' to a  a: hello  b: 1 |
| **Datatypes**  Python has several built-in data types. For now, we will focus on three of the basic ones: strings (str), integers (int), and Boolean (bool). Note that since we are really dealing with objects, types can be considered **classes** (i.e., an object is an *instance* of a class) and thus have **methods**.  **Strings** are specified with double or single quotes. There is no difference between the two other than if a double or single quote is used inside of the string itself. In such a case, simply use the opposite type of quote.  some\_string = 'The dog said "bark!"'  *Three quotes* (double or single)can be used for multi-line strings (i.e., newline characters will be included automatically). These are often used, unassigned, as a form of code commenting.  **Integers** exist as positive and negative numbers without decimals (**floats** have decimals). If a number is specified with quotes, it will instead be interpreted as a string. The size of the integer is limited only by available memory.  some\_number = -342  some\_other\_number = 123.03  some\_huge\_number = 19872304972340982304978230492734092  Conversion between types can be done with the corresponding *datatype*() function where *datatype* is the type to which to convert.  some\_int = int("342")  some\_str = str(342)  **Booleans** exist as either True or False. However, any built-in data type can be interpreted as “truthy” or “falsey” in conditions (which can help keep code readable and concise). Expressions evaluating to False are empty sequences or collections (e.g., empty strings or lists), numeric values of zero, the object None, or the Boolean constant False. Everything else is True.  **Boolean** **False** Values: False, 0, 0.0, '', []  **Boolean** **True** Values: True, 1, 5, '00', 'true', 'false'  Why is 'false' evaluated to True?  ??  Booleans can be combined with the Boolean operators **and**, **or**, and **not**. We’ll see more of this later in flow control. | **Example 4**: Datatypes and conversions  $ vi datatypes.py  #!/usr/bin/env python3  pi = 3.14  radius = 3  circumference = 2 \* pi \* radius  # ints must be converted to strings to concatenate them  msg = "c = 2 \* " + str(pi) + " \* " + str(radius)  print(msg)  print("c = " + str(circumference))  $ chmod u+x datatypes.py  $ datatypes.py  c = 2 \* 3.14 \* 3  c = 18.84 |
| **Lists**  Python includes many datatypes for holding collections of data including associative arrays, sets, and tuples. For now, we will consider the list.  **Lists** are similar to arrays in that they reference many data items and are zero indexed. A crucial difference is that, unlike arrays, lists are *not contiguous in memory.* Each element is really a reference to an object with its own data type.  Lists can be initialized with square brackets (**[]**).  >>> my\_list = ["cat", 23, "bird"]  Recall that all types are really just classes and all variables are instances of those classes (i.e., objects). This means we can have **methods** for those objects.  Objects can be added to a list with **append()**.  >>> my\_list.append("new element")  >>> print(my\_list)  ["cat", 23, "bird", **"new element"**]  Since lists have a "size", they can be passed to **len()** to retrieve the length.  >>> print(len(my\_list))  3  Note that strings are also a datatype with a size and thus can be passed into len().  >>> print(len("hello world"))  11  **Access** and **Assignment** are accomplished with square brackets.  Element access:  *list*[*index*]  Element assignment:  *list*[*index*] = *expr*  >>> print(my\_list[1])  23  >>> my\_list[1] = "twenty three"  >>> print(my\_list)  ["cat", **"twenty three"**, "bird", "new element"]  Lists can be **sliced** to extract sub-lists. A slice of an array can be accessed by specifying:  *list*[*start*:*end*]  where *start* and *end* denote the indices of the first and last elements (it's important to remember that list[end] is *not* included in the slice). These can be left blank to specify the beginning or end of the list.  A step value can be included by adding :step. This can be negative to move backwards through the list.  Unlike basic types, lists are *mutable* meaning the references they hold can be modified (as seen with assignment above). | **Example 5**: lists  $ vi lists.py  numbers = ["one", "two", "three", "four", "five"]  colors = ["red", "orange", "YELLOW", "beige", "grey", "violet"]  # access  print("number[0] is " + numbers[0])  print("The last index in numbers is " + str(len(numbers)-1))  print("The last number is " + numbers[len(numbers)-1])  # slicing  print("Slicing odd and even numbers:")  odd\_numbers = numbers[::2]  even\_numbers = numbers[1::2]  print(odd\_numbers)  print(even\_numbers)  # assignment  print("Fixing colors with assignments:")  colors[2] = "yellow"  colors[3:5] = ["green", "blue", "indigo"]  print(colors)  # strings as lists  print("Substring with list notation:")  greeting = "hello world"  print(greeting[6:])  $ python3 lists.py  number[0] is one  The last index in numbers is 4  The last number is five  Slicing odd and even numbers:  ['one', 'three', 'five']  ['two', 'four']  Fixing colors with assignments:  ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']  Substring with list notation:  World  Since we can treat strings like lists, would greeting[0] = "H" work?  ?? |
| **Important Operators**  **Integers**  *arg1* + *arg2* addition  *arg1* - *arg2* subtraction  *arg1* \* *arg2* multiplication  *arg1* / *arg2* division (the result produces a **floating-point**, unlike many other languages)  *arg1* // *arg2*  truncating division (the result produces a floored integer)  *arg1* % *arg2* modulus  *var = expr* assignment (data type of *expr* determines the data type of the resulting object).  *var += expr* add *expr* to the specified variable  *var -= expr* subtract *expr* to the specified variable  Note that there are no increment (++) or decrement (--) operators.  **Strings (and Lists)**  *arg1 + arg2* concatenation  *arg1 +=* *arg2* append  **Some Notes on Assignment**  Recall that basic data types are immutable, meaning they cannot be changed once they have been placed in memory. This means that re-assignment of a variable (e.g., a += 1) will create a new object and point the variable reference to it. | **Example 6**: operators  $ vi ops.py  import sys  if len(sys.argv) != 3:  print("Usage: " + sys.argv[0] + " <int> <int>")  sys.exit(1)  # note that this can result in an error if ints aren't passed  # we'll learn how to handle this later  int\_add = int(sys.argv[1]) + int(sys.argv[2])  str\_add = sys.argv[1] + sys.argv[2] # sys.argv contains strings  print("Adding as integers results in " + str(int\_add))  print("Adding as strings results in " + str\_add)  $ python3 ./ops.py  $ ./ops.py 1 2  Adding as integers results in 3  Adding as strings results in 12 |
| **Logical Operators**  Python provides the typical comparison operators which can be used to compare objects.   |  |  | | --- | --- | | **Operator** | **Returns** | | == | True if equal | | != | True if not equal | | < | True if less than | | > | True if greater than | | <= | True if less than or equal to | | >= | True if greater than or equal to |   The logical operators "and", "or", and "not"can be used to create compound logical expressions (similar to &&, ||, and ! in other languages). Python uses short-circuit logic to evaluate a compound expression from left to right.  Comparison operators can also be **chained** together as shorthand for using logical **and**.  >>> a = 5  >>> print(1 <= a <= 10)  True  How could the above condition be rewritten using logical and?  ??  The **identity** operator, "is", can be used to test whether two variables reference the same object. Equality of the *values* of two variables is tested with "==".  >>> a = "hello world"  >>> b = "hello world"  >>> print(a == b)  True  >>> print(a is b)  False  >>> b = a  >>> a is b  True  The **membership** operator, "in", can be used to test whether an object *value* exists in a collection such as a list or string.  >>> some\_string = "hello world"  >>> some\_list = ["hello", "world"]  >>> print("ll" in some\_string)  True  >>> print("hello" in some\_list)  True  >>> print("hi" **not** in some\_list)  True | **Example 7**: logical operators  $ vi logic.py  #!/usr/bin/env python3  import sys  if len(sys.argv) != 2:  print("Usage: " + sys.argv[0] + " <string>")  sys.exit(1)  str\_length = len(sys.argv[1])  if str\_length > 5 and not str\_length % 2:  print("'" + sys.argv[1]  + "' is longer than 5 chars and has even length.")  $ chmod +x ./logic.py  $ ./logic.py 'hello world!'  'hello world!' is longer than 5 chars and has even length. |
| **Reading Input from stdin**  input(*prompt*)   * Prints *prompt*, if supplied, to stdout. * Reads a line from stdin, **removing the newline character.** * Returns the line as a **string**. * If EOF is received, input() raises an EOFError exception.   What if we want to read an integer from stdin?  ?? | **Example 8:** input  $ vi input.py  #!/usr/bin/env python3  adjective = input("Enter an adjective: ")  noun = input("Enter a noun: ")  noun2 = input("Enter a noun: ")  time = input("Enter a unit of time: ")  print("After storing the " + adjective + " " + noun  + " in his " + noun2 + " for a " + time +  ", he began to regret his life choices.")  $ chmod +x input.py  $ ./input.py  Enter an adjective: stinky  Enter a noun: cheese  Enter a noun: nose  Enter a unit of time: month  After storing the stinky cheese in his nose for a month, he began to regret his life decisions |
| **Exceptions**  Many Python functions, and eventually some of your own, raise exceptions when an error occurs. We can prepare for potential exceptions and handle them with a *try* statement.  try:  *suite*  except *exception1* as *var*:  *exception\_suite1*  except *exception2* as *var*:  *exception\_suite2*  In the above code, *suite* is the code we hope to execute. If an exception is raised during execution, the following *except* blocks are checked to see if the exception raised matches the corresponding exception. For the first one which it does, the corresponding *exception\_suite* is executed with *var* populated with the exception type.  Some common exception types:   * TypeError Inappropriate type supplied to a function or statement * ValueError Correct type, but bad data (e.g. int("one")) * EOFError Reached EOF * Exception All built-in, non-system-exiting exceptions are derived from this class. Will catch most exceptions, but it is generally bad practice to just use this for everything. Good for the last except block. | **Example 9:** exceptions **–** reading ints from input()  $ vi except.py  #!/usr/bin/env python3  valid = False  while not valid:  try:  a = int(input("Enter an integer: "))  b = int(input("Enter another integer: "))  valid = True # the above lines didn't cause an exception  except ValueError:  print("You didn't enter an int! Start over.")  valid = False  except Exception as e:  print("An exception occurred that I didn't account for: "  + str(e))  valid = False  total = a + b  print("Sum: " + str(total))  $ chmod +x except.py  $ ./except.py  Enter an integer: 1  Enter another integer: one  You didn't enter an int! Start over.  Enter an integer: 1  Enter another integer: 2  Sum: 3 |
| **Flow Control**  In Python, blocks of code, or ***suites***, are represented when consecutive statements are at the same indentation level. Flow control statements consist of clause headers containing the corresponding condition which guards the suites.  It is important to remember that suites are *required* for all clauses. If you need to have an empty code block, the **pass** keyword can be used as a statement which does nothing. This is like a no-op statement or empty curly brackets in other languages.  Note that suites can also be represented as semicolon-separated statements on the same line.  **while loop**  while *boolean\_expression*:  *suite*  **if-elif-else**  if *boolean\_expression*:  *suite*  elif b*oolean\_expression2:*  *suite2*  else *boolean\_expression3:*  *suite3* | **Example 10:** print each character per line  $ vi flow\_while.py  #!/usr/bin/env python3  import sys  if len(sys.argv) != 2:  print("Usage: " + sys.argv[0] + "<string>")  sys.exit(1)  count = 0  word = sys.argv[1]  while count < len(word):  print(word[count])  count += 1  $ chmod +x flow\_while.py  $ ./flow\_while.py hello  h  e  l  l  o |
| **for … in Statement**  The **for** statement allows for iteration through *iterable* objects such as strings and lists.  for *variable* in *iterable*:  *suite*  Note that the “in” keyword used in this context is different than when used for testing membership. Here, *variable* represents the element *in* the iterable object from the current iteration. | **Example 11:** implementation of flow\_while.py with **for** loop  $ vi flow\_for.py  #!/usr/bin/env python3  import sys  if len(sys.argv) != 2:  print("Usage: " + sys.argv[0] + " <string>")  sys.exit(1)  for letter in sys.argv[1]:  print(letter) $ perl sumN.pl 5  $ chmod +x flow\_for.py  $ ./flow\_for.py hello  h  e  l  l  o |
| **break and continue**  The **break** and **continue** statements can be used in **while** and **for** loops to add more control to iteration execution.  **break** *exits* the loop.  **continue** *continues* to the next iteration, skipping the remaining code in the suite for the current iteration.  Both statements will typically be guarded by an **if** statement to decide whether they should be executed. | **Example 12:** breaking an infinite loop  $ vi break.py  #!/usr/bin/env python3  # continuously read from stdin  while True:  # stop when EOF is reached (or ctrl+d)  try:  course = input("Enter your favorite course: ")  except EOFError:  print("done reading!")  break  if course == "systems" or course == "systems programming":  print("good choice")  elif course == "software engineering":  print("not bad")  else:  print("wrong")  print("done with loop!")  $ chmod u+x break.py  $ ./break.py  Enter your favorite course: history  wrong  Enter your favorite course: linear algebra  wrong  Enter your favorite course: software engineering  not bad  Enter your favorite course: systems  good choice  Enter your favorite course: <ctrl+d> done reading!  done with loop! |
| **Exercise**: Show code for the Python program "vowel.py" which will examine a string, sys.argv[1], and do the following:   * Print every character before the first vowel * Print the length of the entire string   Notes:   * The program should not require iterating through the entire string if a vowel exists before the last character. * Verify that the program is passed one argument. * Make sure to make " vowel.py" executable.   $ vowel.py "hello"  h  5  $ vowel.py "abcdefg"  7 | #!/usr/bin/env python3  import sys  VOWELS = ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']  if ?? != 2:  print("Usage: " + sys.argv[0] + " <string>")  sys.exit(1)  # print only the first consonants in sys.argv[1]  ??  # print the length  ?? |
| **Files**  We can always use input redirection to force input() to read from a file. However, redirection is up to the user. If we want to explicitly read from a file, the following functions are useful.  open(*filename, mode*)   * Opens a file for reading or writing * *filename* indicates the path to the file for opening. * *mode* indicates how to open the file (see table below). * Returns a **file** object which has various methods for manipulating the file. * Note that this function can take many other parameters to customize how the file is opened. Refer to the docs for more info. * Raises various IOErrors for permission/existence issues * Some open modes (the ones we are interested in):  |  |  | | --- | --- | | **'r'** | Open for reading (raises FileNotFound error if the file doesn't exist) | | **'w'** | Open for writing (truncates) | | **'a'** | Open for writing (append) | | **'r+'** | Open for reading and writing (start at beginning) | | **'w+'** | Open for reading and writing (truncates) | | **'a+'** | Open for reading and writing (start at the end) | | **'x'** | Open for creation (raises FileExistsError if the file exists) |   f.read(*bytes*)   * Reads a specified number of bytes from the file f. * This is a **method**, meaning it must be called *on* the **file object** (f)**.** * *size* (optional) indicates the number of bytes to be read. If omitted or negative, the entire file is read. * Returns the bytes as a string. * The file object will keep track of where we are in the file for subsequent reads. * Returns an empty string when EOF is reached.   f.readline()   * Returns a single line from the file f * Includes newline character for all lines except the last * Blank lines are represented with "\n" * End of file returns empty string * **NOTE** that file objects are iterable. Using a file in a for loop will effectively call readline() for each iteration.   f.readlines()   * Reads all lines from the file f * Returns a list of lines   f.seek(*byte\_offset, from\_what*)   * Changes the current position in the file * byte\_offset specifies how far to move in the file. Can be negative. * from\_what can be 0 for beginning of the file, 1 for current position, and 2 for the end. If omitted, defaults to beginning.   f.write(*string*)   * Writes string to the file at the current position * Returns number of characters written   f.close()   * Closes the file, thus freeing up the object from memory and flushing any internal buffers\*.   Most file exceptions descend from IOError.  While not necessary, it’s a good idea to use the **with** keyword with files. This automatically closes the file after the suite finishes  **with** open("somefile", "w") **as** f:  f.write("hello world!")    # f is no longer open here  \*Python uses an internal buffer to write to the disk more efficiently (we’ll learn more about this with C later). If you need to force Python to write to the disk immediately, use f.flush(). | **Example 13-1:** reading and writing. Uses **infile.txt**  $ vi oddlines.py  import sys  if len(sys.argv) != 3:  print("Usage: " + sys.argv[0] + " <inpath> <outpath>")  sys.exit(1)  line\_number = 1  try:  infile = open(sys.argv[1], 'r')  # write odd lines to out file  with open(sys.argv[2], 'w') as outfile:  for line in infile:  if line\_number % 2:  outfile.write(line)  line\_number += 1  except IOError: # catches FileNotFound and PermissionError  print("Error opening one of the files.")  sys.exit(1)  # close in file (out file was closed with 'with' statement)  infile.close()  $ cat infile.txt  One  two  three  four  five  $ python3 oddlines.py infile.txt outfile.txt  $ cat outfile.txt  one  three  five  **Example 13-2:** alternative version oddlines.py  $ vi oddlines2.py  import sys  if len(sys.argv) != 3:  print("Usage: " + sys.argv[0] + " <inpath> <outpath>")  sys.exit(1)  try:  # extract odd lines  with open(sys.argv[1], 'r') as infile:  oddlines = infile.readlines()[::2]  # write odd lines to out file  with open(sys.argv[2], 'w') as outfile:  outfile.write("".join(oddlines))  except IOError: # catches FileNotFound and PermissionError  print("Error opening one of the files.")  sys.exit(1)  $ cat infile.txt  One  two  three  four  five  $ python3 oddlines2.py infile.txt outfile.txt  $ cat outfile.txt  one  three  five |
| **Packages**  Python includes many libraries with lots of functionality useful in systems programming. To import a known library, use the "import" keyword followed by the library name. You can important specific sub-packages/modules of a library with dot notation.  Packages which must be imported are highlighted in blue.  **Checking permissions**  os.access(*file\_path, mode*)   * Returns True if the file\_path has the access specified by mode for the current uid/gid. Returns False otherwise.   Modes   * os.**F\_OK** – exists * os.**R\_OK** – readable * os.**W\_OK** – writeable * os.**X\_OK** – executable   **Checking for type**  os.path.isfile(*file\_path*)   * Returns True if the file\_path is that of a regular file. Returns False otherwise.   os.path.isdir(*file\_path*)   * Returns True if the file\_path is that of a directory. Returns False otherwise.     **Listing Files**  os.listdir(*file\_path*)   * Returns a list of files in the path.   glob.glob(*file\_path, recursive=False*)   * Returns a list of files matching the file pattern in file\_path. * Pass in "recursive=True" to recursively retrieve files   There are tons of packages available in the Python standard library. Check <https://docs.python.org/3/library/> for more info. More packages will be introduced throughout these notes. **Keep an eye out for new imports and calls to library functions.** | **Example 14:** interpret file attributes using modules  $ vi file\_attr.py  #!/usr/bin/env python3  import os  import sys  if len(sys.argv) != 2:  print("Usage: " + sys.argv[0] + " <path>")  sys.exit(1)  if os.path.isdir(sys.argv[1]):  print("This is a directory!")  elif os.path.isfile(sys.argv[1]):  print("This is a file!")  else:  print("This is not a directory or a file!")  if os.access(sys.argv[1], os.W\_OK):  print("It’s also readable!")  else:  print("You cannot read it!")  $ chmod +x file\_attr.py  $ ./file\_attr.py ./break.py  This is a file!  It's also readable! |
| **Security note:** It may seem like a good idea to use access() check permissions before opening files, but this actually creates a new security risk where the attacker may manipulate the file during the access() check. Instead, use a try/except checking for a PermissionError. | ~~if os.access("/some/file", os.R\_OK):~~  ~~f = open("/some/file")~~  try:  f = open("/some/file", "r")  except PermissionError:  return "You don't have permission to read this file" |
| **Commands and Processes**  There are two primary ways to run shell commands, each with their pros and cons.  os.system(*cmd\_string*)   * Takes a command, including arguments, as a string and executes it in a shell (bash). * Returns **only** the exit status.   subprocess.check\_output(*cmd\_string, shell=False*)   * Takes a string including an executable and any arguments, and executes as a subprocess. * Returns output to stdout as a string * If shell is set to True, cmd\_string will execute through a shell * **Waits** for the subprocess to complete. * Output defaults to the parent's (the script) output. * To capture stderr, pass stderr=subprocess.STDOUT.   **Note:** Before using check\_output() or system() check that there isn't already a function in the standard library that does what you want. | **Example 15:** running commands  $ vi command.py  #!/usr/bin/env python  import subprocess  output = subprocess.check\_output('ls -a', shell=True)  # get output as a string  print("---- STRING VERSION ----")  print(output)  # convert it to a list  print("\n---- LIST VERSION ----")  print(output.splitlines())  $ chmod +x command.py  $ ./command.py  ---- STRING VERSION ----  .  ..  break.py  command.py  datatypes.py  example1.py  except.py  file\_attr.py  flow\_for.py  flow\_while.py  func.py  infile.txt  input.py  input2.py  lists.py  logic.py  oddlines.py  oddlines2.py  ops.py  outfile.txt  vars.py  vowel.py  ---- LIST VERSION ----  ['.', '..', 'break.py', 'command.py', 'datatypes.py', 'example1.py', 'except.py', 'file\_attr.py', 'flow\_for.py', 'flow\_while.py', 'func.py', 'infile.txt', 'input.py', 'input2.py', 'lists.py', 'logic.py', 'oddlines.py', 'oddlines2.py', 'ops.py', 'outfile.txt', 'vars.py', 'vowel.py']  What module could we have used instead to get a list of files?  ?? |
| **Functions**  As with most languages, you can define functions in Python. Since it is dynamically typed (i.e., variables can change type during execution), function declaration is simple.  **def** *function\_name*(*parameters*):  *function\_suite*  Functions are then called by their function name.  **Optional Arguments**  To give a parameter a default value, use "=" assignment in the parameter declaration. When calling a function which takes optional parameters, you *must* specify the parameter name when calling it.  def *function\_name*(*parameter1,* ***parameter2=False***):  *function\_suite*  # call without optional param  function\_name("foo")  # call with optional param  function\_name("foo", paramter2=True) | **Example 16:** defining and calling functions  $ vi func.py  #!/usr/bin/env python3  # print a string one letter at a time, optionally backwards  def letter\_lines(string, backwards=False):  if backwards:  string = string[::-1]  for letter in string:  print(letter)  # parse yes or other to Boolean  def parse\_yn(decision):  if decision in ['y', 'Y', 'YES', 'yes']:  return True  return False  inword = input("Enter a word: ")  back = parse\_yn(input("Backwards? (y/n): "))  # call with and without optional parameter  letter\_lines(inword, backwards=back)  letter\_lines(" DONE")  $ chmod +x func.py  $ ./func.py  Enter a word: hello  Backwards? (y/n): y  o  l  l  e  h  D  O  N  E |
| **Exit Codes**  As with all programs, Python programs should return an exit code to their environment when they terminate (normally or abnormally). By **default**, Python will **return 0** when a program finishes or **non-zero** on an **unhandled exception**.  You can specify an exit code with the sys.exit(*exit\_code*) function. As usual, you should specify **0 on success** and **non-zero on failure**. | >>> import sys  >>> sys.exit(999)  $ echo $?  999 |

**©2018 Rocky Slavin,** UTSA CS students may make copies for their personal use