Unit 3 Module 1 Working with Python Modules

Section 1 Using Python Modules

- Importing Modules
  - Importing the whole library - `import math`
  - Importing the whole library and renaming it - `import math as ml`
  - Importing only the pow function - `from math import pow`

- Using Math Functions
  - Math Library: https://docs.python.org/3/library/math.html
  - Square root
    - Syntax: `math.sqrt(var/value)`

- Other arithmetic operators
    - Division operator / - Normal – Output has decimals
  - Integer division operator // - keep the integer and discard the fractional
    - Example: 5//2 = 2
  - Modulo operator % - keeps the remainder of a division.
    - Example: 5%2 = 1
  - Exponent operator **
    - Example: 2**3 = 8

- Operator Precedence
  - Parentheses, Exponents – Multiplication/Division/Integer Deivions, Modulo, Add/Subtract

- Rounding Numbers
  - Ceil Syntax: `math.ceil(var/value)`
  - Truncate Syntax: `math.trunc(var/value)`
  - Floor Syntax: `math.floor(var/value)`

- Generating Random Numbers
  - Random integer between a and b
      Syntax: `randint(start, stop)`
  - Random integer from a range
    - Syntax: `randrange(start, stop, step)`
  - Selecting an element from a list
    - randomly chosen element from a list passed as an argument.
    - Syntax: `choice(listName)`
  - Shuffling the elements of a list
    - The function shuffle shuffles the elements of list in place
    - Syntax: `shuffle(listName)`

Section 2 Working with Dates and Times

The datetime Module

- The time Object
  - Used to store and manipulate time information.
  - You can specify the hour, minute, second, and microsecond attributes
    - The default values of 0.
  - Valid ranges:
    - 0 ≤ hour < 24
    - 0 ≤ minute < 60
    - 0 ≤ second < 60
    - 0 ≤ microsecond < 1000000
  - If any of the attributes is outside its valid range, you will get a ValueError message.

  - Creating a time object
    - Example 1: **`t = time(20, 55, 20, 500)`**
    - Example 2: **`t = time(minute = 10, hour = 9, microsecond = 900000, second = 20)`**
    - Example 3: **`t = time(hour = 1, minute = 10)`**

  - Getting an attribute
    - You can access a single attribute or all attributes of a time variable separately.
    - Example: **`h = t.hour`**

  - Modifying attributes of an assigned time variable using **the .replace() function**
    - then reassign the new variable to the original variable t, which modifies t to reflect the desired changes.
    - Example: **`t = t.replace(hour = 8, minute = 8)`**
- Date Objects
  - date(year, month, day)
    - All of the date attributes are <u>required</u>
    - Valid ranges:
      - 1 ≤ year ≤ 9999
      - 1 ≤ month ≤ 12
      - 1 ≤ day ≤ number of days in the given month and year
  - Assigning a date object
    - Syntax: `dt = date(year, month, day)`
    - Example 1: **`date1 = date(2013, 5, 7)`**
      Example 2: **`date2 = date(day = 23, month = 4, year = 1999)`**

  - Getting a date attribute
    - The attributes of a date object can be accessed individually, in the same way you access the attributes of a time object.
    - Example: `y = SpecialDate.year`

- Modifying the attributes of an assigned date object using .replace()
  - The replace function can be used to modify the attributes of a date object in the same way it is used to modify attributes of an assigned time object.
    - Example: `dt = SomeDate.replace(year = 2016, day = 29)`

- Getting the current local date using .today()
  - Example: `d = date.today()`

- datetime Objects
  - datetime(year, month, day, hour = 0, minute = 0, second = 0, microsecond = 0)
  - The attributes have the same ranges as those of the individual time and date objects:
    - 1 ≤ year ≤ 9999
    - 1 ≤ month ≤ 12
    - 1 ≤ day ≤ number of days in the given month and year
    - 0 ≤ hour < 24
    - 0 ≤ minute < 60
    - 0 ≤ second < 60
    - 0 ≤ microsecond < 1000000

  - Assigning a datetime object
    - Example 1: `dt = datetime(2022, 7, 4, 16, 30)`
    - Example 2: `dt = datetime(day = 4, month = 7, year = 2022, minute = 30, hour = 16)`

  - Getting a datetime attribute
    - Example: `year = dt.year)`

  - Modifying the attributes of an assigned datetime object using .replace
    - Example: `dt = dt.replace(year = 2020, second = 30)`

  - Setting a datetime object to the current local date and time using .today()
    - Example: `dt = datetime.today()`

  - Splitting a datetime object into separate date and time objects
    - Example:

```
# get today's date and current local time
dt = datetime.today()
# split into time t and date d
t = dt.time()
print("Time is: ", t)
d = dt.date()
print("Date is: ", d)
```

  - Combining separate date and time objects into a single datetime object using .combine
    - Example: `dt = datetime.combine(date = d, time = t)`

- Formatting Dates and Times

- strftime() applies to time, date, and datetime objects. It reads the attributes of the object, applies a formatting directive, and returns a formatted string.
- The strftime() is passed a string containing all necessary formatting directives along with any necessary slashes, commas, colons, and so on.
- Date Formatting Directives

Directive	Meaning	Example
%a	Abbreviated weekday name	Sun, Mon, ..., Fri
%A	Full weekday name	Sunday, Monday, ..., Friday
%d	Day of the month as a zero-padded decimal	01, 02, 03, ... 31
%b	Abbreviated month name	Jan, Feb, ..., Dec
%B	Full month name	January, February, ..., December
%m	Month as a zero-padded decimal	01, 02,..., 12
%y	2 decimal year number (without century)	00, 01, ..., 99
%Y	4 decimal year number (with century)	1900, 1989, ..., 2015

- Time Formatting Directives

Directive	Meaning	Example
%H	Hour in 24-hour clock (zero-padded)	00, 01, ..., 23
%I	Hour in 12-hour clock (zero-padded)	00, 01, ..., 12
%p	AM or PM	AM, PM
%M	Minutes as zero-padded decimal	00, 01, ..., 59
%S	Seconds as zero-padded decimal	00, 01, ...,59

- The Python Documentation site has more information on the `strftime()` function at https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior.

- Formatting time objects

```
from datetime import time
t = time(hour = 10, minute = 15)

# display as 10:15 AM
# string passed to strftim includes all necessary spaces
and semicolons
formatted_string = t.strftime("%I:%M %p")
print("First format: ", formatted_string)

# display as 10:15:00 (24 hour clock, no AM/PM)
formatted_string = t.strftime("%H:%M:%S")
print("Second format: ", formatted_string)
```

- Formatting date objects

```
from datetime import date
d = date(year = 1999, month = 11, day =3)

# display as November, 03, 1999
# string passed to strftime includes all necessary spaces
and commas
formatted_string = d.strftime("%B, %d, %Y")
print("First format: ", formatted_string)

# display as Nov 03 99
formatted_string = d.strftime("%b %d %y")
print("Second format: ", formatted_string)
```

- Formatting datetime objects

```
from datetime import datetime
dt = datetime(year = 1999, month = 11, day = 3, hour =
10, minute = 15)

# display as November, 03, 1999 @ 10:15 AM
formatted_string = dt.strftime("%B, %d, %Y @ %I:%M %p")
print("First format: ", formatted_string)

# display as Nov 03 99 / 10:15:00
formatted_string = dt.strftime("%b %d %y / %H:%M:%S")
print("Second format: ", formatted_string)
```

Section 3: Date and Time Arithmetic

- timedelta Objects

- timedelta - represents a duration of time rather than a point in time.
- Syntax:

```
timedelta(weeks = 0, days = 0, hours = 0, minutes = 0,
seconds = 0, milliseconds = 0, microseconds = 0)
```

- Values default to 0 if not specified
- Explicit definition
  - You can define a timedelta object in the same way that you define a time, date, or datetime object.
  - Example:

```
from datetime import timedelta
delta1 = timedelta(days = 7,  hours = 2)
print(delta1)
Output: 7 days, 2:00:00
```

- Calculating the difference between two datetime objects
  - You can define a timedelta object as the result of subtracting a datetime or date object from another.
  - Example:

```
from datetime import datetime
dt1 = datetime(year = 2017, month = 1, day = 1)
dt2 = datetime(year = 2017, month = 2, day = 1)
delta2 = dt2 - dt1
print(delta2)
Output: 31 days, 0:00:00
```

- Getting the timedelta attributes
  - Note: only the days, seconds, and microseconds are stored in the object.
  - Example:

```
print(delta1.days)
Output: 7

print(delta1.seconds)
Output: 7200
```

    - It's apparent that Python has converted the 2 hours into 2 × 60 × 60 = 7200 seconds.
- Total number of seconds
  - Python can also compute the total number of seconds in a timedelta object using the total_seconds() function.
  - Example:

```
print(delta1.total_seconds())
Output: 612000.0
```

- Performing datetime Arithmetic
  - You can perform basic arithmetic operations on timedelta objects.
  - The following table lists some of the available operations; a complete listing is available on the Python Documentation site at https://docs.python.org/3/library/datetime.html#timedelta-objects.

-

Operation	Result
t1 = t2 + t3	sum of objects
t1 = t2 - t3	different between objects
t1 = t2 * i	multiply by an integer i
f = t2 / t3	division, return float
t1 = t2 // i	integer division, return int
t1 = t2 % t3	modulo or remainder

- Comparing datetime Objects
  - You can compare datetime objects to find out which date precedes the other. The result of the comparison can be used in a conditional or loop structure to control the flow of your program. You can use any of the operators in the following table to perform a comparison.

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to

Section 4: File System

- Platform Identification
    - Python scripts can run on different platforms, including Windows, Mac, and Linux. The sys module provides access to several system variables, including the platform. If you know the platform your Python code is running on, you may be able to change the behavior of your application to accommodate that platform.
    - Example

```
import sys
print(sys.platform)
output: linux
```

- Directory Operations Summary

Method	**Purpose**
`os.getcwd()`	Returns a string containing the current working directory path.
`os.chdir(path)`	Changes the working directory where path is a string
`..`	Effectively changes the working directory one level up into the parent directory when used in os.chdir() as the path
os.listdir()	returns a list of the files and directories in the current working directory
os.mkdir(path)	Create a new directory manually using where path is the path and name of the new directory
os.rmdir(path)	Remove a directory where path is the name of the directory to be deleted
os.rename(src, dst)	Rename a large number of files to match a predefined pattern where src is a string containing the path of a source file or directory, and dst is a string containing the new (destination) path and/or name.
os.path.exists(path)	Tests whether path (relative or absolute) exists in the file system
os.path.isfile(path)	returns True if path (relative or absolute) refers to an existing file
os.path.isdir(path)	returns True if path refers to an existing directory

- Directory Structure

```
parent_dir
|
|__  child1_dir
|    |
|    |_ leaf1.txt
|
|__ child2_dir
|    |
|    |_ leaf2.txt
|
|__ parent_leaf.txt
|
|__ text_file.txt
```

  - Directories
    - The directories in this structure are parent_dir, child1_dir, and child2_dir.
  - Files
    - The files in this structure are leaf1.txt, leaf2.txt, parent_leaf.txt, and text_file.txt.

- Path Operations
  - Relative and absolute paths
    - **Relative paths**: path to a file or directory from a specific location (or current working directory)
    - **Absolute paths**: path to a file or directory from a root. In UNIX flavors, a root is "/"; whereas, on Windows machines a root is "C:\"
    - Example:

```
import os, os.path

# Print the current working directory (should be
"parent_dir")
print('The current working directory is:',
os.getcwd())

# Find the absolute path to child1_dir/leaf1.txt
abs_path = os.path.abspath('child1_dir/leaf1.txt')
print("Absolute path to leaf1.txt is: ", abs_path)

# Test whether the path exists
if(os.path.exists(abs_path)):
    print("Path exists")

    # Test to see if it's a file or directory
    if(os.path.isfile(abs_path)):
       print("It's a file")
```

```
    elif (os.path.isdir(abs_path)):
        print("It's a dir")

else:
    print("Path doesn't exist")
```

- Testing the existence of paths, files, and directories

  - The module os.path contains common pathname manipulation functions.
  - The Python Documentation site at https://docs.python.org/3/library/os.path.html has more information on the os.path module.
  - Example:

```
import os.path

name = input("Enter a file or directory name: ")

if (os.path.exists(name)):
    print("The file or directory exists in the
current working directory")
else:
    print("The file or directory does NOT exist in
the current working directory")
```

Unit 3 Module 2 More Powerful Statements

Section 1 Boolean Expressions and Compound Conditionals

- bool data type
  - Adding two numbers in Python results in another number; however, if two numbers were compared (i.e. to test for equality), the result would be of Boolean (bool) type. Unlike int and float, which can take many values, bool has only two values True and False.
  - Examples:
    5 == 6 → False
    5 <= 6 → True

Operator	Description
<	Less than
<=	Less than or equal to

>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

- Comparing string variables or numerical variables is performed using relational operators, the following table lists the relational operators supported in Python

- Fundamental bool operators
  - The three basic Boolean operators are: not, and, or.
  - not
    - Is a unary operator; it operates on one variable (operand) by negating it.
    - The following truth table shows the result of negating a bool variable B.

B	not B
False	True
True	False
  - and
    - Is a binary operator; it operates on two variables (operands).
    - It produces True when both variables are True and produces False otherwise.
    - The following truth table shows the effect of the and operator on combining A and B.

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

- or
    - Is a binary operator; it operates on two variables (operands).
      It produces True if either (or both) of the variables is True and produces False otherwise.
      The following truth table shows the effect of or on combining A and B.

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

    - These operators are typically used to combine relational expressions rather than the constants True and False.
    - NOTE: not has the highest precedence, followed by and, then or.
    - Similar to arithmetic operators, you can control the precedence using the parentheses operator () to group relational expressions.

- Combining Comparison
    - The basic bool operators can be used to combine multiple relational tests.
    - For example, say you want to test whether a variable x contains a number within a certain range (i.e. $10 \leq x \leq 20$).
        - You can perform this test, by checking whether the number is greater than or equal to 10 and whether the number is less than or equal to 20.
    - Examples
      x = 11
      (x >= 10) and (x <= 20)
      Result: True

      x = 9
      (x >= 10) and (x <= 20)
      Result: False

    - The relational tests need not be numerical. For example, you can test whether a variable c contains a capital letter (i.e. $'A' \leq c \leq 'Z'$).
    - Examples:
      c = 'N'
      (c >= 'A') and (c <= 'Z')
      Result: True

      c = 'n'
      (c >= 'A') and (c <= 'Z')
      Result: False

- Boolean Expressions Equality
    - It is possible to write two (or more) different Boolean expressions that yield the same results.
    - Example:
      x = 11
      (x >= 10) and (x <= 20) has the same results as not((x < 10) or (x > 20))

- Compound Conditionals
    - Combined Boolean expressions can be used within the testing conditions of if and elif statements.
    - This will let you write compound conditionals and give you fine control over the program flow.
        - For example, you can test the validity of user input against several combined conditions.
        - Similarly, you can also check the content of a variable against multiple ranges.
    - Examples:

```
# Test number for validity
if ((x > 0) and (x % 2 != 0)):
   print(x, "is a valid number")
else:
   print(x, "is NOT a valid number")

#Test for inclusion in a range
if (y < 1970):
   print("You were born before 1970!")
elif (y >= 1970 and y < 1980):
   print("You were born in the 70s!")
elif (y >= 1980 and y < 1990):
   print("You were born in the 80s!")
elif (y >= 1990 and y < 2000):
   print("You were born in the 90s!")
elif (y >= 2000 and y < 2010):
   print("You were born in early 2000s!")
else:
   print("You were born in the current decade!")
```

Section 2 Advanced Loop Structures

- pass Statements
    - The pass statement does nothing; however, it still has useful applications.
    - Used as a placeholder, indicating that it will be replaced by valid functioning code.
    - Used to avoid interpreter errors.
- Changing Loop Iterations
    - break statements
        - Use to stop a loop and then resume executing the code that follows.
        - Called breaking a loop and can be achieved in Python using a break statement
        - Can be used with while or for loops and it breaks the innermost loop of nested loops.
        - Use to improve the efficiency of your code.
            - For example, assume you are looping over the elements of an unsorted list looking for the first occurrence of a 0. If a 0 is encountered earlier in the search, there is no need to continue the search and a loop break

would be very desirable. break statements can also be used to break out of infinite loops.

- continue statements
  - Use when you want to skip an iteration and jump to the next one.
  - Can be used with while or for loops, and it applies to the innermost loop of nested loops.
  - Used to improve the efficiency of code by skipping unnecessary iterations.
- Breaking infinite loops
  - Infinite loops have some applications, and a break statement would be necessary to break such loops and continue executing the rest of a program.
    - For example, infinite loops are useful for prompting the user for specific input, and should be broken when valid input is entered.

- Nested Loops
  - You can use a loop inside another loop, this concept is known as nested loops.
  - Used to iterate over the elements of a table.
  - A table can be represented in Python as a 2 dimensional list, which is a list containing other lists.
  - You will need two loops to iterate over all the elements of a table, the outer loop iterates over the rows of the table, while the inner loop iterates over the columns in each row.
  - Example:

```
# list of lists
table = [[5, 2, 6], [4, 6, 0], [9, 1, 8], [7, 3, 8]]
for row in table:
    for col in row:
        # print the value col followed by a tab
        print(col, end = "\t")

    # Print a new line
    print()
```

    Output:

```
5       2       6
4       6       0
9       1       8
7       3       8
```

- Loops Containing Compound Conditional Expressions
  - Compound conditional expressions utilizing Boolean logic can be nested within loops.
  - Example:

```
lst = [102, 34, 55, 166, 20, 67, 305]
# Nesting a compound conditional in a for loop
largest = 0
for num in lst:
    # test if num is even and greater than largest
```

```
    if((largest < num) and (num % 2 == 0)):
        largest = num

print("Largest even number in the list is: ", largest)
```

Section 3 Containment, Identity, and Operator Precedence

- Containment (in, not in)
    - Need to test whether an element is contained in another container (i.e. list, tuple, string, etc.).
    - Example: Test whether 5 is an element of the list [4, 8, 5, 6
    - It is possible to perform these tests by iterating over the elements of the container (list) and comparing them one by one to the element of interest. However, because this procedure is commonly used, Python has a containment operator (in), which can perform this procedure much more efficiently.
    - NOTE: You can test if an element is NOT contained in another container by using the (not in) operator.
    - Example:

```
lst_container = [4, 8, 5, 6]
x = 5
if (x in lst_container):
    print(x, "is contained in list")
else:
    print(x, "is NOT contained in list")
```

    - You can test if a substring is contained in another larger string.
    - Example

```
sentence = "This is a test sentence"
word1 = "test"
word2 = "something"

# testing if word1 is a substring of sentence
if (word1 in sentence):
    print(word1, "is contained in:", sentence)
else:
    print(word1, "is not contained in:", sentence)
```

- Identity (is, is not)
    - Python is an object-oriented programming language that utilizes objects.
    - You have been using objects; however, you have called them variables and lists.
    - Python saves objects in certain memory locations, knowing the locations is of little importance; however, knowing if two seemingly different objects are at the same memory location is important.
    - This will be critically important when dealing with sequences such as list, tuples, strings, and dictionaries.

- In Python, the concepts of identity and equality are related but not the same, for example:
  - When two objects are saved in the same memory location, they are equal and identical
  - When two objects are saved in different memory locations and contain the same information, they are equal but not identical
  - When two objects are saved in different memory locations and contain different information, they are not equal nor identical
  - You can test whether two objects contain the same information using the equality operators == or !=. To test whether two objects are identical, you need to use the identity operators is or is not.

- Identity of variables containing numerical literals
  - int literals
    - Equal int literals are saved in the same memory location
  - Example

```
# x, y: equal, identical
x = 5
y = 5
print("x equal y ? ", x == y)
print("x is identical to y ?", x is y)

# x, y: not equal, not identical
x = 5
y = 6
print("x equal y ? ", x == y)
print("x is identical to y ?", x is y)
```

  - float literals
    - Equal float literals are not identical. In other words, equal float literals are saved in different memory locations.

- Identity of variables containing lists
  - Equal but not identical lists
    - You can create two equal but not identical lists, by assigning the same list literal to two different variables.
    - A change in one of the list does not have any effect on the content of the other.
  - Example:

```
x = [4, 9, 8]
y = [4, 9, 8]
# x and y are equal, because they contain the same data
# x and y are NOT identical, because they are saved in different memory locations
```

        - Equal and identical lists
        - Two variables referring to the same list are called identical variables.
        - They can be treated as two names for the same list
        - Both of the variables refer to the same memory location and a change in one is reflected as the same change in the other
        - Example
          # Identical list
          x = [4, 9, 8]
          y = x
          # x and y are equal, because they contain the same data
          # x and y are identical, because they are saved in the same memory location

    - Identity of variables containing string literals
        - String identity and equality is very similar to that of lists.
        - However, when you assign equal strings to different variables, the interpreter might detect this and optimize the code by making the variables identical.
        - Example:
          # s1, s2: equal, not identical
          s1 = 'whole milk'
          s2 = 'whole milk'

- Operator Precedence
    - The following table summarizes operator precedence from lowest precedence to highest precedence. Operators in the same row have the same precedence, and the precedence is resolved from left to right, for example in (3 * 6 / 9), * and / have the same precedence, and Python will perform the multiplication first (18 / 9) followed by the division (2).

Operator	**Short Description**
or	Boolean or
and	Boolean and
not	Boolean not
in, not in, is , is not, <, <=, >, >=, !=, ==	Containment and identity test, relational comparison
+, -	Arithmetic addition and subtraction
*, /, //, %	Arithmetic, multiplication, division, int division, modulo
**	Exponentiation
(), [], {}	Parentheses, brackets

    - NOTE: When in doubt, use the parentheses operator to control the precedence in an expression

Section 4 Formatting

- Using Old-Style Formatting
  - There are many ways to display output in Python.
  - The easiest way is to use the print function and pass the items you want to display as comma-separated arguments.
  - Example:
    g = 4 #gallons of gas
    m = 127.3 #miles
    print('The fuel efficiency of a car consuming', g ,'gallons of gas for every', m, ' miles =', m/g, 'MPG')
    Output: The fuel efficiency of a car consuming 4 gallons of gas for every 127.3 miles = 31.825 MPG
  - No control over the precision or alignment of the printed numbers.
  - You can use the old-style formatting that is common among other programming languages such as C.
  - Use positional format placeholders that will be replaced by the values you want to display. You can rewrite the preceding program as follows:
  - Example:
    g = 4 #gallons of gas
    m = 127.3 #miles
    print('The fuel efficiency of a car consuming **%d** gallons of gas for every **%4.2f** miles = **%4.2f** MPG' **% (g, m, m/g)**)
  - Output:
    The fuel efficiency of a car consuming 4 gallons of gas for every 127.30 miles = 31.82 MPG
  - Placeholders in the string; each starts with %.
  - The string is followed by the modulo operator % and a **tuple** containing the values that should replace the placeholders in order.
    - A Tuple is a collection of Python objects separated by commas.
  - The following table shows these placeholders and the values that replaced them.

Format placeholder	**Replaced by**
%d	g
%4.2f	m
%4.2f	m/g

  - Each placeholder has the following pattern:
    %[flags][width][.precision]type
  - Example: the number of miles m is formatted as %4.2f which means:
    - No flags are used
    - The total number of characters (<u>including the .)</u> that should be displayed is (width = 4)
    - The number of decimal digits is (precision = 2)

    - The numerical value type is float (f)
- If you use the format placeholder %08.3f instead, you will increase the number of char displayed to 8, padded with zeros on the left, and the precision to 3 decimal points.
- Example:

```
g = 4 #gallons of gas
m = 127.3 #miles
print('The fuel efficiency of a car consuming %d gallons of gas for every %08.3f miles = %4.2f MPG' % (g, m, m/g))
Output:
The fuel efficiency of a car consuming 4 gallons of gas for every 0127.300 miles = 31.82 MPG
```

- The following tables show commonly used flags and types. The Python Documentation site at https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting has a complete listing.

Flag	**Meaning**
0	The conversion will be padded by zeros for numeric values
-	The converted value is left adjusted
' '	A blank will be placed in front of positive numbers, - will be placed in front of negative numbers
+	A sign (+ or -) will be placed in front of the converted number

Type	**Meaning**
d	Signed integer
i	Signed integer
e	Floating point exponential format (lowercase)
E	Floating point exponential format (uppercase)
f	Floating point decimal format
F	Floating point decimal format
c	Single character (accepts integer or single character string)
s	String
%	No argument is converted, results in a % character in the result

- Example of formatting strings and characters

```
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")
print('The first name starts with: %c' % (first_name[0]))
print('The last name ends with the 2 chars: %s' %(last_name[-2:]))
Output:
Enter your first name: John
Enter your last name: Smith
The first name starts with: J
The last name ends with the 2 chars: th
```

- Alignment Example

```
from math import pi
```

print("Right adjusted: **%20.2f**" **%(pi)**)
print("Left adjusted: %-20.2f" %(pi))
Output:
Right adjusted: 3.14
Left adjusted: 3.14

- Using Python Style Formatting
  - Python comes with a much more flexible string-formatting method: str.format().
    - More control over how to format a print output.
    - The general form of this style is:
      string_sequence.format(p0, p1, ..., k0=v0, k1=v1, ...)
    - The string_sequence variable contains format placeholders and the string you are trying to display. The format placeholders are similar to those used in the old-style formatting; however, you do not need to use the % operator in the Python formatting style.
    - Examples:
      - Without position index, Without formatting string
        - Use {} as a placeholder within the string wherever you want a variable value to appear. Then pass the variables you want to display as positional arguments to the .format() method; the variables must be passed in order.

-

g = 4 #gallons of gas
m = 127.3 #miles
print('The fuel efficiency of a car consuming **{}** gallons of gas for every **{}** miles = **{}** MPG'**.format(g, m, m/g)**)
Output:
The fuel efficiency of a car consuming 4 gallons of gas for every 127.3 miles = 31.825 MPG

- Without position index, With formatting string
  - Use {:4.2f} as a placeholder, then pass the variables you want to display as positional arguments to the .format() method.
  - Note that the placeholder starts with :, followed by a formatting string similar to that of an old-style formatter.
  - This method is the closest to the old-style formatting style.

g = 4 #gallons of gas
m = 127.3 #miles
print('The fuel efficiency of a car consuming **{:d}** gallons of gas for every **{:4.2f}** miles **= {:4.2f}** MPG'**.format(g, m, m/g)**)
Output:
The fuel efficiency of a car consuming 4 gallons of gas for every 127.30 miles = 31.82 MPG

- With position index, With formatting string
  - Use {0:4.2f} as a placeholder, then pass the variables you want to display as positional arguments to the .format() method.
  - Note that the placeholder starts with an index, followed by :, followed by a formatting string similar to that of an old-style formatter.
  - Example:

    g = 4 #gallons of gas
    m = 127.3 #miles
    print('The fuel efficiency of a car consuming **{0:d}** gallons of gas for every **{1:4.2f}** miles = **{2:4.2f}** MPG'**.format(g, m, m/g)**)
  - Output: The fuel efficiency of a car consuming 4 gallons of gas for every 127.30 miles = 31.82 MPG
  - Note: The indices 0, 1, 2 refer to the order of the variables passed to the .format() method in order
  - Example:

    g = 4 #gallons of gas
    m = 127.3 #miles
    print('The fuel efficiency of a car consuming {1:d} gallons of gas for every {2:4.2f} miles = {0:4.2f} MPG'.format(m/g, g, m))
    Output:
    The fuel efficiency of a car consuming 4 gallons of gas for every 127.30 miles = 31.82 MPG
- Using keywords
  - Use a keyword instead of an index in the placeholder {a:4.2f}, then pass the variables you want to display as key/value pairs to the .format() method.
  - Example:

    g = 4 #gallons of gas
    m = 127.3 #miles
    print('The fuel efficiency of a car consuming **{a:d}** gallons of gas for every **{b:4.2f}** miles = **{c:4.2f}** MPG'**.format(a = g, b = m, c = m/g)**)
    Output:
    The fuel efficiency of a car consuming 4 gallons of gas for every 127.30 miles = 31.82 MPG
- Using flags
  - The formatter string in the placeholder follows a similar syntax to that of the old-style formatter.
  - Syntax:

    :[flags][width][.precision]type
  - The following table shows commonly used flags for the Python display style

Flag	**Meaning**
<	Left-align within available space

>	Right-align within available space
^	Center-align within available space
0	The conversion will be padded by zeros for numeric values
,	Use a comma as thousands separator
' '	A blank will be placed in front of positive numbers, - will be placed in front of negative numbers
+	A sign (+ or -) will be placed in front of the converted number

Unit 3 Module 3 Methods & Structures for Robust Code

Section 1 Error Handling

Exception Types

- Exception = error
  - Example: ZeroDivisionError: division by zero
    - Exception indicating that it was raised by a division by zero (#/0)
- IndexError
  - Raised by trying to access (or write into) an out-of-range index of a sequence (i.e. list).
- NameError
  - Raised by accessing an unspecified (undefined) variable.
- TypeError
  - Raised when performing an unsupported operation on a specific type, such us dividing strings.
- More information about the built-in exception types is available from the Python Documentation site at https://docs.python.org/3/library/exceptions.html#bltin-exceptions.

Handling Errors

- You can modify your program to handle specific exceptions and to continue execution without stopping.
- Handling Exceptions:
  - The process is called **exception handling**, and can be done using a `try...except` statement
  - Syntax:

```
try:
    code that may raise an exception
except ExceptionType:
    code block to handle exception
```

- Handling Multiple Exceptions
  - Your program can raise **exceptions of different types**. You can use the `try...except` statement to handle each of the types differently by adding another `except` clause.
  - Syntax:

```
try:
    code that may raise an exception
```

```
except ExceptionType_1:
    code block to handle exception of type ExceptionType_1
except ExceptionType_2:
    code block to handle exception of type ExceptionType_2
```

- Handling Unexpected Exceptions
  - If your code encounters an exception you did not anticipate or explicitly handle, your program will stop execution and display an error message. You can handle **unexpected exceptions** using an `**except**` clause without specifying the exception type
  - Syntax:

```
try:
    code that may raise an exception
except ExceptionType_1:
    code block to handle exception of type
    ExceptionType_1
except ExceptionType_2:
    code block to handle exception of type
    ExceptionType_2
except:
   code block to handle unexpected exceptions of any type
```

- Handling exceptions raised by other functions
  - When a function raises an exception without handling it, the caller function must handle the exception or raise it up another level.
  - Example: the `sqrt` function from the `math` module expect a positive number for an argument. Passing a negative number to `sqrt` will raise a `ValueError` exception. The `sqrt` function doesn't handle the exception internally, so your code should handle it or raise it to an upper level.
- Error messages
  - The error message associated with a raised exception can be stored in a variable using (`as exception_object:`) in the `except` clause, right after the exception type:
  - Syntax:

```
try:
    code that may raise an exception
except ExceptionType_1 as exception_object:
    code block to handle exception of type
    ExceptionType_1
except ExceptionType_2 as exception_object:
    code block to handle exception of type
    ExceptionType_2
except Exception as exception_object:
    code block to handle unexpected exceptions of any
    type
```

  - NOTE: For unexpected exceptions, use (`Exception as exception_object:`) to store the error message.

Else and Finally

- else
  - Run some code when a try statement does not raise an exception.
    - Use optional else statement.
    - The code block within the else statement will be executed when the code reaches the end of the try code block without raising any exceptions.
  - Syntax:

```
try:
    code that may raise an exception
except ExceptionType_1:
    code block to handle exception of type
    ExceptionType_1
except ExceptionType_2:
    code block to handle exception of type
    ExceptionType_2
except:
    code block to handle unexpected exceptions of any
    type
else:
    code block to run when no exceptions were raised
```

- Finally
  - To execute some code whether or not an exception has been raised.
  - The optional **finally** clause can handle this situation and run a code block whenever you exit a try...except statement.
    - The code block within the finally statement will be executed in the following situations:
      - when an exception has been raised,
      - upon reaching the end of the try statement without raising an exception, and
      - when exiting a try statement by using break, continue, or return.
  - finally is typically used for **clean-up actions**, i.e. releasing external resources such as files or network connections.
  - Syntax:

```
try:
  code that may raise an exception
except ExceptionType_1:
  code block to handle exception of type
   ExceptionType_1
except ExceptionType_2:
  code block to handle exception of type ExceptionType_2
except:
  code block to handle unexpected exceptions of any type
finally:
  code that will always execute whether an exception was
   raised or not
```

- **NOTE**: Though Python allows you to use an else clause followed by a finally clause, the code generated will be difficult to understand and you should avoid using these two clauses together.

Raising Exceptions

- To define and raise a **forced exception** using the raise statement
- Syntax:

```
raise ExceptionType("Error Message")
```

- The ExceptionType can be one of the built-in exceptions supported by Python, or you can define a new type.
- The Error Message will be associated with this exception when it is raised.
- Example: You can raise an exception when a user does not provide a valid response to a request for input between 1 and 10. However, numbers outside the range do not necessarily generate errors in Python, so you have to **explicitly force a raised exception**.

Section 2: Files

Deleting Files

- In module 1 you explored the os module and used some of its methods to interact with the file system.
  - Changed the working directory,
  - listed the content of a path,
  - created new directories,
  - removed directories, and
  - renamed files and directories.
- In addition to these utilities, Python's os module allows you to **remove specific files using the os.remove(path) or os.unlink(path) functions**.
  - Both functions are semantically identical; however, their functionality slightly differs depending on the platform running your program.
  - For now, we will consider them equivalent and **use os.remove(path) to delete a file**.

Checking File Existence

- Removing a file that doesn't exist
  - Raises a FileNotFoundError exception.
  - Syntax:

```
# Removing a file that does not exist
file_path = "parent_dir/fictitious_file.txt"
os.remove(file_path)
```

  - Exception:

```
FileNotFoundError: [Errno 2] No such file or directory:
'parent_dir/fictitious_file.txt'
```

- Removing a directory using **os.remove**
  - When a directory is passed as an argument for os.remove a PermissionError is raised.
  - Syntax

```
# Passing a directory path to os.remove
dir_path = "parent_dir"
os.remove(dir_path)
```

- Exception:

```
PermissionError: [Errno 1] Operation not permitted:
'parent_dir'
```

Handling File Exceptions

- Example of handling unexpected file exceptions.

```
import os.path
file_path = "parent_dir/fictitious_file.txt"
# Remove a file
try:
    os.remove(file_path)
except FileNotFoundError as exception_object:
    print("Cannot find file: ", exception_object)
except PermissionError as exception_object:
    print("Cannot delete a directory: ", exception_object)
except Exception as exception_object:
    print("Unexpected exception: ", exception_object)
```

with Statements

- It is very important to close the file to ensure that all output is written properly and the resources are freed.
- Sometimes an exception is raised before reaching the close() statement, and the file is kept open.
  - This issue can be **resolved by placing the close() statement inside a finally clause**.
  - However, because the process of opening and closing a file is very common, Python provides a succinct with statement that performs the same task.
  - Syntax:

```
with open(FILE_NAME, MODE) as VARIABLE:
    code block
```

Section 3: Tuples

Tuple Basics

- A **list** is a very useful data structure that can manipulate a sequence of elements.
- A **tuple** is another data structure type that can also **store** and **manipulate** a **sequence** of elements.
  - Like lists, the elements of a tuple can be of **any type** (i.e. int, float, string, list, another tuple).
  - However, unlike lists, a tuple is an **immutable data structure**, which means that after you define a tuple, you are not allowed to change it.
    - In other words, you cannot reorder the elements of a tuple, cannot append additional elements, and cannot delete elements.

- The importance of tuples is most obvious when **passing to and returning from a function**.
- They are also important to communicate to people reading your code that you do **not intend for the content of the tuple to be mutated (changed).**

Creating a tuple

- You can create a tuple by enclosing its elements within parentheses.

- Syntax:

```
T = (val, val, val…)
```

- Example:

```
T = (13, 5, 92)
```

Creating an empty tuple

- You can create an empty tuple using empty parentheses.
- Example:

```
T = ()
```

Creating a tuple with a single element

- When creating a tuple with a single element, you have to add a comma , after the single element. Otherwise, Python will assume you are using the parentheses as part of an expression rather than to create a tuple.
- Example:

```
T2 = (5,) # note the comma after 5
```

Accessing elements of a tuple

- You can access the individual elements of a tuple in the same way you access the elements of a list or a string, by using the index of the element of interest.
- Example:

```
T = (13, 5, 92)
T[0]
Output: 13
T[-1]
Output: 92
```

Converting a list into a tuple

- You can convert a list into a tuple by passing it into the **tuple function**.
- Example:

```
L = [4, 5.3, 'name']     #list
T = tuple(L)             #tuple
```

Converting a tuple into a list

- You can convert a tuple into a list by passing it into the list function.
- Example:

```
T = ('name', [2, 4], 5.3, 19)       #tuple
```

```
L = list(T)                          #list
print(L)
Output: ['name', [2, 4], 5.3, 19]
```

Changing a tuple element

- Tuples are immutable objects and cannot be changed. If you try to change the content of a tuple, a TypeError exception will be raised.
- Example:

```
T = (13, 5, 92)
T[0] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- A list can be one of the elements of a tuple
  - the tuple doesn't store the whole list, but stores a reference to the storage location of the list in memory.
  - Therefore, if the memory location storing the list is changed, the tuple references the changed list.

- In the following example, the second element in the tuple T is a list [2, 4].
  - T[1] stores the location of the list rather than the list itself; therefore, if you change the content of the list, (i.e. change 2 to a 5), the tuple T now references the new list [5, 4].
  - Technically speaking, the tuple did not change; it still references the same memory location. However, the memory location changed. This concept will be more relevant when you use tuples with functions or copy tuples.
- Example:

```
T = ('name', [2, 4], 5.3, 19)
T[1][0] = 5
T
```

Output: ('name', [5, 4], 5.3, 19)

Deleting a tuple

- Like any other variable, a tuple can be deleted by using the **del() function**.
- Example:

```
del(T)
type(T)
Output:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'T' is not defined
```

Slicing & Unpacking Tuples

- Slicing a tuple (or a list) lets you access a subset of a tuple.
- You can then assign this subset as a new tuple, pass it to a function, or do anything you would do with the full tuple.

- Syntax:
  `T[initial_index:final_index]:`
- Note:
  - The initial_index is optional; if left empty, slicing starts from 0
  - The final_index is optional; if left empty, slicing goes to the last element of the tuple
  - The T[initial_index] element is included in the sliced tuple; the T[final_index] is excluded
  - You can use negative numbers for both indices, where the final tuple element has index -1, the preceding has index -2, and so on
  - Whether you use negative or positive indices you should always keep initial_index < final_index

Unpacking tuples

- A tuple can be unpacked and its values can be assigned to multiple variables at the same time.
- Useful when a function returns a single tuple containing all the returned variables.
- Example:

```
>>> (a, b) = (4, 5)
>>> print(a)
4
>>> print(b)
5
>>>
```

The parentheses can be left off when unpacking tuples.

- Example:

```
>>> x, y, z = 1, 2, 3
>>> x
1
>>> y
2
>>> z
3
```

Returning function values

- Tuples can be used in functions to return multiple values.

Tuple Operations and Functions

- Python support several tuple operations, such as containment, identity, and concatenation.
- Containment
  - You can test whether an object is contained in a tuple using `in, not in`
- Identity (is) & Equality (==)
  - You can test whether two tuples contain equal elements by using the identity `==` operator.
  - Similarly, you can test whether tuples are identical by using the `is` operator.
  - Remember, if two tuples (and lists) are identical, changing an element in one will generate the same change in the other.
- Concatenation (+)

    - You can concatenate two tuples and save the result in a new tuple.
- Length of a tuple (len)
    - The length of a tuple is the number of elements in it.
    - If a list (or another tuple) is an element of the tuple, it is counted as 1 regardless of how many subelements it actually contains.
    - Knowing the length of a tuple might be helpful when iterating over the elements of the tuple.

Section 4: Dictionaries

- Another data structure to store a collection of data.
- It does **not store a sequence of data but rather an association of (key, value) pairs**.
- All **keys** in a dictionary are **unique**, while a value can be associated with multiple keys.
- A dictionary **can be searched**.
    - If you are searching for a specific element in an unordered list (or tuple) you will have to traverse the whole list until you find the element; on the other hand, when you search a Python dictionary, there is no need to traverse all of its elements.
    - A Python dictionary uses a mathematical process called **hashing** that optimizes the search capability by using the keys of a dictionary.

Creating a dictionary

- A dictionary can be created by using the braces operator **{key:value}**.
- Can have to specify the keys and their associated values.
- `int keys`
    - Keys can be integer values starting from 0, much like lists and tuples.
    - Example:

```
D = {0:5.5, 1:10.3, 2:43.2, 3:85.3}
type(D)
Output: <class 'dict'>
```

    - Keys do not have to start from 0, neither do they have to be in order.
    - Example:

```
D = {1343:5.5, 2:10.3, 3234235234:43.2, -324:85.3}
type(D)
Output: <class 'dict'>
```

- float keys
    - Keys can be float.
    - Example:

```
D = {1.5:5.5, 2.9:10.3, -3.8:43.2, 5:85.3}
type(D)
Output: <class 'dict'>
```

- str keys
    - As you might have noticed by now, keys can be of different types, and keys within the same dictionary can be of different types.

- **A key should be of a hashable type**; therefore, you cannot use mutable objects such as lists or other dictionaries.
- Example:

```
D = {'Name':'Skye', 'Age':35, 'Temperature':98.7}
```

- NOTE: An empty dictionary can be created as D = {}.

Accessing dictionary items

- You can **access** the value associated with a key of a dictionary **by using the subscript [] operator**, in much the same way you did with lists and tuples.
- Example:

```
D = {'Name':'Skye', 'Age':35, 'Temperature':98.7}
print("The name is: {}".format(D['Name']))
Output: The name is: Skye
print("Age is:", D['Age'])
Output: Age is: 35
```

Modifying dictionary items

- **Dictionaries are mutable objects and can be modified**.
- For example, you can change the value associated with a key by accessing it using the [] operator amd then assigning it a new value.
- Example:

```
D = {'Name':'Skye', 'Age':35, 'Temperature':98.7}
D['Name'] = 'Tamara'
D
Output: {'Name': 'Tamara', 'Age': 35, 'Temperature': 98.7}
```

Adding dictionary items

- You can **add a new key:value pair** to a dictionary by using the **operator [].** Just make sure the key doesn't already exist in the dictionary. (You will see later that you can use the containment operator in to test a key's existence.)
- Output:

```
D = {'Name':'Skye', 'Age':35, 'Temperature':98.7}
D['Last Name'] = 'Babic'
D
Output: {'Name': 'Skye', 'Age': 35, 'Temperature': 98.7, 'Last
Name': 'Babic'}
```

Deleting dictionary items

- You can **delete a dictionary key:value pair** using the **pop(key)** method, the key:value pair will be deleted and the method will return the value associated with the key.
- Example:

```
D = {'Name':'Skye', 'Age':35, 'Temperature':98.7, 'Last Name':
'Babic'}
a = D.pop('Age') # remove the key:value ('Age':35) pair
a # D['Age']
Output: 35
D
```

```
   Output: {'Name': 'Skye', 'Temperature': 98.7, 'Last Name':
'Babic'}
```

- If you try to delete a key:value pair not in the dictionary, a KeyError exception will be raised.
- Example:

```
   D = {'Name':'Skye', 'Age':35, 'Temperature':98.7, 'Last Name':
'Babic'}
   D.pop('Middle Name') # There is no middle name in D
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   KeyError: 'Middle Name'
```

Clearing and deleting a dictionary

- You can delete all the elements in a dictionary by using the .**clear()** method.
- Example:

```
   D = {'Name':'Skye', 'Age':35, 'Temperature':98.7, 'Last Name':
'Babic'}
   D
   Output: {}
```

- You can delete the whole dictionary by using **del** like you would delete any other variable.
- Example:

```
   D = {'Name':'Skye', 'Age':35, 'Temperature':98.7, 'Last Name':
'Babic'}
   del(D)
   D
   Output: Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   NameError: name 'D' is not defined
```

Looping Over Dictionary Items

- You can traverse the elements of a dictionary in much the same way you loop over the elements of a list or a tuple; however, with dictionaries you can use the keys, the values, or both keys and values when iterating over the dictionary elements.

Dictionary Iterations

Iterating over the keys

- You **can iterate over all the keys** of a dictionary using the **.keys()** method in a for loop.
- Example:

```
D = {'Name':'Skye', 'Age':35, 'Temperature':98.7, 'Last Name':
'Babic'}
# Iterate over the keys of D
for key in D.keys():
    print("D[{}] = '{}'".format(key, D[key]))
```

Iterating over the sorted keys

- Python stores the _key:value_ pairs of a dictionary in an order based on its optimization algorithm.
- You can use the **`sorted` method to traverse the keys of a dictionary in a sorted order**.
- Example

```
D = {'Name':'Skye', 'Age':35, 'Temperature':98.7, 'Last Name':
'Babic'}

# Iterate over the sorted keys of D
# Keys sorted alphabetically because they are all strings
for key in sorted(D.keys()):
    print("D[{}] = '{}'".format(key, D[key]))
```

Iterating over the values

- You can **iterate over the elements of a dictionary** using the values in the same way you did when iterating using the keys. This can be achieved using the **.values()** method.
- NOTE: Because the values in this example are of different types, you cannot sort them. If they were all of the same type, you could use the sorted() method to iterate over them in an ordered way.
- Example:

```
D = {'Name':'Skye', 'Age':35, 'Temperature':98.7, 'Last Name':
'Babic'}
for value in D.values():
    print(value)
```

Iterating over the keys and values

- You **can iterate over the key:value pairs** of a dictionary using the **.items()** method. The method returns a tuple as (key, value) for each pair in the dictionary.
- Example:

```
D = {'Name':'Skye', 'Age':35, 'Temperature':98.7, 'Last Name':
'Babic'}
for (key, value) in D.items():
    print(key,':', value)
```

Unit 3 Module 4 Proper Functions

Section 1 Script Environment and Command Line Arguments

Using the Terminal in Notebooks.Azure.com

- Launch the Terminal from the Library page
- Identify the Shell **`$ echo $0`**

- Print the Working Directory **`$ pwd`**
- List the Directory **`$ ls`** or **`$ dir`**
- Clear the Terminal Display **`$ clear`**
- Change the Directory **`$ cd [directory path]`** or **`>cd ..`**
- Run a Python file **`$ python3 [file_path]`**
- Run Python in Terminal **`$ python3 `** and exit Python **`$>>>exit()`**

Script Environment

- A Python script can be executed directly or imported as a module in another script.
- When running a script, a Python interpreter goes through a **special setup procedure** that defines some **environment variables**.
    - One of those variables is the `**__name__**` variable, which can distinguish between the two cases.
        - When the script is executed directly, `__name__` contains the string `"__main__"`; otherwise, it contains the name of the module.
        - This distinction allows you to run parts of the code when the script is run directly, and run other parts when the script is imported as a module.

Running a script directly

- If a script (`main_script.py`) contains the following code:

```
print(__name__)
```

Running the script from a terminal window, will give you:

```
$ python main_script.py
__main__
```

Importing the script as a module

- If the script (`main_script.py`) containing:

```
print(__name__)
```

Is imported into another script (`secondary_script.py`) that contains the following code:

```
import main_script
```

- Running (`secondary_script.py`) will give you the name of the imported module (which is the name of the `main_script.py` file in this case)

```
$ python secondary_script.py
  main_script
```

`main()` function

- Generally, you test the value of `__name__` and execute a (`main()`) function if you are running the script directly. Otherwise, you do not run any function.

```
if __name__ == "__main__":
    main()
```

Command Line Structure

- A UNIX command is used to run a program and it has the following syntax.

```
command [arguments] [options]
```

- The arguments and options might be optional depending on the nature and purpose of the program.

Changing the working directory to `command_line`

- Necessary so all generated files are saved in this directory, the cell will generate an error message if you are already in the `command_line` directory.
- Syntax:

```
%cd command_line
```

- Commands, Options and Arguments for **`ls`** command

Command	Purpose
ls	This command **lists the content of a directory**. The command ls can be run without options or arguments to display the content of the current working directory.
-l	ls option that shows **more detailed** description about the content of the current working directory.
-a	ls option **displays hidden files and directories**. Note the `.` and `..` directories.
../parent_dir	**Argument for ls** for example, you can pass it a path to a directory to display its content. You can also use the options along with the arguments.

Parsing Command Line Arguments

- Scripts can be **executed from a terminal**.
- They can also be **executed with command-line arguments and options**.
- These arguments and options can be used within the script **to control the flow** of the program.
- The arguments and options are captured by an environment variable **`argv`**, which can be accessed using the sys module.
- The **`argv`** variable
    - **captures the command-line arguments and options as a list**.
    - The first element **argv[0]** is always the command itself, which is the name of the script file.
    - The rest of the list elements are the arguments and options.
    - It is possible to process these arguments and options by writing code around the argv list; however, it is a daunting and tedious task.
    - Command-line arguments are used by many applications; therefore, the Python standard library provides an **argparse module** that is much more robust and versatile and will make parsing command-line arguments very easy.

Parsing Command-Line Arguments

- Change the working directory to command_line
- Necessary so all generated files are saved in this directory, the cell will generate an error message if you are already in the command_line directory.

- Syntax:
  %cd command_line
- Using the argv environment variable

```
%%writefile command_line.py

import sys

# Number of arguments
argc = len(sys.argv)
print(argc, "arguments and options were passed")

# List of arguments and options
print("The arguments and options passed are: ")
for i in range(argc):
    print("argv[{:d}] = {}".format(i, sys.argv[i]))
```

Running the `command_line.py` script will generate:

- Code:

```
%%bash
python3 command_line.py arg1 arg2 -option1 -option2
```

- Output:

  5 arguments and options were passed
  The arguments and options passed are:
  argv[0] = command_line.py
  argv[1] = arg1
  argv[2] = arg2
  argv[3] = -option1
  argv[4] = -option2

Generating random numbers

-
- Use the **argparse module**
  - makes it easy to write user-friendly command-line interfaces
  - program defines what arguments it requires, and argparse will figure out how to parse those out of sys.argv
  - also automatically generates help and usage messages and issues errors when users give the program invalid arguments.
- Example:
- This program imports the `argparse` module to define an object of type `**argparse.ArgumentParser**`, then parses the command line arguments using `parse_args()`

```
%%writefile rand.py

import argparse
from random import randint

# Define an argument parser object
parser = argparse.ArgumentParser()
```

```
# Parse command-line arguments
args = parser.parse_args()

# Program
print(randint(0, 10))
```

- Running the script from a terminal will generate:

  Code:
  ```
  %%bash
      python3 rand.py
  ```

  Output: a random number

- **NOTE**: if we pass an unrecognized argument to the script, the argparse module will generate an appropriate usage message and automatically build a help page.

Adding Arguments

- The `**count**` argument **holds the number of random integers to print**.
  - If `count` is not provided by the user, the script won't work and the user will be presented with a usage message
  - The help is updated accordingly
  - The argument passed is stored in `args.count`
- When passing 4 as an argument, the script generates 4 random numbers as expected
  - The `add_argument` method takes several parameters:
  - Name of the argument, which is also the name of the variable storing the count.
  - Type of the argument; if not specified, the default is string
  - Message to be displayed when a user requests the help message by using the `-h` option
- The `add_argument` method takes more optional parameters depending on the way you want to capture the arguments. We will explore a few more in the next examples.
- The argument (count) is a **positional argument** because it is <u>required</u> and its position depends on the command itself.
  - '**`-r`**' is the short notation of the new argument; '--range' is the long notation. You can use them interchangeably.
  - **`metavar`** is the name that will be used in the help message.
  - **`nargs`** is the number of expected options after -r or --range; use '*' for unlimited. In this example, it will be 2, the lower and upper integer range limits.
  - **`type`** is the expected type (string by default).
  - **`default`** is the default range when not specifying a range.
  - You can access the range options using **`args.range[0]`** and **`args.range[1].`** If nargs was larger you could use the appropriate index to access the numbers passed.

    ```
    %%writefile rand.py

    import argparse
    from random import randint

    # Define an argument parser object
    parser = argparse.ArgumentParser()
    ```

```
# Add positional arguments
parser.add_argument('count', type = int, help = 'Count of random integers to be
generated')

# Add optional arguments
parser.add_argument('-r', '--range', metavar = 'number', nargs = 2, type = int, default =
[0, 10], help = 'Integer range [a, b] from which the random numbers will be chosen')

# Parse command-line arguments
args = parser.parse_args()

# Program
for i in range(args.count):
   print(randint(args.range[0], args.range[1]))
```

- More about **`metavar`**
  - In the previous example, the number of expected arguments after -r (or --range) was nargs = 2.
  - The help message illustrated that by -r number number (or --range number number).
  - The word number was specified using the metavar parameter.
  - The metavar was repeated 2 times to account for the 2 required arguments.
  - It is also possible to specify different names for each of the required arguments by putting them in a tuple.
  - In this example, the numbers passed to -r are renamed to lower and upper by assigning a tuple to metavar.
    %%writefile rand.py

```
import argparse
from random import randint

# define an argument parser object
parser = argparse.ArgumentParser()

# Add positional arguments
parser.add_argument('count', type = int, help = 'Count of random integers to be
generated')

# Add optional arguments
parser.add_argument('-r', '--range', metavar = ('lower', 'upper'), nargs = 2, type = int,
default = [0, 10], help = 'Integer range [a, b] from which the random numbers will be
chosen')

# parse command line arguments
args = parser.parse_args()

# program
```

```
for i in range(args.count):
    print(randint(args.range[0], args.range[1]))
```

- The metavar parameter can also be used with positional arguments to use an alternative name in the help message. However, only the displayed name is changed; the parse_args() attribute still has the original name.

- More about **`action`**
    - In the previous example, the action = 'store_true' was used to make -v (or --verbose) a Boolean flag that can be set to True or False.
    - Python supports other actions:
        - store: The default action for all arguments; it stores the value passed on the command line to the argument.
        - store_true and store_false: Make an argument a Boolean flag and set it to True or False when entered by the user.
        - store_const: Stores a value specified by the keyword const in the argument. This is a more general form of store_true that allows you to store non-Boolean values in the argument.
        - count: The number of times an argument is used by the user.
    - The example shows how these actions behave.

Section 2: Variable Scope

- A variable in Python lives within a **scope**; the scope **determines how the variable is accessed and when it is deleted**.
- A variable scope is **determined by the place where it is initially assigned**. There are two types of scopes: **local** and **global**.
- Example: Parameters passed to a function and variables assigned within it are within the local scope of the function and are called local variables; variables assigned outside all functions in a module are within the global scope of the module and are called global variables.
- Generally speaking:
    - Local variables cannot be read or modified from the global scope
    - Local variables cannot be read or modified from other local scopes
    - The same variable name can be used in different functions without causing a conflict

Isolated Local Scopes

- When a function calls a **subfunction**, the current variables within the function scope are stored in memory, and another temporary local scope is created to accommodate the subfunction variables.
- The temporary local scope is destroyed when the subfunction returns; at that point, the original local scope becomes active again.

Global Variables

- A global variable is **assigned outside all functions** and lives within the global scope of the module.

- It exists from the time of its assignment *until the program ends*.
- Global variables are **visible to all functions** within the module and can be used within their different local scopes.
- Can be used by expressions in the global scope.
- Can be changed from the global scope, and can also be modified from a local scope using the global statement (i.e. global x = 4).
- If (global) was not used, a local variable would be defined instead, and any changes to this new variable will not affect the global variable that bears the same name.
- Summary
    - Global variables are accessible from local scopes
    - Global variables can be changed from the global scope
    - Global variables can be changed from a local scope using the `global` statement
    - If a local variable shares the same name with a global variable, changes in the local will not affect the global

Section 3: Documenting Functions (Docstring)

- Essential part of software development.
- The way developers **record information** for later **reference** or to **communicate** ideas to other developers who will be using the code.
- Documenting code in Python is easily done using documentation strings (docstrings).
    - Docstrings are string literals that are used to document modules, classes, functions, and methods.
    - Docstrings start and end with triple quotes `""" some string """`.

One-line docstrings

- Used for simple functions with clear functionality:
- Example:

```
def double(x):
    """Return doubled x."""
    return x * 2
```

- Summary:
    - Use triple quotes even though it's only a single line docstring.
    - Describe the function as a command, not as a description (i.e. return x, compute b...).
    - End the string with a period.

Multi-line docstrings

- Used to describe functions more elaborately:
- Example:

```
def vowel_count(word):
    """
    Count the number of vowels in a word.

    args:
        word: string under test
```

```
    returns:
        count: number of counted vowels
    """

    count = 0
    for c in word:
        if c in "AEIOUYaeiouy":
            count = count + 1
    return count
```

- General notes:
    - Start with triple quotes.
    - Write a summary line as in the one-line docstring (as a command, ends with a period).
    - Follow the summary with a blank line.
    - You can write more description after the blank line, but this is optional.
    - List the function arguments, return values, and exceptions raised (if any).
    - End the docstring with triple quotes and a blank line.

Accessing docstrings

- Each function contains an attribute __doc__ that contains its docstring. A function's docstring can be accessed through this attribute or by using the help() function in a Python interpreter.

Section 4: Documenting Functions (pydoc)

Generating documentation by Using pydoc

- pydoc is a Python module that can automatically generate documentation using the docstrings in a module.
    - It imports the module to generate its documentation; therefore, you should always use **__name__ == "__main__" to suppress any function from running** when the documentation is being generated.
    - The output of pydoc can be in text format or HTML format.
- To generate text documentation for a module contained in test.py, run the following command:

```
pydoc test
```

- To generate HTML documentation for test.py, run the command with the -w flag as follows:

```
pydoc -w test
```