Python Summary Name:_____

Print Function: the `print()` function takes a list of values to print and writes them to the output, e.g.,

```
print('cat',5,'dog')
print()  # blank line
print('pi is about', 3.14)
```

```
cat 5 dog

pi is about 3.14
```

Optional keyword arguments can be used to replace the defaults: space-character (`' '`) as a `sep`arator, the new-line-character (`'\n'`) as the `end`ing character, and output file of the console (`sys.stdout`). The syntax with default parameters explicitly shown is:

```
print(value,...,sep=' ',end='\n', file=sys.stdout)
```

Print Function	Expected Output

`print('cat',5,end='')`
`print(' horse')`
`print('cow')`
	`cat 5 horse`
	`cow`

`print ('cat',5,'dog',sep='23',end='#')`	`cat23523dog#`
`print ('<',end='')` `print ('cat', 5, 'dog',sep='>, < ', end='>\n')`	`<cat>, <5>, <dog>`
`print ('cat',5,'dog',sep='23','horse')`	error since keyword arguments must be at the end of the parameter list

String Formatting: inside a string we use formatting placeholders (e.g., `%8d, %10s, %5.2f`), follow the string with the format operator (%), and a tuple supplying values for corresponding placeholders. For example: `print("Name: %10s Age: %-7d GPA: %.2f" % ("Bob", 20, 3.138))` outputs the line:

```
Name:        Bob Age: 20      GPA: 3.14
```

`%10s` means left-justify string in 10 spaces, `%-7d` means right-justify decimal/int in 7 spaces, and `%.2f` left-justify float using the minimum spaces but with 2 decimal places. Another example with a character and exponent: `print("character: %c float with exponent %e" % ('$', 123.456))` outputs the line:

```
character: $ float with exponent 1.234560e+02
```

Assignment Statement: the assignment statement creates a variable in memory and sets its value. The syntax is: <variable identifier> = <constant or expression value>, where identifiers must start with a letter or underscore (`'_'`), and then can be followed by letters, underscores, or digits.

Assignment Statements and Print Functions	Expected Output

`a = 123`
`b = a`
`a = a + 1`
`print ('a is', a)`
`print ('b is', b)`
	`a is 124`
	`b is 123`

`c = 'cat'`
`d = c`
`c = c + 'fish'  # string concatenation`
`print('c is', c)`
`print('d is', d)`
	`c is catfish`
	`d is cat`

`e = ['cat', 'dog']`
`f = e`
`e.append('cow')`
`print('e is',e)`
`print('f is', f)`
	`e = ['cat','dog','cow']`
	`f = ['cat','dog','cow']`

NOTE: This last example deals with assigning lists. In `e = ['cat', 'dog']` the variable `e` is assigned a reference/pointer to the list, so `f = e` assigns `f` a reference to the same list. There is only a single list! Thus, when we append `'cow'` to the list using `e`'s reference, it also printed in `f`'s list because it's the same list.

The first two examples deal with integers and strings which are *immutable* (i.e., unchangable). New immutable values are created with new references being assigned. After `b = a` both variables reference `123`. When `a = a + 1` executes, a new integer constant of `124` is created and its reference is assign to variable `a`. Variable `b` still references `123`.

Python Summary Name:_____

Input Function: the `input()` function reads a line from the keyboard (`sys.stdin`) and returns it as a string with the trailing new-line stripped. To input numeric values, the string needs to be explicitly cast (e.g. `eval(input())`). For example, we can input and echo the user's name and age.

```
name = input("Enter your name: ")
age = eval(input("Enter your age: "))
print('Hi', name, end='!')
print('  Your age is', age)
```

```
Enter your name: Bob
Enter your age: 10
Hi Bob!  Your age is 10
```

Control Statements: the body of control statements are indented and there is NO other "end" ("}") marker

`if` statements: An `if` statement allows code to be executed or not based on the result of a comparison. If the condition evaluates to True, then the statements of the **indented body** is executed. If the condition is False, then the body is skipped. The syntax of `if` statements is:

```
if <condition>:
    statement1
    statement2
    statement3
```

```
if <condition>:
    statementT1
    statementT2
else:
    statementF1
    statementF2
```

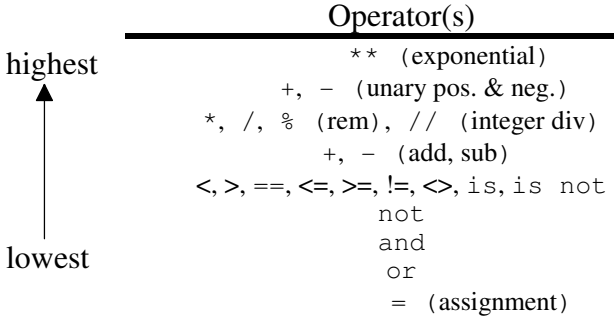
```
if <condition>:
    statementT1
    statementT2
elif <condition2>:
    statement
    statement
else:
    statementF1
    statementF2
```

Typically, the condition involves comparing "stuff" using relational operators (`<, >, ==, <=, >=, !=`).

Complex conditions might involve several comparisons combined using Boolean operators: `not, or, and`. For example, we might want to print "Your grade is B." if the variable score is less than 90, but greater than or equal to 80.

```
if score < 90 and score >= 80:
    print( "Your grade is B." )
```

The precedence for mathematical operators, Boolean operators, and comparisons are given in the table.



`for` loop: the `for` loop iterates once for each item in some sequence type (i.e, list, tuple, string).

```
for value in [1, 3, 9, 7]:
    print(value)
```

```
for character in 'house':
    print( character )
```

The `for` loop iterates over an iterable data-structure object (list, string, dictionary) or a range object created by the built-in `range` function which generate each value one at a time for each iteration of the loop. The syntax of: `range([start,] end, [, step])`, where [ ] are used to denote optional parameters. Examples:

- `range(5)` generates the sequence of values: `0, 1, 2, 3, 4`
- `range(2,7)` generates the sequence of values: `2, 3, 4, 5, 6`
- `range(10,2,-1)` generates the sequence of values: `10, 9, 8, 7, 6, 5, 4, 3`

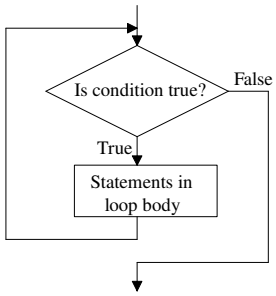
For example:

```
for count in range(1,6):
    print( count, end="  " )
print("\nDone")
```

```
1  2  3  4  5
Done
```

`while` loop: A `while` statement allows code to be executed repeated (zero or more times) as long as the condition evaluates to True. The syntax of a `while` statement is:

```
while <condition>:
    statement1
    statement2
    statement3
```



An *infinite loop* is one that would loop forever. (FYI, in a Python shell ctrl-c (^c) can be used to kill the running program.) Most infinite loops are caused by programmer error, but sometimes they are intentional. The following "*sentinel-controlled*" code uses an infinite loop and a *break* statement that immediately causes control to exit the loop.

```
total = 0
counter = 0
while True:     # an infinite loop
    score = eval(input("Enter a score (or negative value to exit): "))
    if score < 0:
        break
    total += score
    counter += 1
print("Average is", total/counter)
```

Strings: Strings in Python are sequential collections of only characters. **Strings are immutable (i.e., cannot be changed), so new strings are generated by string operations.** Operations on strings (**or any sequence collection**) include:

Operation	**Operator**	**Explanation**	**Example** myString = "Hello!!!" aString = "cat"	**Result of Example**
Indexing	`[ <index> ]`	Access the element specified by the index	`myString[1]`	`'e'`
Slicing	`[ : ]`	Extract a part of the string	`myString[ 1:5 ]`	`'ello'`
Concatenation	`+`	Combine strings together	`myString + aString`	`'Hello!!!cat'`
Repetition	`*`	Concatenate a repeated number of times	`aString * 3`	`'catcatcat'`
Membership	`in`	Ask whether a substring is in a string	`'ell' in myString`	`True`
Length	`len(string)`	How many items are in the string?	`len( myString )`	`8`

Indexing of strings starts with 0 on the left end, and -1 on the right end:

```
                    1111
          01234567890123
cheer = 'GO Panthers!!!'
          -4-3-2-1
```

Omitted indexes in a slice means "from the end." For example, `cheer[:4]` generates `'GO P'`.

Omitted indexes in a slice means "from the end." For example, `cheer[-4:]` generates `'s!!!'`.

String objects also have the following methods: (the `string` module can be imported to provide more operations.)

Method	Usage	Explanation
center	myString.center(w)	Returns a string with myString centered in a field of size w
ljust	myString.ljust(w)	Returns a string with myString left-justified in a field of size w
rjust	myString.rjust(w)	Returns a string with myString right-justified in a field of size w
upper	myString.upper( )	Returns a string with myString in all upper-case characters
lower	myString.lower( )	Returns a string with myString in all lower-case characters
strip	myString.strip( )	Returns a string with leading and trailing whitespace (space, tab, new-line) chars. removed. An optional string parameter can be used to supply characters to strip instead of whitespace.
count	myString.count(sub)	Returns number of occurrences of sub in myString (Optional parameters: myString.count(sub [, start [, end]])
endswith	myString.endswith(sub)	Returns True if myString ends with the substring sub; otherwise it returns False
startswith	myString.startswith(sub)	Returns True if myString starts with the substring sub; otherwise it returns False
isdigit	myString.isdigit( )	Returns True if myString contains only digits; otherwise it returns False
isalpha	myString.isalpha( )	Returns True if myString contains only letters; otherwise it returns False
split	myString.split( )	Returns a list of substrings of myString splits at whitespace characters. An optional string parameter can supply characters to split on.
find	myString.find(sub)	Returns the starting index of the first occurrence of sub. (Optional parameters: myString.find(sub [, start [, end]])
replace	myString.replace(old,new)	Returns a string with all occurrences of substring old replaced by substring new. An additional integer parameter can specify the number of replacements to perform, e.g.,myString.replace(old,new, 3)

Lists: A Python list is also a sequence collection, but a list can contain items of any type (e.g., character, strings, integers, floats, other lists, etc.), and **lists are mutable**. Lists are represented by comma-separated values enclosed in square brackets ('[', ']'). Operations on lists (**or any sequence collection**, e.g., strings) include:

Operation	Operator	Explanation	Example myList=[5,6,7,8] ListB=[8,9]	Result of Example
Indexing	[<index>]	Access the element specified by the index	myList[2]	7
Slicing	[:]	Extract a part of the list	myList[1:3]	[6, 7]
Concatenation	+	Combine lists together	myList + ListB	[5, 6, 7, 8, 8, 9]
Repetition	*	Concatenate a repeated number of times	ListB * 3	[8, 9, 8, 9, 8, 9]
Membership	in	Ask whether an item is in a list	3 in myList	False
Length	len(list)	How many items are in the list?	len(myList)	4

The following list methods are provided by Python:

Method	Usage	Explanation
append	`myList.append(item)`	Adds item to the end of myList
extend	`myList.extend(otherList)`	Extends myList by adding all items in otherList to myList's end
insert	`myList.insert(i, item)`	Insert item in myList at index i
pop	`myList.pop( )`	Remove and return the last item in myList
pop(i)	`myList.pop(i)`	Remove and return the ith item in myList *
del	`del myList[i]`	Deletes the item in the ith position of myList *
remove	`myList.remove(item)`	Removes the first occurrence of item in myList **
index	`myList.index(item)`	Returns the index of the first occurrence of item in myList **
count	`myList.count(item)`	Returns the number of occurrences of item in myList
sort	`myList.sort( )`	Modifies myList to be sorted
reverse	`myList.reverse( )`	Modifies myList to be in reverse order

* Note: raises an IndexError if the index `i` is not in the list

** Note: raises a ValueError if the `item` is not in the list

Tuples: A tuple is another sequence data type, so the sequence operations of indexing, slicing, concatenation, repetition, membership (in), and len() work on tuples too. Tuples are very similar to lists, i.e., comma-separated items enclosed in parentheses. The main difference is that **tuples are immutable** (cannot be modified).

Create two tuples as:

```
student1 = ('Bob', 123456, 'Jr', 3.12)
student2 = 'Sally', 654321, 'Fr', 0.0
```

In addition to indexing, "fields" of a tuple can be *unpacked* using a single assignment statement as:

```
name, idnum, rank, gpa = student1
```

(NOTE: This allows multiple values to be returned from a function)

Dictionaries: A dictionary is an unordered set of key-value pairs (written as key:value). Keys must be unique and immutable (e.g., numerics, strings, tuples of immutable objects). Dictionaries are typically used to lookup the value corresponding to a specified key. Dictionaries can be written as comma-separated `key:value` pairs enclosed in curly braces. For example,

```
phoneNumbers = {'fienup':35918,'gray':35917,'east':32939,'drake':35811,'schafer':32187}
```

Access to individual `key:value` pairs looks syntactically like a sequence lookup using a key instead of an index. For example, `phoneNumbers['east']` returns 32939, and a new `key:value` pair can be added by `phoneNumbers['wallingford'] = 35919`. Additional, methods on dictionaries are:

Method	Usage	Explanation
keys	`myDictionary.keys()`	Returns keys in an interable `dict_keys` object
values	`myDictionary.values()`	Returns values in an interable `dict_values` object
items	`myDictionary.items()`	Returns key:value tuples in an interable `dict_items` object
get item	`value = myDictionary[myKey]`	Returns the value associated with myKey; otherwise raises a *KeyError* if myKey is not in the dictionary
get	`myDictionary.get(myKey)`	Returns the value associated with myKey; otherwise *None*
get	`myDictionary.get(myKey, alt)`	Returns the value associated with myKey; otherwise alt
in	`myKey in myDictionary`	Returns True if myKey is in myDictionary; otherwise False
del	`del myDictionary[myKey]`	Deletes the key:value pair whose key is myKey

Text Files: Below is a summary of the important text-file operations in Python.

File Operations in Python		
General syntax	**Example**	**Description**
open(filename) open(filename, mode)	`f = open('data.txt', 'w')`	Modes: `'r'` read only; `'w'` write only; `'a'` append; `'r+'` both reading and writing. Default mode is `'r'`
f.close()	`f.close()`	Close the file to free up system resources.
loop over the file object	`for line in f:` `  print (line)`	Memory efficient, fast and simple code to loop over each line in the file.
f.readline()	`nextLine = f.readline()`	Returns the next line from the file. The newline ('\n') character is left at the end of the string.
f.write(string)	`f.write('cats and dogs\n')`	Writes the string to the file.
f.read( )	`all = f.read()`	Returns the whole file as a single string.
f.read(size)	`chunk = f.read(100)`	Returns a string of at most 100 (size) bytes. If the file has been completely read, an empty string is returned.
f.readlines()	`allLines = f.readlines()`	Returns a list containing all the lines of the file.
f.readlines(size)	`someLines = f.readlines(5000)`	Returns the next 5000 bytes of line. Only complete lines will be returned.

Below is a summary of the important file-system functions from the `os` and `os.path` modules in Python.

`os` Module File-system Functions	
General syntax	**Description**
`getcwd( )`	Returns the complete path of the current working directory
`chdir(path)`	Changes the current working directory to path
`listdir(path)`	Returns a list of the names in directory named path
`mkdir(path)`	Creates a new directory named path and places it in the current working directory
`rmdir(path)`	Removes the directory named path from the current working directory
`remove(path)`	Removes the file named path from the current working directory
`rename(old, new)`	Renames the file or directory named old to new

`os.path` Module File-system Functions	
General syntax	**Description**
`exists(path)`	Returns True if path exists and False otherwise
`isdir(path)`	Returns True if path is a directory and False otherwise
`isfile(path)`	Returns True if path is a file and False otherwise
`getsize(path)`	Returns the size in bytes of the object named path

NOTE: The initial "current working directory" is the directory where the program is located. Typically, it is useful to access files relative to the "current working directory" instead of specifying an absolute (complete) path. You can use the strings:

- `'.'` to specify the current working directory, e.g., `currentDirectoryList = os.listdir('.')`
- `'..'` to specify the parent of current working directory, e.g., `os.chdir('..')` which changes the current working directory to the parent directory

Functions:

A *function* is a procedural abstract, i.e., a named body of code that performs some task when it is called/invoked. Often a function will have one or more parameter that allows it to perform a more general (variable) task. For example, the cube function below can be called with any numeric value with the corresponding cube of that number being returned.

```
# Function to calculate the cube of a number
def cube(num):
    num_squared = num * num
    return num_squared * num

# call the function
value = 2
print('The value', value, 'raised to the power 3 is', cube(value))
print('The value 3 raised to the power 3 is', cube(3))
```

Terminology:

- a *formal parameter* is the name of the variable used in the function definition. It receives a value when the function is called. In the function cube, `num` is the formal parameter. Formal parameters are only known inside of the function definition. The section of a program where a variable is known is called its *scope*, so the scope of a formal parameter (and other *local variable* defined in the function such as num_squared) is limited to the function in which it is defined.
- an *actual parameter/argument* is the value used in the function call that is sent to the function. In the call to function cube, the variable `value` supplies the actual parameter value of 2.
- a *global variable* is created outside all functions and is known throughout the whole program file, e.g. `value`.

It is helpful to understand the "rules of the game" when a function is called. Memory is used to store the current program and the data associated with it. The memory used to store the data is divided as shown below.

- Global memory is used to store the global variables (and constants).
- The *heap* is used to store dynamically allocated objects as the program runs, e.g. lists, strings, ints, objects
- The *run-time stack* is used to store *call-frames* (or *activation records*) that get *pushed* on the stack when a function is called, and *popped* off the stack when a function returns.

When a function is called the section of code doing the calling is temporarily suspended, and a new call-frame gets pushed on top of the stack before execution of the function body. The call-frame contains the following information about the function being called:

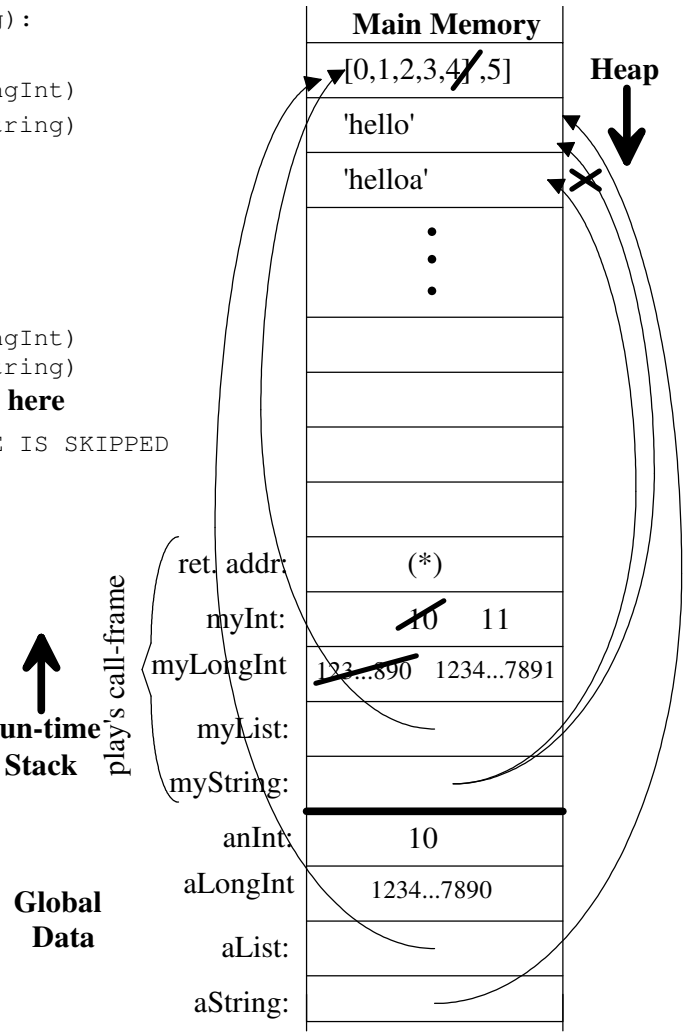
- the *return address* -- the spot in code where the call to the function occurred. This is needed so execution (control) can return there when the end of the function is reached or a `return` statement executes.
- room to store the formal parameters used by the function. In Python, parameters are *passed-by-value* which means that the value of each actual parameter in the function call is assigned to the corresponding formal parameter in the function definition before the function starts executing. However, the memory location for actual parameters for strings, lists, dictionaries, tuples, atomic objects contain only *references* to the heap
- room to store the local variables defined in the function (these are probably references to objects in the heap)

When a function returns, execution resumes at the function call (which is specified by the return address). A function typically sends back a value to the call by specifying an expression after `return` in the return statement. In Python if no expression is specified returned, then the special object `None` is returned.

```
def play(myInt, myLongInt, myList, myString):
    print('START OF play Function')
    print('myInt=',myInt,'myLongInt=',myLongInt)
    print('myList=',myList,'myString=',myString)
    myInt += 1
    myLongInt += 1
    myList.append(1)
    myString +='a'
    print('END OF play Function')
    print('myInt=',myInt,'myLongInt=',myLongInt)
    print('myList=',myList,'myString=',myString)
    return
```

Trace up to here

```
anInt = 10  # 1st STATEMENT EXECUTED, ABOVE IS SKIPPED
aLongInt = 12345678901234567890123456789OL
aList = range(5)
aString = 'hello'
print('BEFORE CALL')
print'anInt=',anInt,'aLongInt=',aLongInt)
print('aList=',aList,'aString=',aString)
play(anInt, aLongInt, aList, aString)
print('AFTER CALL') (*)
print('anInt=',anInt,'aLongInt=',aLongInt)
print('aList=',aList,'aString=',aString)
```



Output of complete program:

```
>>>
BEFORE CALL
anInt= 10 aLongInt= 12345678901234567890123456789O
aList= [0, 1, 2, 3, 4] aString= hello
START OF play Function
myInt= 10 myLongInt= 12345678901234567890123456789O
myList= [0, 1, 2, 3, 4] myString= hello
END OF play Function
myInt= 11 myLongInt= 12345678901234567890123456789O1
myList= [0, 1, 2, 3, 4, 1] myString= helloa
AFTER CALL
anInt= 10 aLongInt= 12345678901234567890123456789O
aList= [0, 1, 2, 3, 4, 1] aString= hello
>>>
```

NOTEs:

- `aList` passes a copy of **its reference to** the list as the initial value of the formal parameter `myList`, so both refer to the single list. In `play` when `myList.append(1)` executes, the single list is changed. Thus, when `play` terminates, `aList` still reflect this change.
- `aString` passes a copy of its reference to the string ('hello') as the initial value of the formal parameter `myString`. Since strings are immutable (cannot be changed) in Python, executing `myString += 'a'` cause a whole new string 'helloa' to be recreated in memory with `myString` referencing it. `aString` still refers to the string 'hello'.
- the `int` objects should actually be stored in the heap like the string objects since `int`'s are immutable .

Classes: A *class* definition is like a blueprint (receipe) for each of the objects of that class.
A class specifies a set of data attributes and methods for the objects of that class

- The values of the data attributes of a given object make up its state
- The behavior of an object depends on its current state and on the methods that manipulate this state
- The set of a class's methods is called its *interface*

The general syntax of class definition is:

```
class MyClass [ ( superClass1 [, superClass2 ]* ) ]:
      '''Document comment which becomes the __.doc__ attribute for the class'''
      def __init__(self, [param [, param]*):
            '''Document comment for constructor method with self be referencing to the object itself'''
            #__init__body

      # defs of other class methods and assignments to class attributes

# end class MyClass
```

```
"""
File: simple_die.py
Description:  This module defines a six-sided Die class.
"""

from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        """The initial face of the die."""
        self._currentRoll = randint(1, 6)

    def roll(self):
        """Resets the die's value to a random number
        between 1 and 6."""
        self._currentRoll = randint(1, 6)

    def getRoll(self):
        """Returns the face value of the die."""
        return self._currentRoll

    def __str__(self):
        """Returns the string representation of the die."""
        return str(self._currentRoll)
```

Consider the following script to test the Die class and its associated output:

```
# testDie.py - script to test Die class
from simple_die import Die

die1 = Die()
die2 = Die()
print('die1 =', die1)    #calls __str__
print('die2 =', die2)
print()
print('die1.getRoll() = ', die1.getRoll())
print('die2.getRoll() = ', die2.getRoll())
die1.roll()
print('die1.getRoll() = ', die1.getRoll())
print('str(die1): ' + str(die1))
print('die1 + die2:', die1.getRoll() + die2.getRoll())
```

```
>>>
die1 = 2
die2 = 5

die1.getRoll() =  2
die2.getRoll() =  5
die1.getRoll() =  3
str(die1): 3
die1 + die2: 8
>>>
```

Classes in Python have the following characteristics:

- all class attributes (data attributes and methods) are *public* by default, unless your identifier starts with a single underscores, e.g, `self._currentRoll`
- all data types are objects, so they can be used as inherited base classes
- **objects are passed by reference when used as parameters to functions**
- all classes have a set of standard methods provided, but may not work properly (__str__, __doc__, etc.)
- most built-in operators (+, -, *, <, >, ==, etc.) can be redefined for a class. This makes programming with objects a lot more intuitive. For example suppose we have two Die objects: `die1` & `die2`, and we want to add up their combined rolls. We could use *accessor methods* to do this:

  ```
  diceTotal = die1.getRoll() + die2.getRoll()
  ```

  Here, the `getRoll` method returns an integer (type int), so the '+' operator being used above is the one for ints. But, it might be nice to "overload" the + operator by defining an __add__ method as part of the Die class, so the programmer could add dice directly as in:

  ```
  diceTotal = die1 + die2
  ```

The three most important features of *Object-Oriented Programming* (OOP) to simplify programs and make them maintainable are:

1. *encapsulation* - restricts access to an object's data to access only by its methods
   - ⇨ helps to prevent indiscriminant changes that might cause an invalid object state (e.g., 6-side die with a of roll 8)
2. *inheritance* - allows one class (the *subclass*) to pickup data attributes and methods of other class(es) (the *parents*)
   - ⇨ helps code reuse since the subclass can extend its parent class(es) by adding addition data attributes and/or methods, or overriding (through polymorphism) a parent's methods
3. *polymorphism* - allows methods in several different classes to have the same names, but be tailored for each class
   - ⇨ helps reduce the need to learn new names for standard operations (or invent strange names to make them unique)

Consider using inheritance to extend the Die class to a generalized AdvancedDie class that can have any number of sides. The interface for the AdvancedDie class are:

Detail Descriptions of the AdvancedDie Class Methods

Method	Example Usage	Description
`__init__`	`myDie = AdvancedDie(8)`	Constructs a die with a specified number of sides and randomly rolls it (Default of 6 sides if no argument supplied)
`getRoll`	`myDie.getRoll()`	Returns the current roll of the die (inherited from Die class)
`getSides`	`myDie.getSides()`	Returns the number of sides on the die (did not exist in Die class)
`roll`	`myDie.roll()`	Rerolls the die randomly (By overriding the `roll` method of Die, an AdvancedDie can generate a value based on its # of sides)
`__eq__`	`if myDie == otherDie:`	Allows == operations to work correctly for AdvancedDie objects.
`__lt__`	`if myDie < otherDie:`	Allows < operations to work correctly for AdvancedDie objects.
`__gt__`	`if myDie > otherDie:`	Allows > operations to work correctly for AdvancedDie objects.
`__add__`	sum = myDie + otherDie	Allows the direct addition of AdvancedDie objects, and returns the integer sum of their current roll values.

`__str__`	Directly as:
	`myDie.__str__()`
	`str(myDie)`
	or indirectly as:
	`print myDie`
		Returns a string representation for the AdvancedDie. By overriding the __str__ method of the Die class, so the "print" statement will work correctly with an AdvancedDie.

Consider the following script and associated output:

```
# testAdvancedDie.py - script to test
AdvancedDie class
from advanced_die import AdvancedDie

die1 = AdvancedDie(100)
die2 = AdvancedDie(100)
die3 = AdvancedDie()

print( 'die1 =', die1 )    #calls __str__
print( 'die2 =', die2 )
print( 'die3 =', die3 )

print( 'die1.getRoll() = ', die1.getRoll())
print( 'die1.getSides() =', die1.getSides())
die1.roll()
print( 'die1.getRoll() = ', die1.getRoll())
print( 'die2.getRoll() = ', die2.getRoll())
print( 'die1 == die2:', die1==die2)
print( 'die1 < die2:', die1<die2)
print( 'die1 > die2:', die1>die2)
print( 'die1 != die2:', die1!=die2)
print( 'str(die1): ' + str(die1))
print( 'die1 + die2:', die1 + die2)

help(AdvancedDie)
```

```
die1 = Number of Sides=100 Roll=32
die2 = Number of Sides=100 Roll=76
die3 = Number of Sides=6 Roll=5
die1.getRoll() =  32
die1.getSides() = 100

die1.getRoll() =  70
die2.getRoll() =  76
die1 == die2: False
die1 < die2: True
die1 > die2: False
die1 != die2: True
str(die1): Number of Sides=100 Roll=70
die1 + die2: 146
Help on class AdvancedDie in module
advanced_die:

class AdvancedDie(simple_die.Die)
 |  Advanced die class that allows for
any number of sides
 |
 |  Method resolution order:
 |      AdvancedDie
 |      simple_die.Die
 |      __builtin__.object
 |
 |  Methods defined here:
```

Notice that the `testAdvancedDie.py` script needed to `import AdvancedDie`, but not the `Die` class.

The AdvancedDie class that inherits from the Die superclass.

```
"""
File:  advanced_die.py
Description: Provides a AdvancedDie class that allows for any number of sides
Inherits from the parent class Die in module die_simple
"""
from simple_die import Die
from random import randint

class AdvancedDie(Die):
    """Advanced die class that allows for any number of sides"""

    def __init__(self, sides = 6):
        """Constructor for any sided Die that takes an the number of sides
        as a parameter; if no parameter given then default is 6-sided."""

        Die.__init__(self)       # call Die parent class constructor
        self._numSides = sides
        self._currentRoll = randint(1, self._numSides)

    def roll(self):
        """Causes a die to roll itself -- overrides Die class roll"""
        self._currentRoll = randint(1, self._numSides)

    def __eq__(self, rhs_Die):
        """Overrides default '__eq__' operator to allow for deep comparison of Dice"""
        return self._currentRoll == rhs_Die._currentRoll

    def __lt__(self, rhs_Die):
        """Overrides default '__lt__' operator to allow for deep comparison of Dice"""
        return self._currentRoll < rhs_Die._currentRoll

    def __gt__(self, rhs_Die):
        """Overrides default '__gt__' operator to allow for deep comparison of Dice"""
        return self._currentRoll > rhs_Die._currentRoll

   def __str__(self):
        """Returns the string representation of the AdvancedDie."""
        return 'Number of Sides='+str(self._numSides)+' Roll='+str(self._currentRoll)

    def __add__(self, rhs_Die):
        """Returns the sum of two dice rolls"""
        return self._currentRoll + rhs_Die._currentRoll

    def getSides(self):
        """Returns the number of sides on the die."""
        return self._numSides
```