Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

1.1 ADT (Abstract Data Type):- A description of how we understand the given data and the operations which are involved. It does not concern how this data will be implemented.

1.2 Encapsulation:- The ADT (1.1) 'hides' the raw data from the user. In other words, it encapsulates this raw data, a process called "information hiding". Encapsulation is visualized below by the white circle.

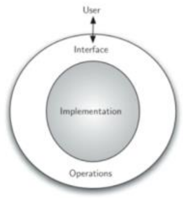


Figure 2: Abstract Data Type

1.3 Data Structure:- For the implementation of an ADT (1.1), referred to as a "data structure", we must first have recognized the data using programming concepts and data types. It is on these concepts and types we will build the more complex model.

1.4 Implementation-independent:- This is a way in which we can view the data, without noticing how it is implemented at the low-level processes.

Example:- The *array* type is an ADT. The user does not recognize how the *array* type is implemented; this is encapsulated. Thus, we can say the way of viewing *arrays* is implementation-independent.

Mnemonic:- A book is the ADT. A telephone book is the implementation.

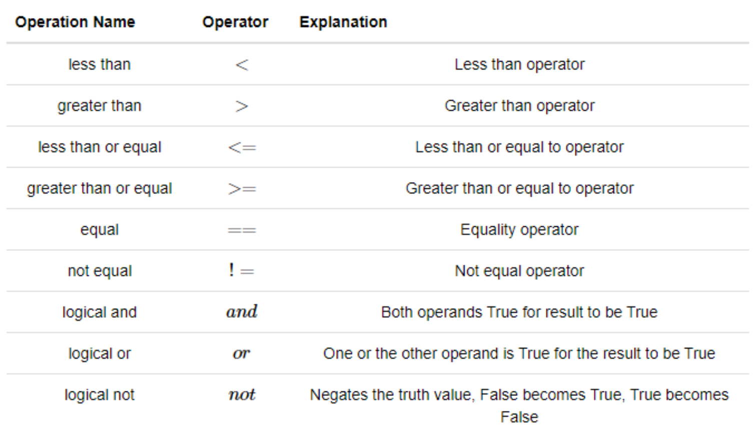
1.5 Classes:- A class is a coded description of the behaviour, and state, of our data. All classes are ADTs (1.1).

1.6 Objects:- Objects are the data items within classes (1.5) are called objects. Moreover, an object is an instance of a class.

1.7 Boolean objects:- A Boolean object is used mainly for representing truth or state values. These values can either be True, or False.

1.8 Logical operators:- To represent a logical *and* one should write "and". To represent a logical *or* one should write "or". To represent a logical *not* one should write "not". Refer to the table on the right (Table 1).

Table 1: Relational and Logical Operators



Operation Name	Operator	Explanation
less than	$<$	Less than operator
greater than	$>$	Greater than operator
less than or equal	$<=$	Less than or equal to operator
greater than or equal	$>=$	Greater than or equal to operator
equal	$==$	Equality operator
not equal	$!=$	Not equal operator
logical and	*and*	Both operands True for result to be True
logical or	*or*	One or the other operand is True for the result to be True
logical not	*not*	Negates the truth value, False becomes True, True becomes False

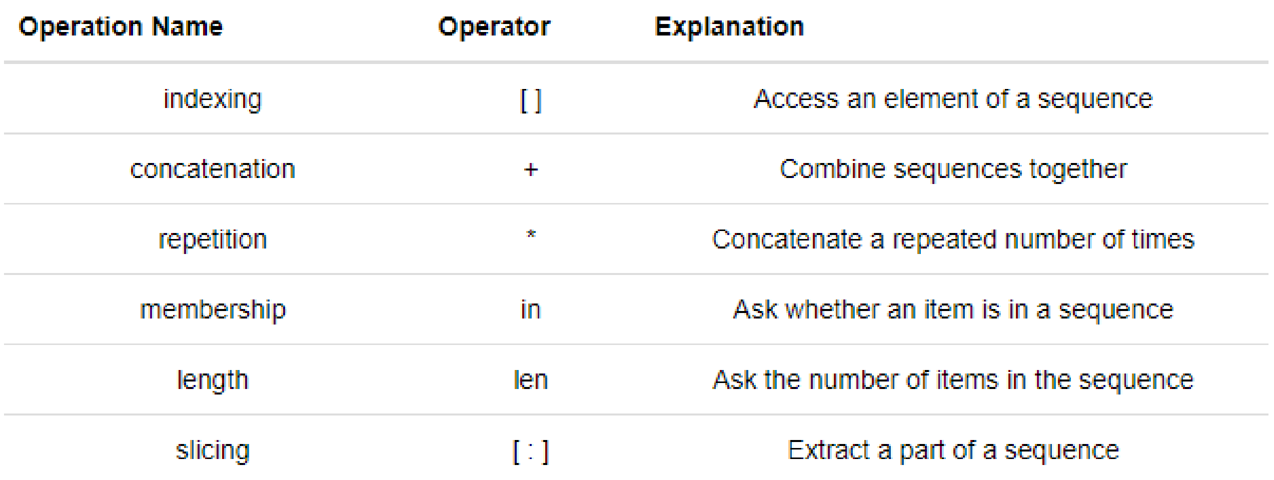
Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

1.9 Identifiers:- Identifiers are names given to variables. The naming convention is such that we are to start with either an underscore, "_", or a lower-case letter. If an identifier consists of multiple words, we are to separate those words using an underscore.

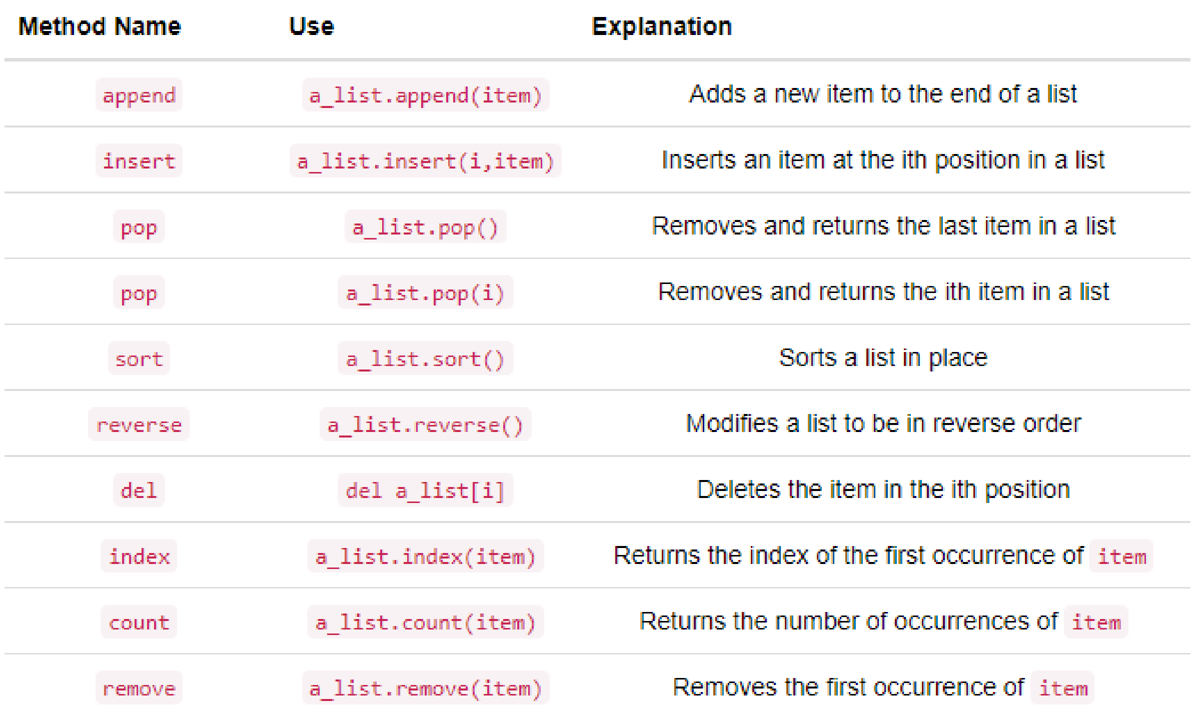
2.0 Lists:- A list is an ordered collection of references to objects. The items within lists are written as comma-delineated values, enclosed in square brackets. Lists can have multiple data classes (i.e., you may have integers, bool, and float values). Lists are indexed starting at zero, 0. Below is a table of list operations (Table 2). Further below is a table of manipulation methods for lists (Table 3).

Table 2: Operations on Any Sequence in Python



Operation Name	Operator	Explanation
indexing	[ ]	Access an element of a sequence
concatenation	+	Combine sequences together
repetition	*	Concatenate a repeated number of times
membership	in	Ask whether an item is in a sequence
length	len	Ask the number of items in the sequence
slicing	[:]	Extract a part of a sequence

Table 3: Methods Provided by Lists in Python



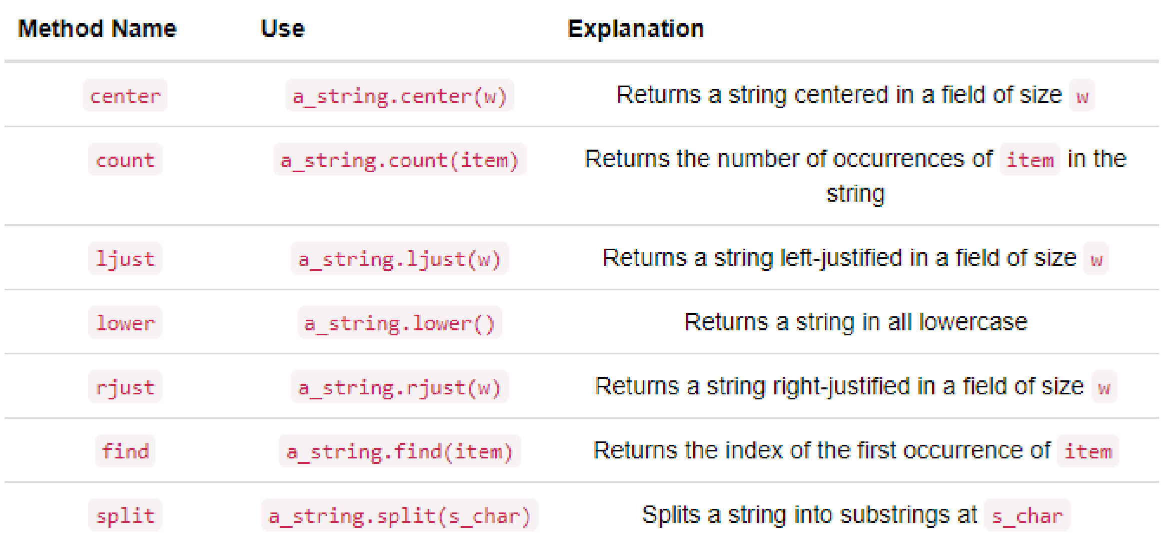
Method Name	Use	Explanation
`append`	`a_list.append(item)`	Adds a new item to the end of a list
`insert`	`a_list.insert(i,item)`	Inserts an item at the ith position in a list
`pop`	`a_list.pop()`	Removes and returns the last item in a list
`pop`	`a_list.pop(i)`	Removes and returns the ith item in a list
`sort`	`a_list.sort()`	Sorts a list in place
`reverse`	`a_list.reverse()`	Modifies a list to be in reverse order
`del`	`del a_list[i]`	Deletes the item in the ith position
`index`	`a_list.index(item)`	Returns the index of the first occurrence of `item`
`count`	`a_list.count(item)`	Returns the number of occurrences of `item`
`remove`	`a_list.remove(item)`	Removes the first occurrence of `item`

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

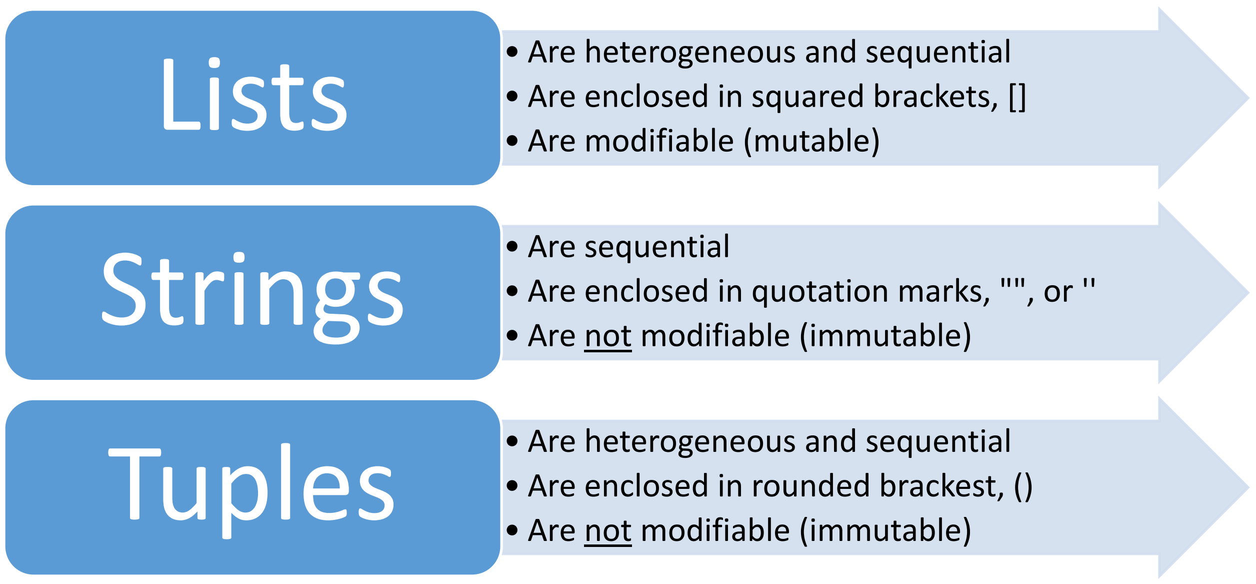
2.1 Strings:- Strings are collections of characters. Similarly, strings are sequential. Thus, the methods in the above two tables are applicable (Table 2, Table 3). Below is a table of manipulation methods for strings (Table 4).

Table 4: Methods Provided by Strings in Python



Method Name	Use	Explanation
`center`	`a_string.center(w)`	Returns a string centered in a field of size `w`
`count`	`a_string.count(item)`	Returns the number of occurrences of `item` in the string
`ljust`	`a_string.ljust(w)`	Returns a string left-justified in a field of size `w`
`lower`	`a_string.lower()`	Returns a string in all lowercase
`rjust`	`a_string.rjust(w)`	Returns a string right-justified in a field of size `w`
`find`	`a_string.find(item)`	Returns the index of the first occurrence of `item`
`split`	`a_string.split(s_char)`	Splits a string into substrings at `s_char`

2.2 Tuples:- Tuples are like lists, they are heterogeneous (meaning that they can have mixed classes) sequences of data. They are comma-delineated and enclosed using rounded brackets. Tuples cannot be changed once initialized. Thus, they can only use the operations outlined in Table 2.



Lists
- Are heterogeneous and sequential
- Are enclosed in squared brackets, []
- Are modifiable (mutable)

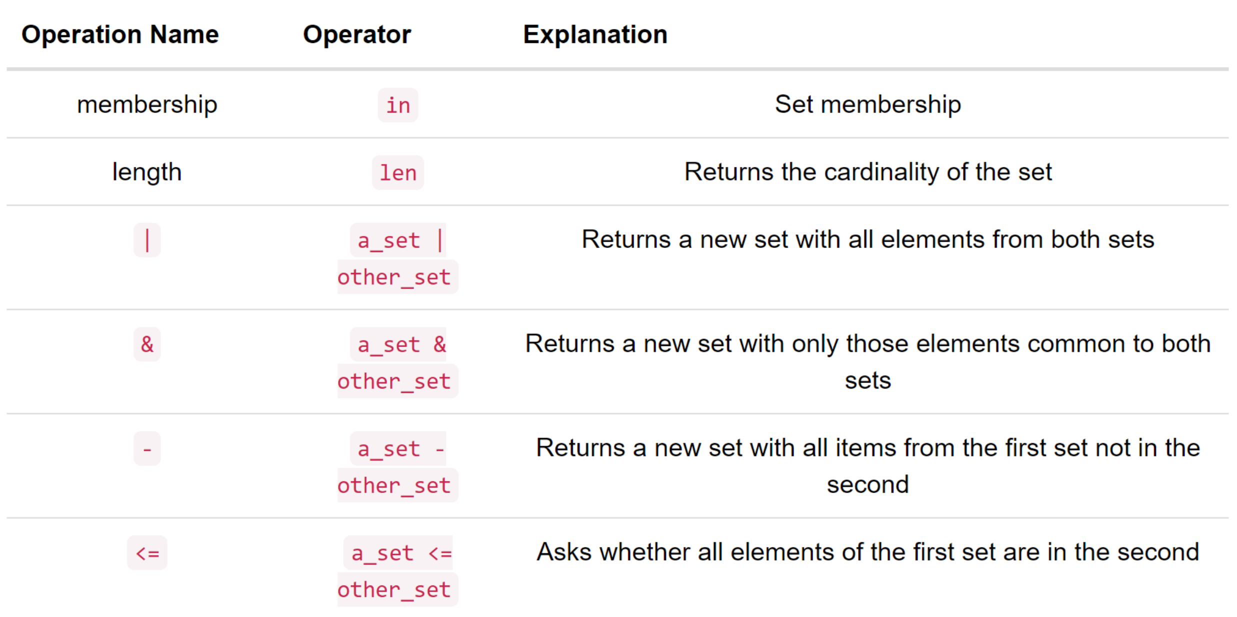
Strings
- Are sequential
- Are enclosed in quotation marks, "", or ''
- Are not modifiable (immutable)

Tuples
- Are heterogeneous and sequential
- Are enclosed in rounded brackest, ()
- Are not modifiable (immutable)

2.3 Mutability:- Classes which can be modified are referred to as "mutable". Classes which cannot be modified are referred to as "immutable".

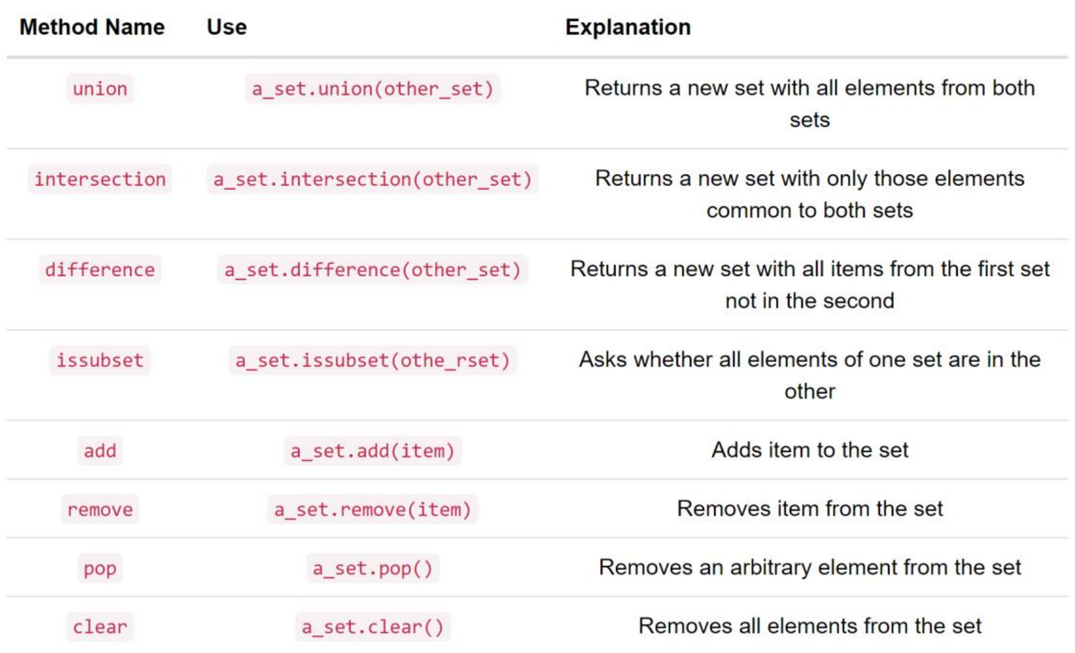
2.4 Sets:- A set is an unordered collection of immutable data objects. We cannot have any duplicate items in a set. They are comma-delineated, and enclosed in curly brackets. They are heterogeneous and are not considered sequential. Although, they do support the following operations (Table 5). Sets do also supports some manipulation methods, as can be seen in the table below (Table 6).

Table 5: Operations on a Set in Python



Operation Name	Operator	Explanation
membership	`in`	Set membership
length	`len`	Returns the cardinality of the set
`|`	`a_set	other_set`	Returns a new set with all elements from both sets
`&`	`a_set & other_set`	Returns a new set with only those elements common to both sets
`-`	`a_set - other_set`	Returns a new set with all items from the first set not in the second
`<=`	`a_set <= other_set`	Asks whether all elements of the first set are in the second

Table 6: Methods Provided by Sets in Python



Method Name	Use	Explanation
`union`	`a_set.union(other_set)`	Returns a new set with all elements from both sets
`intersection`	`a_set.intersection(other_set)`	Returns a new set with only those elements common to both sets
`difference`	`a_set.difference(other_set)`	Returns a new set with all items from the first set not in the second
`issubset`	`a_set.issubset(othe_rset)`	Asks whether all elements of one set are in the other
`add`	`a_set.add(item)`	Adds item to the set
`remove`	`a_set.remove(item)`	Removes item from the set
`pop`	`a_set.pop()`	Removes an arbitrary element from the set
`clear`	`a_set.clear()`	Removes all elements from the set

2.5 Dictionaries:- Dictionaries are comma-delineated unordered collections of associated pairs of items. Each pair has a "key" and a subsequent "value", usually written as "key:value". They are enclosed in curly brackets. Dictionaries can be manipulated by accessing a value, via a key, or by adding a new key:value pair. The syntax is like that of sequences, yet we replace the index with the key. The operations available for dictionaries is shown in the below table (Table 7). Dictionaries can be manipulated using the methods shown in the second table below (Table 8).

Table 7: Operators Provided by Dictionaries in Python

Operator	Use	Explanation
`[]`	`a_dict[k]`	Returns the value associated with `k`, otherwise its an error
`in`	`key in a_dict`	Returns `True` if key is in the dictionary, `False` otherwise
`del`	del `a_dict[key]`	Removes the entry from the dictionary

Table 8: Methods Provided by Dictionaries in Python

Method Name	Use	Explanation
`keys`	`a_dict.keys()`	Returns the keys of the dictionary in a dict_keys object
`values`	`a_dict.values()`	Returns the values of the dictionary in a dict_values object
`items`	`a_dict.items()`	Returns the key-value pairs in a dict_items object
`get`	`a_dict.get(k)`	Returns the value associated with `k`, `None` otherwise
`get`	`a_dict.get(k, alt)`	Returns the value associated with `k`, `alt` otherwise

Example:- A dictionary can be initialized, and manipulated as follows:

```python
capitals = {"Iowa": "Des Moines", "Wisconsin": "Madison"}
print(capitals["Iowa"])
capitals["Utah"] = "Salt Lake City"
print(capitals)
capitals["California"] = "Sacramento"
print(len(capitals))
for k in capitals:
    print(capitals[k],"is the capital of", k)

```

Which would output the following:

```
Des Moines
{'Iowa': 'Des Moines', 'Wisconsin': 'Madison', 'Utah': 'Salt Lake City'}
4
Des Moines is the capital of Iowa
Madison is the capital of Wisconsin
Salt Lake City is the capital of Utah
Sacramento is the capital of California
```

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

2.6 Formatted strings:- Formatted strings are templates which allow for the string to update with the input given by the user. In the below table all of the different string formatting methods can be found (Table 9, Table 10).

Table 9: String Formatting Conversion Characters

Character	Output Format
`d`, `i`	Integer
`u`	Unsigned integer
`f`	Floating point as m.ddddd
`e`	Floating point as m.dddde+/-xx
`E`	Floating point as m.ddddE+/-xx
`g`	Use `%e` for exponents less than -4 or greater than $+5$, otherwise use `%f`
`c`	Single character
`s`	String, or any Python data object that can be converted to a string by using the `str` function
`%`	Insert a literal `%` character

Table 10: Additional formatting options

Modifier	Example	Description
number	`%20d`	Put the value in a field width of 20
`-`	`%-20d`	Put the value in a field 20 characters wide, left-justified
`+`	`%+20d`	Put the value in a field 20 characters wide, right-justified
`0`	`%020d`	Put the value in a field 20 characters wide, fill in with leading zeros
`.`	`%20.2f`	Put the value in a field 20 characters wide with 2 characters to the right of the decimal point
`(name)`	`%(name)d`	Get the value from the supplied dictionary using `name` as the key

Formatted string can be either Tuples (2.2) or Dictionaries (2.5).

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

Example:- Formatting a tuple string may follow these pieces of code

```
>>> price = 24
>>> item = "banana"
>>> print("The %s costs %d cents" % (item, price))
The banana costs 24 cents
>>> print("The %+10s costs %5.2f cents" % (item, price))
The     banana costs 24.00 cents
>>> print("The %+10s costs %10.2f cents" % (item, price))
The     banana costs      24.00 cents
```

Formatting a dictionary string may follow this piece of code

```
>>> itemdict = {"item": "banana", "cost": 24}
>>> print("The %(item)s costs %(cost)7.1f cents" % itemdict)
The banana costs    24.0 cents
```

2.7 Formatter class:- The new Formatter class is often used to implement complex string formatting.

Example:- Here is an example of the new Formatter class

```
>>> print("The {} costs {} cents".format(item, price))
The banana costs 24 cents
>>> print("The {:s} costs {:d} cents".format(item, price))
The banana costs 24 cents
```

2.8 F-strings:- f-strings are a way to use undeclaraized item names, instead of placeholders. The alignment symbols are different from those shown above (Table 9, Table 10). Please find the appropriate formatting options in the table below (Table 11).

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

Table 11: f-string Formatting Options

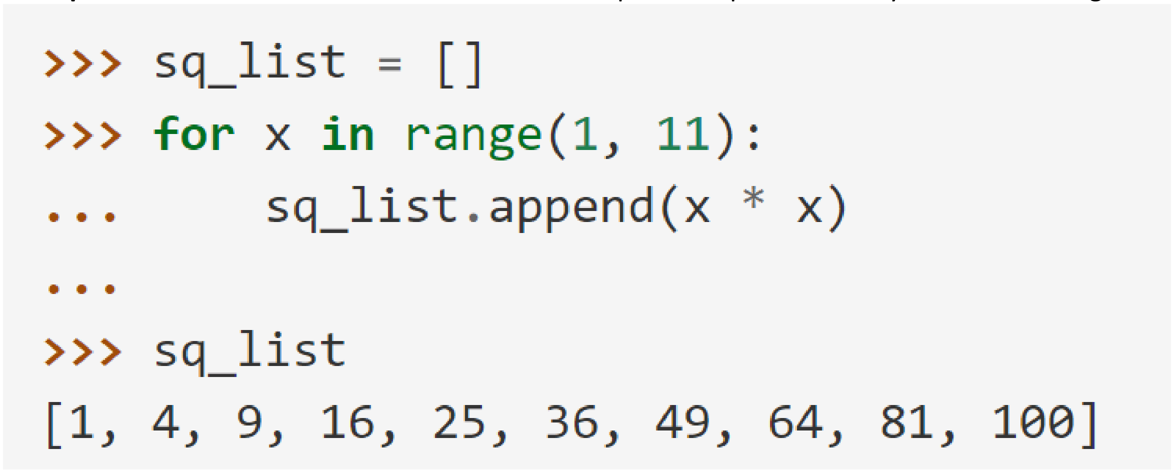
Modifier	Example	Description
number	`:20d`	Put the value in a field width of 20
`<`	`:<20d`	Put the value in a field 20 characters wide, left-aligned
`>`	`:>20d`	Put the value in a field 20 characters wide, right-aligned
`^`	`:^20d`	Put the value in a field 20 characters wide, right-aligned
`0`	`:020d`	Put the value in a field 20 characters wide, fill in with leading zeros.
`.`	`:20.2f`	Put the value in a field 20 characters wide with 2 characters to the right of the decimal point.

Example:- Here is an example of implementing the f-string method

```
>>> print(f"The {item:10} costs {price:10.2f} cents")
The banana     costs      24.00 cents
>>> print(f"The {item:<10} costs {price:<10.2f} cents")
The banana     costs 24.00      cents
>>> print(f"The {item:^10} costs {price:^10.2f} cents")
The   banana   costs   24.00    cents
>>> print(f"The {item:>10} costs {price:>10.2f} cents")
The     banana costs      24.00 cents
>>> print(f"The {item:>10} costs {price:>010.2f} cents")
The     banana costs 0000024.00 cents
>>> itemdict = {"item": "banana", "price": 24}
>>> print(f"Item:{itemdict['item']:.>10}\n" +
... f"Price:{'$':.>4}{itemdict['price']:5.2f}")
Item:....banana
Price:...$24.00
```

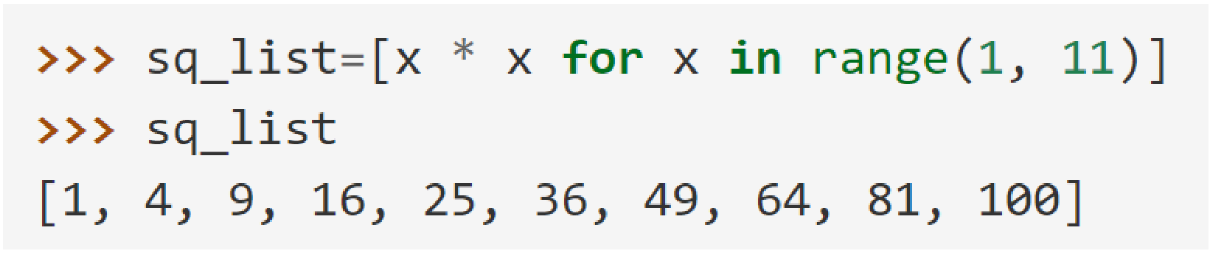
2.9 List comprehension:- List comprehension allows you to create a list based on some criteria.

Example:- If we want to create a list of the first ten perfect squares we may do the following



```
>>> sq_list = []
>>> for x in range(1, 11):
...     sq_list.append(x * x)
...
>>> sq_list
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Using list comprehension we can shorten this to

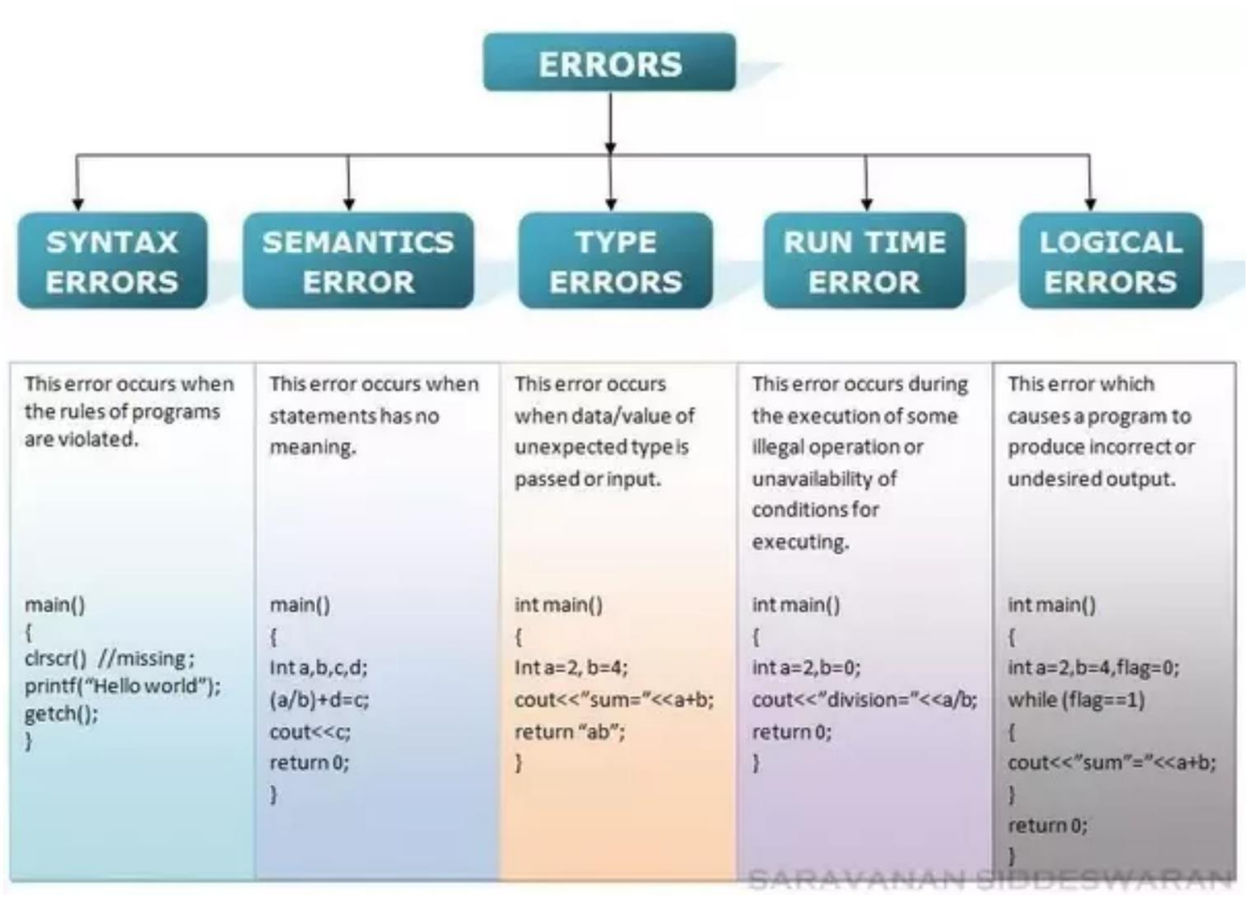


```
>>> sq_list=[x * x for x in range(1, 11)]
>>> sq_list
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

3.0 Logic errors:- A logic error is a problem with the the result that a programme provides. The code runs problem-free (i.e., not syntax errors, or otherwise) but the final answer is incorrect. Logic errors may lead to runtime errors. The diagram below outlines the differences in error types.

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]



SYNTAX ERRORS	SEMANTICS ERROR	TYPE ERRORS	RUN TIME ERROR	LOGICAL ERRORS
This error occurs when the rules of programs are violated.	This error occurs when statements has no meaning.	This error occurs when data/value of unexpected type is passed or input.	This error occurs during the execution of some illegal operation or unavailability of conditions for executing.	This error which causes a program to produce incorrect or undesired output.
main() { clrscr() //missing ; printf("Hello world"); getch(); }	main() { Int a,b,c,d; (a/b)+d=c; cout<<c; return 0; }	int main() { Int a=2, b=4; cout<<"sum="<<a+b; return "ab"; }	int main() { int a=2,b=0; cout<<"division="<<a/b; return 0; }	int main() { int a=2,b=4,flag=0; while (flag==1) { cout<<"sum"="<<a+b; } return 0; }

3.1 Exceptions:- Exceptions are runtime errors which lead to programme termination (i.e., division by zero). We say that exceptions are "raised" when they appear, and they are "handled" once they're fixed.

3.2 Try statement:- The try statement is a way to handle an exception.

3.3 Except statement:- The except statement follows the try statement (3.2). It catches the exception and prints a message describing where the exception occurred.

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

Example:- Kindly refer to this example as it pertains to except and try statements.

```
>>> try:
...    print(math.sqrt(a_number))
... except:
...    print("Bad value for the square root function")
...    print("Using the absolute value instead")
...    print(math.sqrt(abs(a_number)))
...
Bad value for the square root function
Using the absolute value instead
4.795831523312719
```

If the except portion of the code is reached (i.e., the try portion fails) the programme will not terminate but rather calculate the square root using the absolute value.

3.3 Raise statement:- The raise statement can be used to force a runtime exception. This still terminated the programme, but the error statement is now what the programmer decides it to be.

Example:- Kindly refer to this example for the raise statement.

```
>>> if a_number < 0:
...    raise RuntimeError("You can't use a negative number")
... else:
...    print(math.sqrt(a_number))
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
RuntimeError: You can't use a negative number
```

3.4 Fraction class:- The fraction class allows a user to create an accurate representation of decimal numbers (i.e., a fraction with both a numerator and denominator $\frac{a}{b}$). These types can behave the same as any other number (subtraction, summing, multiplication, division). The fractions types always return with the lowest common denominator.

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

Notation:-

```
class Fraction:
    """Class Fraction"""
    def __init__(self, top, bottom):
        """Constructor definition"""
        self.num = top
        self.den = bottom
```

If we wish to call the function we should use the following format

```
my_fraction = Fraction(3, 5)
```

In order to print the above, "my_fraction" follow this format

```
def show(self):
        print(f"{self.num}/{self.den}")
```

```
>>> my_fraction = Fraction(3, 5)
>>> my_fraction.show()
3/5
>>> print(my_fraction)
<__main__.Fraction object at 0x40bce9ac>
```

We can also print using this __str__ method

```
def __str__(self):
    return f"{self.num}/{self.den}"
```

```
>>> my_fraction = Fraction(3, 5)
>>> print(my_fraction)
3/5
>>> print(f"I ate {my_fraction} of pizza")
I ate 3/5 of pizza
>>> my_fraction.__str__()
'3/5'
>>> str(my_fraction)
'3/5'
```

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

3.5 Self:- Self is a reference back to the object itself. It must always be <u>the first formal parameter</u>. The self.den parameter creates to the denominator, likewise with the self.num.

3.6 Show:- Show is a way for an object to print itself (shown in above notation).

3.7 __str__:- The __str__ method converts an object into a string. Thereby, a string-ified version of the object can be printed using print(…). If this method is used with the fraction class type then there will be a "/" placed in between the numerator value and denominator value.

3.8 __add__:- If we want to add fractions we cannot simply do Fraction1 + Fraction2. We must use the __add__ method.

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

Example:- Adding using the __add__ method can be done as follows

```
def __add__(self, other_fraction):
     new_num = self.num * other_fraction.den + \
                  self.den * other_fraction.num
     new_den = self.den * other_fraction.den

     return Fraction(new_num, new_den)
```

```
>>> f1 = Fraction(1, 4)
>>> f2 = Fraction(1, 2)
>>> f3 = f1 + f2
>>> print(f3)
6/8
```

To find the LCD use the following code

```
def gcd(m, n):
    while m % n != 0:
        m, n = n, m % n
    return n

print(gcd(20, 10))
```

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

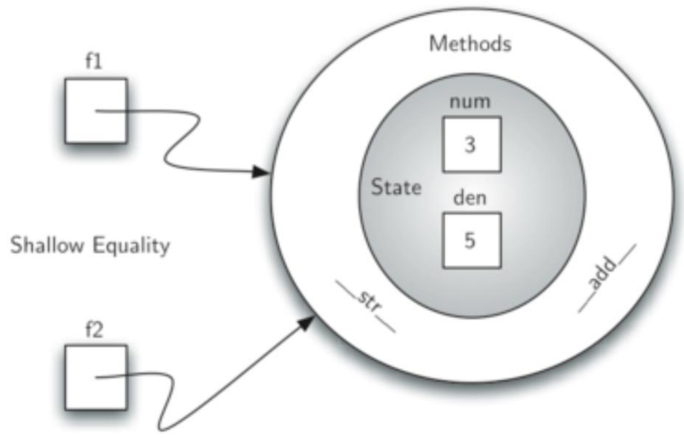
Example:- We can combine the above example and notation to get the fraction summing operation as follows

```
def __add__(self, other_fraction):
    new_num = self.num * other_fraction.den + \
                 self.den * other_fraction.num
    new_den = self.den * other_fraction.den
    common = gcd(new_num, new_den)
    return Fraction(new_num // common, new_den // common)
```

So our code can now work as follows

```
>>> f1 = Fraction(1, 4)
>>> f2 = Fraction(1, 2)
>>> f3 = f1 + f2
>>> print(f3)
3/4
```

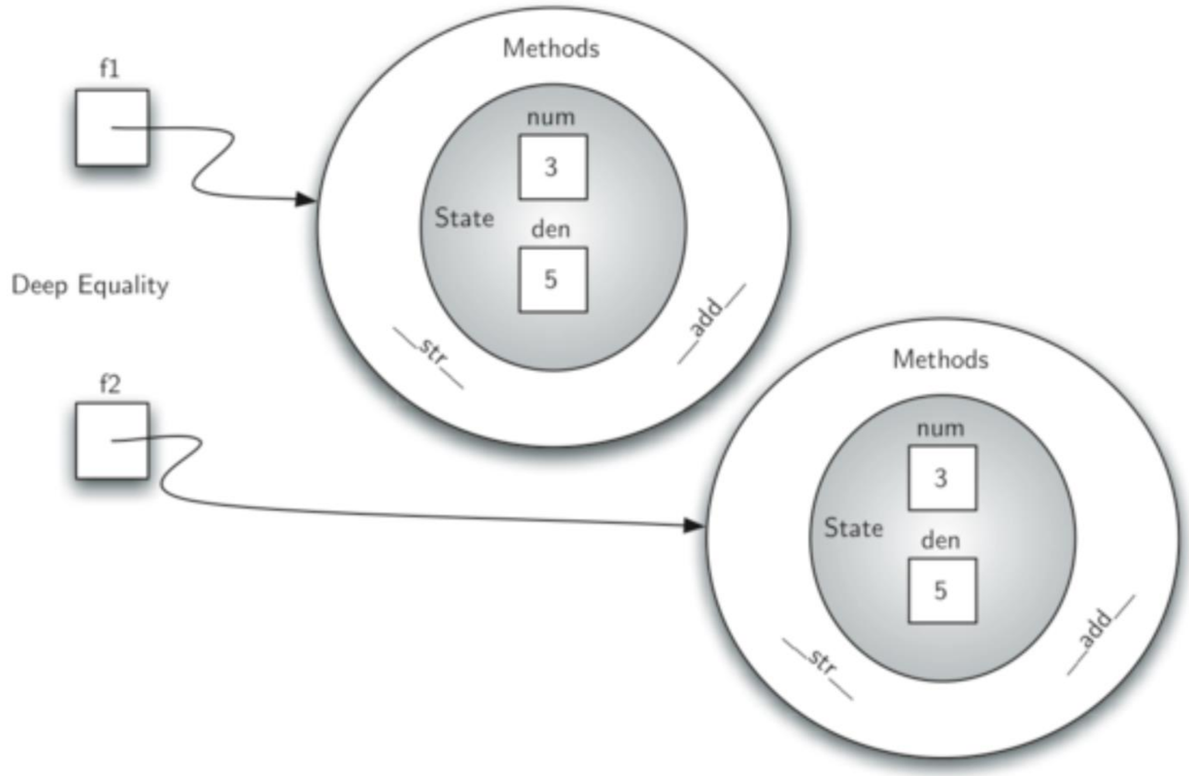
3.9 Shallow equality:- Two different objects with a shared numerator and denominator are <u>not</u> considered equal. The only way two fractions, f1 and f2, are considered equal is they are both references to the same object, this is described as shallow equality.



Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

4.0 Deep equality:- Deep equality is when two fractions, f1 and f2, are equal only by the same value, not the same reference (which is shallow equality). In order to set up deep equality we may override the __eq__ method



4.1 __eq__:- The __eq__ method is a standard method available for any class. The method compares two objects and returns True if their <u>values</u> are the same, otherwise <u>False</u>.

4.2 __le__:- The __le__ method compares two objects and returns true iff the primary value is less than the secondary value.

Example:- Kindly refer to this example of the __eq__ method

```python
def __eq__(self, other_fraction):
    first_num = self.num * other_fraction.den
    second_num = other_fraction.num * self.den

    return first_num == second_num
```

Example:- Combining the above examples, we get

```python
def gcd(m, n):
    while m % n != 0:
        m, n = n, m % n
    return n

class Fraction:
    def __init__(self, top, bottom):
        self.num = top
        self.den = bottom

    def __str__(self):
        return "{:d}/{:d}".format(self.num, self.den)

    def __eq__(self, other_fraction):
        first_num = self.num * other_fraction.den
        second_num = other_fraction.num * self.den

        return first_num == second_num
    def __add__(self, other_fraction):
        new_num = self.num * other_fraction.den \
        + self.den * other_fraction.num
        new_den = self.den * other_fraction.den
        cmmn = gcd(new_num, new_den)
        return Fraction(new_num // cmmn, new_den // cmmn)

    def show(self):
        print("{:d}/{:d}".format(self.num, self.den))

x = Fraction(1, 2)
x.show()
y = Fraction(2, 3)
print(y)
print(x + y)
print(x == y)

```

Reading one – Summary notes

G. Benni K. Evensen [11-01-2021]

4.3 Key terms:- Below is a list of the key terms for chapter 1.

abstract data type (ADT)	abstraction	algorithm
class	computable	data abstraction
data structure	data type	deep equality
dictionary	encapsulation	exception
format operator	formatted strings	f-string
Has-a relationship	implementation-independent	information hiding
inheritance	inheritance hierarchy	interface
Is-a relationship	list	list comprehension
method	method override	mutability
object	procedural abstraction	programming
prompt	`self`	shallow equality
simulation	string	subclass
superclass	truth table	tuple

4.3 Building a proper class:- Follow these guidelines when creating a proper class:

- Each cass should have a docstring to give some documentation
- Each class should have a __str__ string representation
- Each class should have a proper __repr__ method for representation in the interactive shell
- Each class should be comparable so it can be meaningfully sorted.
    - Use __eq__
    - Use __lt__
- You should think about access control for each variable

If your class is a container for other classes then:

- You should be able to find out how many things the container holds using len
- You should be able to iterate over the items in the container
- You may want to allow users to access the items in the container using the square bracket index notation

4.4 Comparison operations:-

`__lt__`	less than `<`
`__gt__`	greater than `>`
`__eq__`	equal to `==`
`__le__`	less than or equal to `<=`
`__ge__`	greater than or equal to `>=`
`__ne__`	not equal to `!=`