Programming with Python

Consolidated Study Material By

Smt. Shreevarshini M R

Selection Grade Lecturer, Computer Science Department,

MEI Polytechnic

Acknowledgements:

The conception of these notes was heavily reliant on the material at
https://www.w3schools.com

Other sources that were referred to:
https://realpython.com
https://docs.python.org

Fundamental Concepts

BRIEF HISTORY

Python was conceived in the late 1980s by Guido van Rossum in the Netherlands as a successor to the ABC programming language. Its implementation began in December 1989 and was finally released in 1991.

FEATURES

- It is a **High Level Programming Language** i.e. the computer instructions are in a way that is more easily understandable and closer to human language.
- It is an **Object Oriented Language** i.e. it relies on the concept of classes and objects. A class is a code template which acts as a blueprint for creating objects (a particular data structure).
- It is **Portable** i.e. it is available on a very wide variety of platforms
- It is **Easy to Learn**. Python has relatively few keywords, simple structure, and a clearly defined syntax which allows the student to pick up the language in a relatively short period of time.
- It is **Easy to Read**. Python does away with most of the distractions of other programming languages like having semi-colon at the end of every line, putting code blocks in curly braces, etc. This makes python code a lot more easy to read than other languages.
- It has a **Large Standard Library**. Python's standard library has many packages and modules with common and important functionality. If you need something that is available in this standard library, you don't need to write it from scratch. Because of this, you can focus on more important things.
- It is **Interpreted**. Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.

APPLICATIONS OF PYTHON

- Web and Internet Development
- Scientific and Numeric Computing
- Artificial Intelligence and Machine Learning
- Software Development
- Education
- Embedded Applications
- Business Applications

PYTHON VERSIONS

- The very first released version that came out in 1991 was Python 0.9.0
- Python 1 launched in 1994 with new features for functional programming like lambda, map, filter and reduce. Python 1.4 brought its next set of features, such as keyword arguments and support for complex number operations.
- Python 2 launched in 2000 with big changes to source code storage. It introduced many desired features like unicode support, list comprehension, and garbage collection. A bit later, Python 2.2 introduced unification of types and classes under one hierarchy, allowing Python to be truly object-oriented.
- Python 3, released in 2008, is the biggest change the language has undergone. The main mission of the update was to remove fundamental design flaws with the language by deleting duplicate modules and constructs. Some of the most notable changes are that `print()` is now a built-in function, removal of the Python 2 `raw_input()`, unification of `str` and `unicode`, and changes in integer division.

PYTHON DISTRIBUTIONS

A distribution of Python is a bundle that contains an implementation of Python along with a bunch of libraries or tools. Examples are CPython, ActivePython, Anaconda, PocketPython, etc.

PYTHON IDE

A Python integrated development environment (IDE) is software for building applications that combines many common developer tools into a single graphical user interface (GUI). An IDE can understand your code much better than a simple text editor. It usually provides features such as build automation, code linting, testing and debugging.

EXECUTION OF PYTHON PROGRAMS

When working with a text editor, the python code that is typed up is first saved in a `.py` file. Then from command-line, the python program can be executed by running the command:

```
python filename.py
```

Where 'filename' is replaced by the name of the file containing the python code.

When working with an IDE, executing the program is as simple as clicking on the run button.

BEST PRACTICES FOR PYTHON PROGRAMMING

- **Having Proper Comments and Documentation**. This becomes very important when you are collaborating with multiple people on a project. Making good code documentation and putting comments explaining the code helps other people know and understand what is in the code.
- **Proper naming of Variables, Classes, Functions and Modules**. Having single letter variable names and heavily abbreviated function or class names makes the code less readable.
- **Writing Modular Code**. Python has many modules and libraries under a repository known as PyPI or the Python Package Index. It contains numerous models written by Python developers, which you can implement in your code just by installing them from the internet. This prevents you from implementing all the logic from scratch and makes your script compact and readable.

INDENTATION

Python uses indentation to indicate a block of code. All lines at the same indent level are considered to be part of the same block. Indentation can be done using some non-zero number of spaces or even using tab spaces.

Python gives an error if indentation is skipped. For example the following code will throw an error.

```
if 5 > 2:
print("Five is greater than two!")
```

Python also gives an error if indents are arbitrarily and unnecessarily put in the code.
For example the following code will throw an error.

```
print("One")
  print("Two")
print("Three")
```

COMMENTS

Python comments are made by starting the comment with # symbol. For example

```
# This is a comment
print("hello")
```

Python comments can also be placed at the end of a line. For Example

```
print("hello") # This is a comment
```

CHARACTER SET

Python supports all ASCII / Unicode characters that include:

- **Alphabets:** All capital (A-Z) and small (a-z) alphabets.
- **Digits:** All digits 0-9.
- **Special Symbols:** Python supports all kind of special symbols like, " ' l ; : ! ~ @ # $ % ^ ` & * ( ) _ + – = { } [ ] \ .
- **White Spaces:** White spaces like tab space, blank space, newline, and carriage return.
- **Other:** All ASCII and UNICODE characters are supported by Python that constitutes the Python character set.

TOKENS

A token is the smallest individual unit in a python program. All statements and instructions in a program are built with tokens. The various tokens in python are:

1. Keywords: Keywords are words that have some special meaning or significance in a programming language. They can't be used as variable names, function names, or any other random purpose. Examples are: True, False, for, while, break, continue, etc.

2. Identifiers: Identifiers are the names given to any variable, function, class, list, methods, etc. for their identification. Here are some rules to name an identifier:

- Python is case-sensitive. So case matters in naming identifiers. Hence name and Name are two different identifiers.
- Identifier starts with a capital letter (A-Z), a small letter (a-z) or an underscore (_). It can't start with any other character.
- Except for letters and underscore, digits can also be a part of identifier but can't be the first character of it.
- Any other special characters or whitespaces are strictly prohibited in an identifier.
- An identifier can't be a keyword.

3. Literals or Values: Literals are the fixed values or data items used in a source code. Python supports different types of literals such as:

- String literals
- Numeric literals
- Boolean literals
- Special Literal - None
- Literals Collections

4. Operators: These are the tokens responsible to perform an operation in an expression. The variables on which operation is applied are called operands. Operators can be unary or binary.

5. Punctuators: These are the symbols that used in python to organise the structures, statements, and expressions. Some of the punctuators are: [ ] ( ) # @ etc.

Basic I/O

OUTPUT

To display output, `print()` is used. For example,
`print("Hello World")`
Will display the output: `Hello World`
The argument for `print()` can be of any datatype, and not necessarily a string.
For example we could also do
`print(5)`
And this would give the output: `5`
`print()` can also take multiple arguments. All the arguments will be printed in the same line separated by a space. For example,
`print(5, "is a number")`
Will display the output: `5 is a number`

Note: `print` works differently in Python 2 and Python 3. In Python 2 `print` is a statement and used like this: `print 5, "is a number"`. Whereas in Python 3 `print()` is a function.

VARIABLES

Variables are containers for storing data values. Unlike some other programming languages, Python has no command for declaring a variable. A variable is created the moment you first assign a value to it. Example:

```
x = 1
y = "hello"
print(x)
print(y)
```

Output:

```
1
hello
```

Variables also need not be declared with any particular type, and can even change type after they have been set. Example:

```
x = 5 # x is of type int
print(x)
x = "world" # x is now of type str
print(x)
```

Output:

```
5
world
```

Python allows you to assign values to multiple variables in one line. Example:

```
x, y, z = "apple", "mango", "banana"
print(x)
print(y)
print(z)
```

Output:

```
apple
mango
banana
```

Note: The number of variables must match the number of values, or else you will get an error.

It is also possible to assign the same value to multiple variables in one line. Example:

```
x = y = z = "orange"
print(x)
print(y)
print(z)
```

Output:

```
orange
orange
orange
```

INPUT

While executing the Python program, it is possible to take input strings from the user.
But this is done differently in Python 2 and Python 3.
In Python 3 this is achieved using the `input()` function. An example is shown below.

```
nm = input("enter your name")
```

In Python 2 string input is taken using the `raw_input()` function. An example is shown below.

```
nm = raw_input("enter your name")
```

In both the above cases the code will display the string `enter your name` and accept input from the user. The input given by the user will be stored in the variable `nm` as a string.

DATATYPES

Python has the following main datatypes:
`str` or string datatype which is the datatype of text data.
`int` for integers
`float` for decimals
`complex` for complex numbers
`bool` for `True`/`False`

To get data type of any object, we can use the `type()` function.
For example,

```
x = 5
print(type(x))
```

Will output: `<class 'int'>`

Type Casting

In python type casting can be done using the functions `str()`, `int()` and `float()`
The above functions convert the argument data to `str`, `int` and `float` respectively.
Here are various examples:
String Conversion

```
x = str(2) # x will be "2"
y = str(4.7) # y will be "4.7"
```

Integer Conversion

```
x = int(3.6) # x will be 3
y = int("5") # y will be 5
```

Float Conversion

```
x = float(3) # x will be 3.0
y = float("5.2") # y will be 5.2
```

OPERATORS

Arithmetic Operators

Operator	Name	Example
+	Addition	5+2 gives 7
–	Subtraction	5–2 gives 3
*	Multiplication	5*2 gives 10
/	Division	5/2 gives 2.5
%	Modulus	5%2 gives 1
**	Exponent	5**2 gives 25
//	Floor Division	5//2 gives 2

Comparison/Relational Operators

Operator	Name	Example
==	Equal	5==5 gives True
!=	Not equal	5!=5 gives False
>	Greater than	5>5 gives False
<	Lesser than	5<7 gives True
>=	Greater than or equal to	5>=5 gives True
<=	Lesser than or equal to	5<=2 gives False

Logical/Boolean Operators

Operator	Description	Example
and	Return True if both statements are true	5>2 and 5<7 gives True
or	Return True if at least one of the statements is true	5>2 or 5<3 gives True
not	Reverses the result i.e. Returns True if the result is false and vice versa	not(5>2 and 5<7) gives False

Bitwise operators

Operator	Name	Description
&	AND	Sets output bit to 1 if both input bits are 1
|	OR	Sets output bit to 1 if at least one of the two input bits is 1
^	XOR	Sets output bit to 1 if exactly one of the two input bits is 1
~	NOT	Inverts all bits
<<	Left Shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Right Shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

String Operators

Operator	Description	Example
+	Concatenates the two strings	"ab"+"cd" gives "abcd"
*	Creates multiple copies of a string	"ab"*3 gives "ababab"
[]	Return a character from an index	"abcde"[2] gives "c"
[:]	Returns a substring between two indices	"abcde"[1:3] gives "bc"
in	Returns True if a given substring exists in the string	"ab" in "abcd" gives True
not in	Returns False if a given substring exists in the string	"ef" not in "abcd" gives True

Operator Precedence

Precedence	Operator	Name
1	()	Paranthesis
2	**	Exponentiation
3	+a, -a, ~a	Unary plus, minus, complement
4	/ * // %	Divide, Multiply, Floor Division, Modulo
5	+ -	Addition, Subtraction
6	>> <<	Shift Operators
7	&	Bitwise AND
8	^	Bitwise XOR
9	|	Bitwise OR
10	>= <= > <	Comparison Operators
11	!= ==	Equality Operators
12	+= -= /= *= =	Assignment Operators
13	is, is not, in, not in	Identity and Membership Operators
14	and, or, not	Logical Operators

The above list is from highest to lowest precedence. Operators having the same precedence will be executed from left to right in the order that they appear. For example, the expression 5/2*3/4 will be evaluated as (((5/2)*3)/4)

Control Flow: Conditional Statements

IF

The `if` statement is used to execute a block of code if a given condition is `True`.
An example usage of `if` is show below:

```
if 7>5:
   print("hello world")
```

The above code will output `hello world` as the condition `7>5` is `True`.

In general, `if` is followed by any expression that is evaluated to a boolean value, and the indented block of code just below the `if` statement is executed if the expression evaluates to `True`.

Python will throw an error if there isn't an indented block of code below the `if` statement.
For Example, the below code will throw an error:

```
if 7>5:
print("hello world")
```

ELIF

The `elif` statement is used after an `if` statement to check a new condition if the previous condition was `False`.
An example usage of `elif` is shown below:

```
if 1>2:
   print("hello")
elif 5==5:
   print("bye")
```

The above code will output `bye` as the condition `1>2` is `False`, but the condition `5==5` is `True`.

`elif` can be used multiple times if needed. For example:

```
if 1>2:
   print("One")
elif 2>5:
   print("Two")
elif 5>4:
   print("Three")
```

The above code will output: `Three`

ELSE

This is used to execute a block of code if none of the conditions in the previous `if` and `elif` statements were `True`.
An example is shown below:

```
if 1>4:
   print("One")
elif 5>9:
   print("Two")
else:
   print("Three")
```

The above code will output: `Three`

SHORT HAND IF

If you have only one statement to execute, you can put it on the same line as the `if` statement.
For example:

```
if 1==1: print("hello")
```

SHORT HAND IF ... ELSE

If you have only one statement to execute for `if` and one for `else`, you can put it all in the same line as shown below:

```
print("hello") if 1>2 else print("bye")
```

The above code will output `bye` as the condition `1>2` is `False`.

The short hand `if` ... `else` is basically a ternary operator that takes an expression, a condition, and another expressions; and returns one of the two expressions depending on whether the condition is `True` or `False`. So it can also be used as shown below:

```
a = 1 if 1>2 else 2
```

Or even like this

```
print("hello" if 1>2 else "bye")
```

NESTED IF

If an `if` statement is placed inside the execute block of another `if`, `elif` or `else` statement, it is called nested `if`. Example:

```
if 2>1:
   if 3>2:
      print("hello")
```

Output: `hello`

Control Flow: Loops

WHILE LOOP

The `while` loop is used to execute a block of code again and again as long as a given condition is `True`. An example is shown below:

```
a=0
while a<5:
    a=a+1
    print(a)
```

The above code will give the output:

```
1
2
3
4
5
```

FOR LOOP

In python `for` loops iterate through any given sequence, which could be a list, tuple, set, dictionary or even a string. An example is shown below:

```
for word in ["apple", "banana", "mango", "orange"]:
    print(word)
```

The above code will output:

```
apple
banana
mango
orange
```

Here again indentation is important, and the indented lines will be the block that is executed again and agin by the `while` or `for` loop.

The range() Function

If we want to iterate through a sequence of numbers in a `for` loop, a handy way to generate the sequence is to use the `range()` function. An example of it's use is shown below:

```
for i in range(1,4):
    print(i)
```

The above code will output:

```
1
2
3
```

We could also use `range(n)` with a single integer argument `n`. That would create a sequence with `n` consecutive integers starting from `0`. An example is shown below:

```
for i in range(3):
    print(i)
```

The above code will output:

```
0
1
2
```

We could also specify the increment (default is 1) in the sequence generated by `range()` by adding a third argument. An example is shown below:

```
for i in range(1,10,3):
   print(i)
```

The above code will output:

```
1
4
7
```

ELSE BLOCK

The `else` keyword when used with a `for` or `while` loop specifies a block of code to be executed when the loop is finished.

Example:

```
for i in range(2,5):
   print(i)
else:
   print("done")
```

The above code will output:

```
2
3
4
done
```

Another Example:

```
i=1
while i<4:
   print(i)
   i=i+1
else:
   print("done")
```

The above code will output:

```
1
2
3
done
```

CONTROLLING LOOP EXECUTION

Break

When used inside a loop the `break` statement stops the execution of the loop statements and exits the loop. An example is shown below:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
   print(x)
   if x == "banana":
      break
```

The above code will output:

```
apple
banana
```

Continue

With the `continue` statement we can stop the current iteration of the loop, and continue with the next. An example is shown below:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
   if x == "banana":
      continue
   print(x)
```

The above code will output:

```
apple
cherry
```

NESTED LOOPS

When a `for` or `while` loop is placed inside another `for` or `while` loop, it is called nested looping.

Example:

```
for i in range(1, 6):
   out = ""
   for j in range(1, i+1):
      out = out + str(j)
   print(out)
```

Output:

```
1
12
123
1234
12345
```

Another Example:

```
for i in range(5, 0, -1):
   out = ""
   for j in range(1, i+1):
      out = out + str(j)
   print(out)
```

Output:

```
12345
1234
123
12
1
```

An example with nested `while` loop:

```
i = 2
while i < 100:
   j = 2
   while j <= (i/j):
      if i % j == 0: break
      j = j + 1
   if j > i/j: print(i)
   i = i + 1
```

Output:

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

The above code has printed all prime numbers less than 100.

Data Collections

Python has four collection data types: Lists, Tuples, Sets and Dictionaries. Of these Tuples and Sets are immutable data collections i.e. the items of these collections are not changeable.

TUPLES

A tuple is a collection which is ordered, unchangeable, and allows duplicate members. Unlike sets, lists and dictionaries, it is not possible to add elements to or remove elements from a tuple.

Tuples are created as follows using `()`

```
tpl = (1,2,2,3)
```

Or

```
tpl = ("apple", "banana", "orange")
```

Or even

```
tpl = ("apple", "banana", 3, 4)
```

We could also have tuples inside a tuple (nested tuples) as follows:

```
tpl = ((1,2),(2,3))
```

Or

```
tpl = ((1,2),2)
```

Another way to create tuples is by using the `tuple()` constructor. This can be used to create tuples from lists, sets, dictionaries or strings. Here's an example:

```
s = {1,2,3} # This is a set
l = [1,2,3] # This is a list
d = {"brand": "Ford", "model": "Mustang"} # This is a dictionary
# sets, lists, and dictionaries will be covered in detail later
st = "hello"
print(tuple(s))
# Set is fundamentally unordered
# So the tuple can be created in any order of the set items
print(tuple(l))
print(tuple(d))
print(tuple(st))
```

The above code will output:

```
(2, 3, 1)
(1, 2, 3)
('brand', 'model')
('h', 'e', 'l', 'l', 'o')
```

The length of a tuple can be obtained using `len()`. An example is shown below:

```
tpl = (1,2,2,3)
print(len(tpl))
```

The above code will give the output: `4`

It is possible to use the membership operator `in` to check if an item is in a tuple.
For example, `2 in (1,2,3)` will evaluate to `True`.

Accessing Tuple Items

Since tuple items are ordered, they can be accessed by their index. The first item has index 0 and the consecutive items have index 1, 2, 3 so on. It is also possible to use negative indices to access tuple items. The index -1 corresponds to the last element, -2 corresponds to the second last element and so on. Below is an example showing this.

```
tpl = ("apple", "banana", "mango", "orange")
print(tpl[0])
print(tpl[1])
print(tpl[-1])
print(tpl[-2])
```

The above code will output:

```
apple
banana
orange
mango
```

We can also take out slices of a tuple and get a new tuple by specifying where to start and end the slice. Below is an example showing this.

```
all_fruits = ("apple", "banana", "cherry", "kiwi", "mango", "orange")
some_fruits = all_fruits[2:5]
print(some_fruits)
```

In the above code, the range will start at index 2 (included) and end at index 5 (not included). The output of the code will be:

```
('cherry', 'kiwi', 'mango')
```

If we leave out the start value, the range will start at the first item and go till the given index (not included). For example,

```
("apple", "banana", "mango", "orange")[:2]
```

Will give the tuple `("apple", "banana")`

If we leave out the end value, the range will start at the given index (included) and go on till the end of the tuple. For example,

```
("apple", "banana", "mango", "orange")[2:]
```

Will give the tuple `("mango", "orange")`

We could also give negative indices as the start and end values. For example,

```
("apple", "banana", "mango", "orange")[-3:-1]
```

Will give the tuple `("banana", "mango")`

If the start index comes after the end index in a specified range, then the empty tuple is returned. For example, the following statements will give the empty tuple `()`

```
("apple", "banana", "mango", "orange")[3:2]
("apple", "banana", "mango", "orange")[-1:1]
```

It is also possible to loop through tuples using a `for` loop as follows:

```
for city in ("Bangalore", "Mysore", "Chennai")
   print(city)
```

The above code will output:

```
Bangalore
Mysore
Chennai
```

Unpacking Tuples

It is possible to unpack tuples as follows:

```
fruits = ("apple", "banana", "orange")
(fruit1, fruit2, fruit3) = fruits
# Equivalently, we could also unpack without the brackets () as below
# fruit1, fruit2, fruit3 = fruits
print(fruit1)
print(fruit2)
print(fruit3)
```

The above code will output:

```
apple
banana
orange
```

Tuple Methods

Python has two inbuilt functions that can be used on tuples. They are `count()` and `index()`.
`count()` returns the number of times a given argument occurs in a tuple. For example,
`(1,3,2,5,3,4).count(3)`
Will return `2`
`index()` returns the index of the first occurrence of a given argument in a tuple. For example,
`(1,3,2,5,3,4).index(3)`
Will return `1`

SETS

A set is a collection which is unordered, unchangeable (i.e. it is not possible to edit set elements) and disallows duplicate members. However, unlike tuples, it is possible to add elements to or remove elements from a set.

Sets are created as follows using `{}`

```
s = {1,2,3}
```

Or

```
s = {"apple", "banana", "orange"}
```

Or even

```
s = {"apple", "banana", 3, 4}
```

If duplicates are included while defining a set, they are considered only once.
Consider the below example:

```
s1 = {2,3,2,4,3,1,2,3}
s2 = {1,2,3,4}
print(s1==s2)
```

The above code will give the output: `True`

We could also have sets inside a set as follows:

```
s = {{1,2},{2,3}}
```

Or

```
s = {{1,2},2}
```

Note: The empty set is not `{}`, it is `set()` i.e. the `set()` constructor (explained next) acting on nothing. `{}` is an empty dictionary (which is dealt with in greater detail later).
The statement `type({})` will output: `<class 'dict'>`
But the statement `type(set())` will output: `<class 'set'>`

A set could also be created from tuples, lists, dictionaries and strings using the `set()` constructor. Here's an example:

```
t = (1,2,3) # tuple
l = [1,2,3] # list
d = {"brand": "Ford", "model": "Mustang"} # dictionary
s = "hello"
print(set(t))
print(set(l))
print(set(d))
print(set(s))
```

The above code will output:

```
{3, 2, 1}
{2, 3, 1}
{'model', 'brand'}
{'h', 'l', 'o', 'e'}
```

The length of a set can be obtained using `len()` function. An example is shown below:

```
s = {1,2,3,4}
print(len(s))
```

The above code will output: `4`

We can check if a specified item is in a set using the `in` operator. Below is an example:

```
s = {"apple", "orange", "banana"}
print("orange" in s)
```

The above code will output: `True`

Accessing Set Items

Since sets are unordered, their elements won't have any indices. Hence it is not possible to access set elements using an index like in tuples.

However it is possible to loop through the elements of a set using `for` loop.
An example is shown below:

```
for city in {"Bangalore", "Mysore", "Chennai"}
   print(city)
```

The above code will output:

```
Bangalore
Chennai
Mysore
```

Again due to the lack of a defined order for a set, every time the code is executed, the cities can be printed in a different order.

Adding and Removing Set Items

Items can be added to a set using the inbuilt method `add()`. Below is an example:

```
s = {"apple", "orange", "banana"}
s.add("mango")
print(s)
```

The above code will give the output: `{'apple', 'mango', 'banana', 'orange'}`

Items can be removed from a set using the inbuilt method `remove()`. Below is an example:

```
s = {"apple", "orange", "banana"}
s.remove("banana")
print(s)
```

The above code will give the output: `{'apple', 'orange'}`

Note: An alternate way to remove elements from a set is to use the `discard()` method. It works the same way as `remove()` with the difference being that `remove()` throws an error if the item to be removed does not exist in the set, however `discard()` won't throw an error in such a case.

More Set Methods

The union of two sets can be taken using the inbuilt function `union()`. Below is an example:

```
x = {"apple", "orange", "melon"}
y = {"melon", "mango", "kiwi"}
z = x.union(y)
print(z)
```

The above code will give the output: `{'apple', 'orange', 'melon', 'mango', 'kiwi'}`

The intersection of two sets can be taken using the inbuilt method `intersection()`
Below is an example:

```
x = {"apple", "orange", "banana", "mango"}
y = {"banana", "mango", "strawberry"}
z = x.intersection(y)
print(z)
```

The above code will give the output: `{'banana', 'mango'}`

The difference between two sets can be obtained using the inbuilt method `difference()`
Below is an example:

```
x = {"apple", "orange", "cherry", "banana"}
y = {"cherry", "banana", "melon"}
z = x.difference(y)
print(z)
```

The above code will give the output: `{'apple', 'orange'}`

LIST

A list is a collection which is ordered, changeable, and allows duplicate members. It is also possible to add elements to or remove elements from a list.

Lists are created as follows using `[]`

```
l = [1,2,2,3]
```

Or

```
l = ["apple", "banana", "orange"]
```

Or even

```
l = ["apple", "banana", 3, 4]
```

We could also have lists inside a list (nested lists) as follows:

```
l = [[1,2],[2,3]]
```

Or

```
l = [[1,2],2]
```

Another way to create lists is by using the `list()` constructor. This can be used to create lists from tuples, sets, dictionaries or strings. Here's an example:

```
t = (1,2,3) # tuple
s = {1,2,3} # set
d = {"brand": "Ford", "model": "Mustang"} # dictionary
st = "hello"
print(list(t))
print(list(s))
print(list(d))
print(list(st))
```

The above code will output:
```
[1, 2, 3]
[2, 3, 1]
['brand', 'model']
['h', 'e', 'l', 'l', 'o']
```

The length of a list can be obtained using `len()`. An example is shown below:
```
l = [1,2,2,3]
print(len(l))
```
The above code will give the output: `4`

To check if an item is in a list, we can use the membership operator `in`.
For example, `2 in [1,2,3]` will evaluate to `True`.

Accessing List Items

List items are ordered and they can be accessed by their index. The first item has index 0 and the consecutive items have index 1, 2, 3 so on. It is also possible to use negative indices to access list items. The index -1 corresponds to the last element, -2 corresponds to the second last element and so on. Below is an example showing this.
```
l = ["apple", "banana", "mango", "orange"]
print(l[2])
print(l[-1])
print(l[-3])
```
The above code will output:
```
mango
orange
banana
```

We can also take out slices of a list and get a new list by specifying where to start and end the slice. Below is an example showing this.
```
all_fruits = ["apple", "banana", "cherry", "kiwi", "mango", "orange"]
some_fruits = all_fruits[2:5]
print(some_fruits)
```
In the above code, the range will start at index 2 (included) and end at index 5 (not included).
The output of the code will be:
```
['cherry', 'kiwi', 'mango']
```

If we leave out the start value, the range will start at the first item and go till the given index (not included). For example,
```
["apple", "banana", "mango", "orange"][:2]
```
Will give the list `["apple", "banana"]`

If we leave out the end value, the range will start at the given index (included) and go on till the end of the list. For example,
```
["apple", "banana", "mango", "orange"][2:]
```
Will give the list `["mango", "orange"]`

We could also give negative indices as the start and end values. For example,
```
["apple", "banana", "mango", "orange"][-3:-1]
```
Will give the list `("banana", "mango")`

If the start index comes after the end index in a specified range, then the empty list is returned. For example, the following statements will give the empty list `[]`
```
["apple", "banana", "mango", "orange"][3:2]
["apple", "banana", "mango", "orange"][-1:1]
```

It is also possible to loop through lists using a `for` loop as follows:

```
for city in ["Bangalore", "Mysore", "Chennai"]
   print(city)
```

The above code will output:

```
Bangalore
Mysore
Chennai
```

Unpacking Lists

It is possible to unpack lists as follows:

```
fruits = ["apple", "banana", "orange"]
[fruit1, fruit2, fruit3] = fruits
# Equivalently, we could also unpack without the brackets [] as below
# fruit1, fruit2, fruit3 = fruits
print(fruit1)
print(fruit2)
print(fruit3)
```

The above code will output:

```
apple
banana
orange
```

Changing List Items

It is possible to change the list item at a particular index as follows:

```
l = [1,2,3]
l[1] = 5
print(l)
```

The above code will output: `[1, 5, 3]`

We could also change a range of list items as follows:

```
l = [1,2,3,4,5,6]
l[2:5] = [7,8,9]
print(l)
```

The above code will output: `[1, 2, 7, 8, 9, 6]`

While replacing a range of list with a new list as above, it is not necessary for the new list to have the same length as the length of the specified range. Consider the below example:

```
l1 = [1,2,3,4,5,6]
l2 = [1,2,3,4,5,6]
l1[2:5] = [7,8]
l2[1:3] = [7,8,9]
print(l1)
print(l2)
```

The above code will output:

```
[1, 2, 7, 8, 6]
[1, 7, 8, 9, 4, 5, 6]
```

We could insert an item to a list at a specified index, without altering the already existing list items by using the inbuilt method `insert()`. Below is an example:

```
l = ["apple", "banana", "mango"]
l.insert(2, "orange")
print(l)
```

The above code will output: `['apple', 'banana', 'orange', 'mango']`

To add an item at the end of a list we could use the inbuilt method `append()`
Below is an example:

```
l = ["apple", "banana", "cherry"]
l.append("watermelon")
print(l)
```

The above code will output: `['apple', 'banana', 'cherry', 'watermelon']`

We can also append a list at the end of a list using `extend()`. Below is an example:

```
l1 = ["apple", "banana", "mango"]
l2 = ["orange", "watermelon", "cherry"]
l1.extend(l2)
print(l1)
```

The above will give the output:

```
['apple', 'banana', 'mango', 'orange', 'watermelon', 'cherry']
```

Note: The argument of `extend()` need not necessarily be a list, it could be any data collection, so a tuple, set, dictionary or even a string would work. Below is an example with a tuple.

```
l1 = [1,2,3]
l2 = (4,5,6)
l1.extend(l2)
print(l1)
```

The above will give the output: `[1, 2, 3, 4, 5, 6]`

To remove a specific item from a list we could use the inbuilt method `remove()`
Below is an example:

```
l = ["apple", "banana", "cherry"]
l.remove("banana")
print(l)
```

The above code will output: `['apple', 'cherry']`

To remove an item from a specific index we could use the inbuilt method `pop()`
Below is an example:

```
l = ["apple", "banana", "cherry", "mango"]
l.pop(2)
print(l)
```

The above code will output: `['apple', 'banana', 'mango']`

When used without any arguments, `pop()` will remove the last item from the list.
Below is an example:

```
l = ["apple", "banana", "cherry", "mango"]
l.pop()
print(l)
```

The above code will output: `['apple', 'banana', 'cherry']`

Note: The item removed using `pop()` can be accessed as shown below.

```
l = ["apple", "banana", "cherry", "mango"]
a = l.pop(2)
print(l)
print(a)
```

The above code will give the output:

```
['apple', 'banana', 'mango']
'cherry'
```

To remove all items from a list, we could use the inbuilt method `clear()`
Below is an example:

```
l = [1,2,3]
l.clear()
print(l)
```

The above code will output: `[]`

More List Methods

`count()`
This is used to the count the number of times a given argument occurs in a list.
Example:

```
l = [1,3,2,4,3,5]
print(l.count(3))
```

Output: 2

`index()`
This is used to get the index of the first occurrence of a given argument in a list.
Example:

```
l = [1,3,2,4,3,5]
print(l.index(3))
```

Output: 1

`reverse()`
This can be used to reverse a list
Example:

```
l = [1,3,2,4,3,5]
l.reverse()
print(l)
```

Output: `[5, 3, 4, 2, 3, 1]`

`sort()`
This can be used to sort a list in ascending order.
Example:

```
l = [1,3,2,4,3,5]
l.sort()
print(l)
```

Output: `[1, 2, 3, 3, 4, 5]`

DICTIONARY

A dictionary is a collection which is ordered*, changeable and does not allow duplicates. It is possible to add items to or remove items from a dictionary. Dictionaries hold data in key:value pairs, and the values can be referred to by using the key name.

*as of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Here's an example creation of a dictionary:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
```

Here' another example:
```
car = {
   "brand": "Ford",
   "electric": False,
   "year": 1964,
   "colours": ["red", "white", "blue"]
}
```

One more example:
```
d = {1:3, 4:2, 3:7}
```

As mentioned before, a dictionary does not allow duplicates. So there can't be more than one value for the same key (it is however possible to have the same value for different keys). If there are duplicates in the definition of a dictionary, then the key:value pair coming after will overwrite the key:value pair coming before. Below is an example of duplicates in a dictionary definition:
```
d = {1:3, 2:5, 2:4, 7:8}
print(d)
```
The above code will output: `{1: 3, 2: 4, 7: 8}`

The length of a dictionary can be obtained using `len()`. Below is an example:
```
d = {1:3, 2:5, 4:9, 7:8}
print(len(d))
```
The above code will output: `4`

The membership operator `in` will tell if a particular key is present in a dictionary.
For example, `"ab" in {"ab": "cd", "ef"; "gh"}` will give `True`.

Accessing Dictionary Items

As mentioned before, a dictionary holds data in key:value pairs. The value can be accessed by specifying the key. Below is an example:
```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
print(car["model"])
```
The above code will output: `'Mustang'`

It is also possible to loop though the keys of a dictionary using a `for` loop as shown below:
```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
for key in car:
   print("Key: " + key + ", " + "Value: " + str(car[key]))
# String concatenation using + requires two strings
# But the value 1964 of the key 'year' is an integer
# Hence type casting is done using str()
```
The above code will give the output:
```
Key: brand, Value: Ford
Key: model, Value: Mustang
Key: year, Value: 1964
```

Changing Dictionary Items

The value of a key:value pair can be changed by referring to the key. Below is an example:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car["year"] = 2021
print(car["year"])
```

The above code will output: `2021`

If the key that is being referred to while assigning a value does not exist in the dictionary, then a new key:value pair will be created in the dictionary. Below is an example:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car["electric"] = False
print(car)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'electric': False}
```

It is also possible to update an old dictionary with a new one using the `update()` method. If the new dictionary contains some keys that are also in the the old one, then the values corresponding to the intersecting keys of the old dictionary will be updated to the values in the new dictionary. Below is an example use of `update()`

```
car1 = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car2 = {
   "electric": False,
   "year": 2021,
   "colours": ["red", "white", "blue"]
}
car1.update(car2)
print(car1)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2021, 'electric': False,
'colours': ['red', 'white', 'blue']}
```

A specific key:value pair can be removed from the dictionary using `pop()`
Example:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car.pop("year")
print(car)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang'}
```

Note: `pop()` also returns the value of the key:value pair that is popped. See the code below:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
a = car.pop("year")
print(car)
print(a)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang'}
1964
```

The last item* in a dictionary can be removed using `popitem()`
Example:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car.popitem()
print(car)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang'}
```

*In Python 3.6 and earlier, dictionary is not ordered. Therefore in those versions, a random item is removed by `popitem()`

Note: `popitem()` also returns the key:value pair that is popped as a tuple. See the code below:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
a = car.popitem()
print(car)
print(a)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang'}
('year', 1964)
```

To remove all items from a dictionary, we could use the inbuilt method `clear()`
Below is an example:

```
d = {1:7, 2:5, 3:8}
d.clear()
print(d)
```

The above code will output: `{}`

STRINGS

Like many other popular programming languages, strings in Python are arrays of characters.
(An array is basically an ordered collection of elements of the same type.)
However, Python does not have a character data type. A single character is simply a string of length 1. But being array-like, many of the thing that we do with other iterables in Python like lists, tuples, sets can be done on strings too, as we will see in this section.

A string can be created by characters within single-quotation (') or double-quotation (") marks. For most of these notes, we have used double-quotation (") marks to represent strings. So from this section onwards we'll use single-quotation (') marks to represent strings.

The length of a string can obtained using `len()`. An example is shown below:

```
a = 'Python'
print(len(a))
```

The above code will output: `6`

The membership operator `in` can be used to check if a particular character or even a word is inside a string. For example,
`'a' in 'apple'` will return `True`,
`'app' in 'apple'` will return `True`,
`'b' in 'apple'` will return `False`

Just like lists and tuples, characters in a string can be accessed from an index. Example:

```
a = 'hello'
print(a[1])
```

The above code will output: `e`

We could also take out slices of a string like in lists and tuples. Example:

```
a = 'watermelon'
print(a[0:-5])
```

The above code will output: `water`

We can also loop though string characters using a `for` loop. Example:

```
for letter in ['bye']
   print(letter)
```

The above code will output:

```
b
y
e
```

String Methods

`upper()`
Converts all characters in string to upper case.
Example:

```
s = 'Python'
print(s.upper())
```

Output: `PYTHON`

`lower()`
Converts all characters in string to lower case.
Example:

```
s = 'Python'
print(s.lower())
```

Output: `python`

`strip()`
Removes all white spaces from before and after the text inside a string.
Example:
```
s = '   apple '
print(s == 'apple')
print(s.strip() == 'apple')
```
Output:
```
False
True
```

`replace()`
Replaces a substring with another string.
Example:
```
s = 'word'
s.replace('rd', 'rld')
print(s)
```
Output: `world`

Note: `replace()` won't modify the string if the given substring to be replaced is not present inside the string.

`split()`
Returns a list by splitting the string at the specified separator
Example:
```
s = 'a, b, c, d'
l = s.split(', ')
print(l)
```
Output: `['a', 'b', 'c', 'd']`

Note: If there is no argument given for `split()`, then the string will be split at whitespaces.
Example:
```
s = 'a b c d'
print(s.split())
```
Output: `['a', 'b', 'c', 'd']`

`count()`
Counts the number of times a specified character appear in a string.
Example:
```
s = 'apple'
print(s.count('p'))
```
Output: `2`

`index()`
Returns the index of the first appearance of a specified character.
Example:
```
s = 'apple'
print(s.index('p'))
```
Output: `1`

Formatting Strings

1. Using %

This is very similar to the `printf()` style formatting in C
Here's an example:

```
"My name is %s" %nm
```

In the above code, `%s` is replaced by the string stored in the variable `nm`
Consider the following code:

```
nm = "Suresh"
out = "My name is %s" %nm
print(out)
```

The above code would output: `My name is Suresh`

We could also do this with numeric literals.
We use `%d` for integers and `%f` for floating point numbers.
For floating point numbers, we can control the number of decimal places that are displayed by writing `%.<number of digits>f`. Here's an example:

```
"pi = %.3f" %3.14159
```

The above code will round off `3.14159` to 3 decimal places and display the output: `pi = 3.142`

Multiple substitutions can also be done as follows:

```
"There are %d students in %s college" %(num, name)
```

The above code will replace `%d` with the integer in `num`, and `%s` with the string in `name`

2. Using `format()`

This is a newer formatting method that is usually more preferred.
Here is an example of this style of formatting:

```
s = "My name is {}"
nm = "Suresh"
print(s.format(nm))
```

The above code will display the output: `My name is Suresh`

As can be seen above, in this formatting method, `{}` in a string is replaced by the argument of the `format()` function.

Here's another example:

```
pi = 3.1415
print("pi = {}".format(pi))
```

The above code will display the output: `pi = 3.1415`

So unlike the `%` method, here just `{}` works for both string and numeric literals.

Multiple substitutions can be done as follows:

```
"There are {} {} and {} {} in the basket".format(5,"apples",4,"oranges")
```

The above string will be formatted as:

```
"There are 5 apples and 4 oranges in the basket"
```

We can include indices in the parentheses to specify which argument is used for substitution.
Here's an example:

```
line = "There are {2} {0} and {2} {1} in the basket"
print(line.format("apples", "oranges", 5, 4))
```

The above code will output: `There are 5 apples and 5 oranges in the basket`

COMPREHENSIONS IN PYTHON

Comprehensions are constructs that allow sequences to be built from other sequences. Python 2 introduced the concept of list comprehensions, and Python 3 took it further by including set and dictionary comprehensions. Below are some examples of comprehensions.

List Comprehension

Suppose we want to take a list of numbers and create a list of their squares, we could do it with a for loop as shown below:

```
l = [1,4,2,5]
ls = []
for i in l:
   ls.append(i*i)
print(ls)
```

Output: `[1, 16, 4, 25]`

Another way to do this would be to use list comprehension as shown below:

```
l = [1,4,2,5]
ls = [x*x for x in l]
print(ls)
```

Output: `[1, 16, 4, 25]`

We could also include a condition in the comprehension to include only selected members from the list. Example:

```
l = [1,4,2,5]
ls = [x*x for x in l if x%2==1]
print(ls)
```

Output: `[1, 25]`

In the above code only odd numbers in `l` are taken and squared to produce `ls`.

List comprehensions can be used to create lists from any iterable (like a list, tuple, set, dictionary or string). The iterable need not necessarily be a list. Example:

```
s = 'hello'
l = [x for x in s]
print(l)
```

Output: `['h', 'e', 'l', 'l', 'o']`

In the above code we are creating a list from a string.

Set Comprehensions

This is very similar to list comprehensions, except that is used to produce a set from a given iterable (which could be a list, tuple, set, dictionary or string). Example:

```
fruits = ['apple', 'banana', 'cherry', 'kiwi', 'melon', 'mango']
s = {x for x in fruits if 'a' in x}
print(s)
```

Output: `{'banana', 'apple', 'mango'}`

In the above code, only the strings in `fruits` that contain the letter `'a'` will be selected and included in the set `s`.

Another Example:

```
word = 'hello'
consonants = {x for x in word if x not in {'a', 'e', 'i', 'o', 'u'}}
print(consonants)
```

Output: `{'h', 'l'}`

In the above code, a set is made by including only the consonants in the string `word`.

Dictionary Comprehensions

Again very similar to list and set comprehensions, except that it is used to produce a dictionary from a given iterable. Example:

```
d = {'a': 0, 'b': 1, 'c': 2, 'd': 3}
nd = {x:d[x]+1 for x in d}
print(nd)
```

Output: `{'a': 1, 'b': 2, 'c': 3, 'd': 4}`

Another Example:

```
word = 'hello'
d = {x:word.count(x) for x in word}
print(d)
```

Output: `{'h': 1, 'e': 1, 'l': 2, 'o': 1}`

General Syntax

Comprehensions are Python's way of implementing the notation for sets as used in mathematics. Suppose we have a set I containing positive integers and want to define a new set S containing the squares of only those integers that are odd. The usual mathematical way for defining this would be:

$S = \{x^2 \mid x \in I \text{ and } x \text{ is odd}\}$

Or in general,

$S = \{expression \mid item \in domain \text{ and } condition\}$

Python's way of implementing this notation is:

expression `for` *item* `in` *iterable* `if` *condition*

Which is the general syntax for comprehensions in python.

If the above syntax is enclosed within `[]`, then it becomes a list comprehension. If it is enclosed within `{}`, then it becomes a set or dictionary comprehension depending on the *expression*. If it is enclosed within `()`, then a generator object is created. (A generator object is also an iterable.) Generator objects will be dealt with in greater detail in the next section on functions.

Functions

A function is a block of code which only runs when it is called.
You can pass data, known as arguments, into a function.
A function can return data as a result.

DEFINING A FUNCTION

In Python a function is defined using the `def` keyword. The indented block of code will be executed when the function is called. Example:

```
def func():
    print('hello world')
```

A function can take some arguments that can be used in the block of code that is executed. Example:

```
def add(a, b):
    print(a+b)
```

A function can be made to return values using the `return` keyword. Example:

```
def add(a, b):
    return a+b
```

The `return` keyword is followed by an expression. This expression will be evaluated and returned by the function.

CALLING A FUNCTION

A function can be called using its name. Example:

```
def func():
    print('hello world')
func() # The function is called
```

Output: `hello world`

Another Example:

```
def add(a, b):
    print(a+b)
add(4, 5) # The function is called with some arguments
```

Output: `9`

One more example:

```
def add(a, b):
    return a+b
c = add(1, 2)
# The function is called and the returned value is stored in a variable
print(c)
```

Output: `3`

MORE ON FUNCTION ARGUMENTS

Arbitrary Arguments

In our functions definitions before, the number of arguments of the function was fixed. However it is also possible to define a function so that any number of arguments can be passed to it. This is done by adding ∗ before the argument name in the function definition. This way the function will receive a **tuple** of arguments, and can access the items accordingly.
Example:

```
def func(*args):
   for x in args:
      print(x)
func(1)
func(4,5)
```

Output:

```
1
4
5
```

Keyword Arguments

Arguments can also be sent to the function with the *key = value* syntax. This way the order of the arguments does not matter. The arguments sent this way are called keyword arguments. Example:

```
def func(a, b):
   print(a)
   print(b)
func(b='world', a='hello')
```

Output:

```
hello
world
```

Arbitrary Keyword Arguments

It is possible to define a function so that any number of keyword arguments can be passed to it. This is done by adding ** before the argument name in the function definition. This was the function will receive a **dictionary** of arguments, and can access the items accordingly.
Example:

```
def func(**args):
   print(args['brand'])
func(brand='Ford', model='Mustang')
```

Output:

```
Ford
```

Default Argument Value

A function with an argument can be defined so that it automatically gives some default value to the argument if none is passed to it. Example:

```
def func(a=1):
   print(a)
func(2)
func()
```

Output:

```
2
1
```

GENERATOR FUNCTIONS

Generator functions are functions with `yield` statement instead of `return`. Unlike normal functions, generator functions will return a generator object, and won't execute immediately. The function will execute till the first `yield` statement when `next()` is called on the object, and pause. It will execute again till the next `yield` statement if `next()` is called again on the object, and pause again, and so on. Example:

```
def func(n):
   yield n
   yield n*n
obj = func(2) # obj is a generator object
print(type(obj))
print(next(obj))
print(next(obj))
```

Output:

```
<class 'generator'>
2
4
```

Another Example:

```
def func():
   yield 1
   yield 2
   yield 3
for x in func():  # generator objects are iterable
   print(x)
```

Output:

```
1
2
3
```

One more Example:

```
def func(n):
   yield n
   yield n*n
for x in func(3):  # generator objects are iterable
   print(x)
```

Output:

```
3
9
```

List comprehensions also create generator objects when enclosed within `()`. Example:

```
a = (x for x in [1, 2, 3])
print(type(a))
print(next(a))
print(next(a))
print(next(a))
# calling next() once more on a will give an error
```

Output:

```
<class 'generator'>
1
2
3
```

SCOPE OF VARIABLES

The scope of a variable is the region of code in which a defined variable is accessible. Outside of the scope of the variable, it will be undefined.

A variable that is defined inside a function cannot be accessed outside of the function. These are called local variables. The example below will throw an error:

```
def func():
    a = 5
print(a)
```

The above code will throw an error because `a` is a local variable defined inside of `func()`. It is undefined outside of the definition of `func()`, and hence it will be as if we are calling a variable that was never even created.

However variables that are created outside the function definition can be accessed from within the function. These are called global variables. Example:

```
a = 5
def func():
    print(a)
func()
```

Output: 5

If a local variable in a function definition has the same name as a global variable defined outside of the function, then for the scope of the function definition, the name will refer to the local variable. The global variable won't be modified. Example:

```
a = 10
def func():
    a = 5
    print(a)
print(a)
```

Output:

```
5
10
```

Although by default, a variable created inside a function definition will be a local variable, it is also possible to define a variable with global scope inside a function using the `global` keyword. Example:

```
def func():
    global a
    a = 5
func()
print(a)
```

Output: 5

In such a case it is necessary for the function to be called for the variable to be created. The below code will throw an error:

```
def func():
    global a
    a = 5
print(a)
```

LAMBDA FUNCTIONS

A lambda function or an anonymous function is a short hand way of defining a function in python.
A lambda function can take any number of arguments, but can only have one expression.
The general syntax for a lambda function is given below:

```
lambda arguments : expression
```

Lambda functions create objects (of class `function`) that act as functions.
Example:

```
x = lambda a : a + 10
print(type(x))
print(x(5))
```

Output:

```
<class 'function'>
15
```

Another Example:

```
x = lambda a, b: a + b
print(x(7,8))
```

Output:

```
15
```

Since lambda functions are objects, they can even be returned by a function. So it will be like the return value of a function is another function. Example:

```
def func(n):
   return lambda a : a*n
double = func(2)
triple = func(3)
print(double(5)) # will multiply 5 by 2 and print
print(triple(5)) # will multiply 5 by 3 and print
```

Output:

```
10
15
```

RECURSION

It is possible to call a function in its own definition. When done so it is called recursion.
Example:

```
def factorial(n):
   return n*factorial(n-1) if n>1 else 1
print(factorial(5))
```

Output: 120

Another Example (function that returns the n^{th} Fibonacci number):

```
def fib(n):
   return fib(n-1)+fib(n-2) if n>1 else n
print(fib(10))
```

Output: 55

Files

OPENING FILES

A file can be opened using the inbuilt function `open()` which returns a file object. Example:
```
f = open('file.txt')
```
In the above code a file with the name `file.txt` is opened in read mode and handled as a text file (which is the default mode), and a file object `f` is created. It is necessary that `file.txt` is in the same directory as the python program.

A file can be opened in various modes:
`'r'` —> Read - Default value. Opens a file for reading, error if the file does not exist.
`'a'` —> Append - Opens a file for appending, creates the file if it does not exist.
`'w'` —> Write - Opens a file for writing, creates the file if it does not exist.
`'x'` —> Create - Creates the specified file, returns an error if the file exists.

In addition you can specify if the file should be handled in text or binary mode:
`'t'` —> Text - Default value. Text mode.
`'b'` —> Binary - Binary mode (e.g. for images).

The mode in which the file is opened is specified as an additional argument while opening the file. If no additional argument is present, then the file is opened in read mode and handled as a text file as mentioned above. Below is an example where a file is opened in write mode.
```
f = open('file.txt', 'w')
# The above line is the same as f = open('file.txt', 'wt')
# But since text mode is default, 't' can be excluded
```
Another example where a file is opened in write mode and handled as binary:
```
f = open('file.txt', 'wb')
```

READING FILES

The file object returned by `open()` has an inbuilt method `read()` that returns the contents of the file. Example:
```
f = open('file.txt', 'rt')
# Here we have specified the read and text modes
# But as mentioned before, this is the default
# So it is not necessary to mention it
# Hence the above line is equivalent to f = open('file.txt')
print(f.read())
```
The above code will output the contents of the file `file.txt`

When `read()` is called with no arguments, the whole file is returned. But it is also possible to specify how many characters should be returned by giving an argument to the `read()` function. Example:
```
f = open('file.txt', 'rt')
print(f.read(5))
```
The above code will output the first 5 characters of `file.txt`

We could also read files line by line by using the built in method `readline()`. Example:
```
f = open('file.txt', 'rt')
print(f.readline())
```
The above code will return the first line in `file.txt`

If `readline()` is called multiple times, then subsequent lines in the opened file are returned.

Example:
```
f = open('file.txt', 'rt')
print(f.readline())
print(f.readline())
```
The above code will return the first two lines in `file.txt`

We could also create a list of all the lines in an opened file by calling `readlines()`.
Example:
```
f = open('file.txt', 'rt')
l = f.readlines()
print(type(l))
print(l[1])
```
Output:
```
<class 'list'>
second line of file.txt
```

WRITING TO FILES

Data can be written to files by using the argument `'a'` or `'w'` in the `open()` function.
`'a'` will append at the end of a file if it already exists.
`'w'` will overwrite the file if it already exists.
Both `'a'` and `'w'` will create a new file if it does not exist.

After a file is opened in one of the above write modes, the built-in method `write()` can be used to write data to the file.
Example:
```
f = open('file.txt', 'at')
print(f.write('hello world'))
```
The above code will add the text `hello world` at the end of the file `file.txt`
Another Example:
```
f = open('file.txt', 'wt')
print(f.write('hello world'))
```
The above code will overwrite `file.txt` so that the text `hello world` is the only data on it.

CLOSING FILES

After working with a file, it is a good programming habit to close it. In some cases, the changes made to the file may not show until the file is closed. Files are closed using the built-in method `close()`. Example:
```
f = open('file.txt')
print(f.read())
f.close()
```

Error Handling

Like many other programming languages, it possible to program how an error/exception that is raised during runtime must be handled. It is also possible throw custom exceptions in a program.

HANDLING EXCEPTIONS

Exceptions are handled using the `try`, `except`, `finally` keywords.
The `try` block lets you test a block of code for errors.
The `except` block lets you handle the error.
The `finally` block lets you execute code regardless of the result of the `try` and `except` blocks.

During runtime, when an exception occurs, python will normally stop execution and generate an error message. However if the line triggering the exception is in a `try` block, then instead of stopping and generating an error message as usual, the code in the `except` block is executed.
Example:

```
try:
    print(x)
except:
    print('This is a custom error message')
```

In the above code the variable `x` is not defined. Hence an exception will be triggered at runtime. But this exception will only cause the code in `except` block to be executed. Hence the output of the above code will be:

```
This is a custom error message
```

Common Exceptions

Exceptions triggered at runtime can be various types. A few common types are:

- `NameError` - Raised when a local or global name is not found. Example: Calling a variable that is not defined will trigger `NameError`
- `TypeError` - Raised when an operation or function is applied to an object of inappropriate type. Example: `'a' + 5` will trigger `TypeError`
- `IndexError` - Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range. If an index is not an integer, `TypeError` is raised.) Example: `(1, 2, 3)[5]` will trigger `IndexError`
- `ZeroDivisionError` - Raised when the second argument of a division or modulo operation is zero. Example: `5/0` will trigger `ZeroDivisionError`
- `StopIteration` - Raised by built-in function `next()` to signal that there are no further items produced by the iterator. Example:

```
x = (i for i in [1, 2])
a = next(x) # a will be 1
b = next(x) # b will be 2
c = next(x) # This will trigger StopIteration exception
```

Catching Specific Exceptions

Exception blocks (or `except` blocks) can be made to trigger only for particular types of exceptions. There can also be multiple exception blocks for a `try` block with exception block made to trigger for a different type of exception. Below is an example with two exception blocks

```
try:
   print(x)
except NameError:
# This block is executed if the exception raised is of type NameError
   print('Variable not defined')
except:
# This is the default except block
# This block is executed if an exception is raised but not caught above
# The default except block must always be last
   print('Some other error occurred')
```

Output: `Variable not defined`

Else Block

An `else` block can added be after `try` … `except` to execute a block of code if no error was triggered in the `try` block. Example

```
try:
   print('Hello')
except:
   print('Something went wrong')
else:
   print('Nothing went wrong')
```

Finally Block

This block is executed regardless of whether or not an exception is triggered in the `try` block. Example:

```
try:
   print(x)
except:
   print('Something went wrong')
finally:
   print('Finished error handling')
```

RAISING EXCEPTIONS

It is possible to raise custom errors for some particular conditions in a program. The `raise` keyword is used to do this. Example:

```
x = -1
if x < 0:
   raise Exception('Only positive numbers are allowed')
```

We could also raise custom errors of a particular type. Example:

```
x = 'hello'
if type(x) != int:
   raise TypeError('Only integers allowed')
```

Modules and Packages

MODULES

A module is like a code library that can be imported and used in a program.
To create a module, we just need to save the required code in a file with the file extension `.py`
A module can be imported for use in a program by using the `import` keyword.

Below is example of module creation and importing.

```
def fib(n):
    return fib(n-1)+fib(n-2) if n>1 else n
```

If the above code is saved in file with `.py` extension, then it can be imported as a module.
Let's say the above code is saved as `fibonacci.py`
Now the module can be imported in a new python program that is in the same directory.
Example of importing:

```
import fibonacci
print(fibonacci.fib(21))
```

Output: `10946`

Another example:

```
person1 = {
    'name'  = 'Suresh'
    'age' = 25
    'state' = 'Tamil Nadu'
}
person2 = {
    'name'  = 'Ganesh'
    'age' = 29
    'state' = 'Karnataka'
}
```

Let's say the above code is stored in `people.py`
It can be imported and used as shown below:

```
import people
print(people.person1['name'])
print(people.person2['state'])
```

Output:

```
Suresh
Karnataka
```

Modules can also be imported with a different name as shown below.

```
import people as persons # from the same people.py created above
print(persons.person1['age'])
```

Output: `25`

We could also import only specific items from a module as show below.

```
from fibonacci import fib # from the same fibonacci.py created above
print(fib(19))
```

Output: `4181`

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module. Example: `fib(19)` is correct, not `fibonacci.fib(19)`

PACKAGES

A python package is a collection of several modules. Physically, a package is a folder containing modules and maybe other folders that themselves may contain more folders and modules. Conceptually, it's a namespace. This simply means that a package's modules are bound together by a package name, by which they may be referenced. A package folder usually contains one file named `__init__.py` that basically tells python that the folder is a package. The init file may be empty, or it may contain code to be executed upon package initialisation.

Let's say the file `fibonacci.py` created before is moved to folder named `pack`.
Now a python program in the same directory as `pack` can access the file `fibonacci.py` as a module as shown below:

```
import pack.fibonacci
print(pack.fibonacci.fib(17))
```

Output: 1597

Here again importing can be done with different name as below.

```
import pack.fibonacci as sequence
print(sequence.fib(23))
```

Output: 28657

Importing specific items using `from` keyword can also be done as before.

```
from pack.fibonacci import fib
print(fib(15))
```

Output: 610

Now suppose the line

```
print('hello')
```

Is typed up in the file `__init__.py` and stored in `pack`. Then upon importing any module from `pack`, the code in `__init__.py` (in this case to print `hello`) is executed. Example:

```
from pack.fibonacci import fib
print(fib(25))
```

Output:

```
hello
75025
```

While importing from a package, it is possible to import all modules in it using `*`
However the `__init__.py` file must contain the list `__all__` containing all the module names in the package. Only the modules whose names are in `__all__` are imported when using `*`
So in our above example, to our file `__init__.py` we must add the line

```
__all__ = ['fibonacci']
```

Then in our program we could do

```
from pack import *
print(fibonacci.fib(21))
```

Output:

```
hello
10946
```

However in general it is a bad programming practice to import all modules from a package as it clutters the namespace.

SOME COMMON PYTHON MODULES

Math

This module contains commonly used mathematical functions like:
Trigonometric functions (`math.sin()`, `math.cos()`, `math.tan()`, etc.)
Inverse Trigonometric functions (`math.asin()`, `math.acos()`, `math.atan()`, etc.)
Hyperbolic functions (`math.sinh()`, `math.cosh()`, `math.tanh()`, etc.)
Inverse Hyperbolic functions (`math.asinh()`, `math.acosh()`, `math.atanh()`, etc.)
And more (`math.exp()`, `math.log()`, `math.factorial()`, etc.)

Random

This module is used to generate pseudo random numbers. Some of it's methods are:
`random()` - Generates random number in the interval [0, 1)
`randint(a, b)` - Generates a random integer N such that a <= N <= b
`choice(seq)` - Returns a random element from a non-empty sequence

NUMPY

What is NumPy?

NumPy is a Python library used for working with arrays.
It also has functions for working in domain of linear algebra, fourier transform, and matrices.
NumPy was created in 2005 by Travis Oliphant. It is open-source and can be used freely.
NumPy stands for Numerical Python.

Why Use NumPy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.
NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
The array object in NumPy is called `ndarray`. It provides a lot of supporting functions that make working with `ndarray` very easy.
Arrays are very frequently used in data science, where speed and resources are very important.

Installing NumPy

NumPy can be installed using the python package manger `pip` via command-line. The command for installing NumPy is:

```
pip install numpy
```

Creating NumPy Arrays

We can create a NumPy `ndarray` object by using the `array()` function. Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5]) # the argument can also be a tuple
print(arr)
print(type(arr))
```

Output:

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

Basic Arithmetic on Arrays

Arithmetic operators on NumPy Arrays act element-wise.

Example:
```
import numpy as np
arr = np.array([1, 2, 3])
print(arr + 1)
```
Output: [2, 3, 4]

Another Example:
```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print(a + b)
```
Output: [5, 7, 9]

Basic Array Manipulation

Slicing
This is same as for usual python lists and tuples. Example:
```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```
Output: [2 3 4 5]

Sorting
NumPy arrays can be easily sorted using the inbuilt function `sort()`. Example:
```
import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```
Output: [0, 1, 2, 3]

PANDAS

What is Pandas?

Pandas is a Python library used for working with data sets.
It has functions for analysing, cleaning, exploring, and manipulating data.
The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

Pandas allows us to analyse big data and make conclusions based on statistical theories.
Pandas can clean messy data sets, and make them readable and relevant.
Relevant data is very important in data science.

Installing Pandas

Pandas can be installed using the python package manger `pip` via command-line. The command for installing Pandas is:
```
pip install pandas
```

Series

A Pandas Series is like a column in a table.
It is a one-dimensional array holding data of any type.
It can be created from a list as below.

```
import pandas as pd
a = [1, 7, 2]
srs = pd.Series(a)
print(srs)
```

Output:

```
0    1
1    7
2    2
dtype: int64
```

DataFrames

A Pandas DataFrame is a two-dimensional data structure, like a two-dimensional array, or a table with rows and columns. It can be created from a dictionary as below.

```
import pandas as pd
data = {
   "calories": [420, 380, 390],
   "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
print(df)
```

Output:

```
   calories  duration
0       420        50
1       380        40
2       390        45
```

DataFrames can also be created from files (and usually is) as shown below.

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df)
```

The above code will load data from a Comma Separated file (CSV file) into a DataFrame and display it.