```
print('hello') #textual literals
```

```
hello
```

```
number = eval(input("Enter an expression:"))
number
```

```
Enter an expression: 33+24
57
```

```
float(5)
```

```
5.0
```

```
int(5.64687)
```

```
5
```

```
str(55.2210)
```

```
'55.221'
```

```
str(10.00)
```

```
'10.0'
```

```
str(10.0)
```

```
'10.0'
```

```
str(10.06)
```

```
'10.06'
```

```
str(10.654)
```

```
'10.654'
```

```
_var = 3
_var
```

```
3
```

```
user_input = eval(input("Enter any expression:"))
user_input
```

```
Enter any expression: 8+9.7+3
20.7
```

```
name = "shawaiz"
age = 20
str(age)
```

```
'20'
```

```
float(age)
```

```
20.0
```

```
x,y,z = 5,8.4,900
print(x) # numerical literals
print(y)
print(z)
```

```
5
8.4
900
```

```
isRunning = False
isRunning
```

```
False
```

```
x = abs(-92)
x
```

```
92
```

```
model = int("57")
model
```

```
57
```

```
model = 2 + 3.2 # implicity type conversion
model
```

```
5.2
```

```
height = 8 // 2 + 2.6
height
```

```
6.6
```

```
fName = "Ahmad"
lName = "Ali"
name = fName + lName
a = name*3 # AhmadAliAhmadAliAhmadAli
b = a[-1]
print(a)
print(b)
```

```
AhmadAliAhmadAliAhmadAli
i
```

```
"""
This is a multi-line comment.
Using triple quotes (single or double).
"""
```

```
'\nThis is a multi-line comment.\nUsing triple quotes (single or double).\n'
```

```
animals = ["zeebra", "deer", "lion", "cheeta", "elephant", "rhino", "leopard", "horse"]
animals
```

```
['zeebra', 'deer', 'lion', 'cheeta', 'elephant', 'rhino', 'leopard', 'horse']
```

```
for i in animals:
    print(i)
```

```
zeebra
deer
lion
cheeta
elephant
rhino
leopard
horse
```

```
print??
```

```
Signature: print(*args, sep=' ', end='\n', file=None, flush=False)
Docstring:
Prints the values to a stream, or to sys.stdout by default.

sep
  string inserted between values, default a space.
end
  string appended after the last value, default a newline.
file
  a file-like object (stream); defaults to the current sys.stdout.
flush
  whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

```
print?
```

```
Signature: print(*args, sep=' ', end='\n', file=None, flush=False)
Docstring:
Prints the values to a stream, or to sys.stdout by default.

sep
  string inserted between values, default a space.
end
  string appended after the last value, default a newline.
file
  a file-like object (stream); defaults to the current sys.stdout.
flush
  whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

```
1 for i in animals:
2     print(i, end=", ")
```

```
zeebra, deer, lion, cheeta, elephant, rhino, leopard, horse,
```

```
1 for index, item in enumerate(animals):
2     print(f"{index}: {item}") # Using f-string
```

```
0: zeebra
1: deer
2: lion
3: cheeta
4: elephant
5: rhino
6: leopard
7: horse
```

```
1 help("for")
```

```
The "for" statement
*******************

The "for" statement is used to iterate over the elements of a sequence
(such as a string, tuple or list) or other iterable object:

   for_stmt ::= "for" target_list "in" starred_list ":" suite
                ["else" ":" suite]

The "starred_list" expression is evaluated once; it should yield an
*iterable* object.  An *iterator* is created for that iterable. The
first item provided by the iterator is then assigned to the target
list using the standard rules for assignments (see Assignment
statements), and the suite is executed.  This repeats for each item
provided by the iterator.  When the iterator is exhausted, the suite
in the "else" clause, if present, is executed, and the loop
terminates.

A "break" statement executed in the first suite terminates the loop
without executing the "else" clause's suite.  A "continue" statement
executed in the first suite skips the rest of the suite and continues
with the next item, or with the "else" clause if there is no next
item.

The for-loop makes assignments to the variables in the target list.
This overwrites all previous assignments to those variables including
those made in the suite of the for-loop:

   for i in range(10):
       print(i)
       i = 5             # this will not affect the for-loop
                         # because i will be overwritten with the next
                         # index in the range

Names in the target list are not deleted when the loop is finished,
but if the sequence is empty, they will not have been assigned to at
all by the loop.  Hint: the built-in type "range()" represents
immutable arithmetic sequences of integers. For instance, iterating
"range(3)" successively yields 0, 1, and then 2.

Changed in version 3.11: Starred elements are now allowed in the
expression list.

Related help topics: break, continue, while
```

```
1 range??
```

```
Init signature: range(self, /, *args, **kwargs)
Docstring:
range(stop) -> range object
range(start, stop[, step]) -> range object

Return an object that produces a sequence of integers from start (inclusive)
to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ..., j-1.
start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2, 3.
These are exactly the valid indices for a list of 4 elements.
When step is given, it specifies the increment (or decrement).
Type:           type
Subclasses:
```

```
1 for i in range(0, 20, 5): # range(start, stop, increment)
2     print(i)
```

```
0
5
10
15
```

```
1 for i in range(20, 0, -5): # range(start, stop, decrement)
2     print(i)
```

```
20
15
10
5
```

```
1 for index in range(len(animals)):
2     item = animals[index]
3     print(f"{index}: {item}")
```

```
0: zeebra
1: deer
2: lion
3: cheeta
4: elephant
5: rhino
6: leopard
7: horse
```

```
1 enumerate??
```

```
Init signature: enumerate(iterable, start=0)
Docstring:
Return an enumerate object.

  iterable
    an object supporting iteration

The enumerate object yields pairs containing a count (from start, which
defaults to zero) and a value yielded by the iterable argument.

enumerate is useful for obtaining an indexed list:
    (0, seq[0]), (1, seq[1]), (2, seq[2]), ...
Type:           type
Subclasses:
```

```
1 for i in enumerate(animals, start=1):
2     print(i)
```

```
(1, 'zeebra')
(2, 'deer')
(3, 'lion')
(4, 'cheeta')
(5, 'elephant')
(6, 'rhino')
(7, 'leopard')
(8, 'horse')
```

```
1 for index, item in enumerate(animals, start=1):
2     print(f"{index}: {item}")
```

```
1: zeebra
2: deer
3: lion
4: cheeta
5: elephant
6: rhino
7: leopard
8: horse
```

```
i = 0
while i < len(animals):
    print(f"{i+1}-{animals[i]}", end=", ")
    i += 1
```

```
1-zeebra, 2-deer, 3-lion, 4-cheeta, 5-elephant, 6-rhino, 7-leopard, 8-horse,
```

```
ls = [34, 2, 5] # list
ls
```

```
[34, 2, 5]
```

```
ls * 2 # list
```

```
[34, 2, 5, 34, 2, 5]
```

```
import numpy as np
arr = np.array([10, 5, 30]) # array
arr * arr
```

```
array([100,  25, 900])
```

```
np.array2string(arr)
```

```
'[10  5 30]'
```

```
str(arr)
```

```
'[34, 2, 5]'
```

```
for index, item in enumerate(arr):
  arr[index] = str(arr[index])
arr
```

```
['34', '2', '5']
```

```
arr * 2
```

```
['34', '2', '5', '34', '2', '5']
```

```
def func():
    print("hello")

func()
```

```
hello
```

```
def func(*argv): # for multiple arguments
    for i in argv:
        print(i)

func("animals", 5, 6.387, 50+65, 0)
```

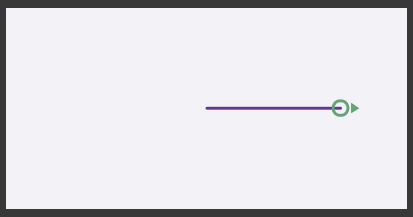
```
animals
5
6.387
115
0
```

```
"""
Reusable programs are usually created by typing definitions into a separate file
called a module or script.
▪ This file is saved on disk so that it can be used over and over again

▪ A Python module file is just a text file with a .py extension, which can be created using
any program for editing text (e.g., notepad or vim)

▪ Namespace Isolation: Each module defines its own namespace, preventing naming conflicts between
identifiers (variables, functions, classes) in different parts of a program.

▪ Built-in Modules: math for mathematical operations, random for generating random numbers,
and os for interacting with the operating system.


path/module.py
-------------------------------------------------------
def printMsg(msg):
```

```
print (msg*2)
------------------------------------------------------------

Notebook or any other python program
------------------------------------------------------------
import sys
sys.path.append('path to folder where module.py exists')
import module as myf

myf.printMsg("hello")
------------------------------------------------------------

"""
```

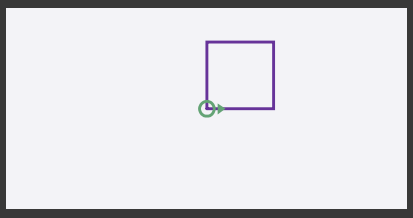
```
!pip install jupyturtle
```

```
import jupyturtle as jt
jt.make_turtle()
jt.forward(100)
```

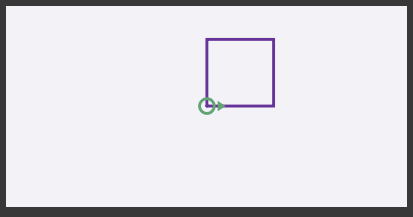


```
from jupyturtle import make_turtle, forward, left, right
```

```
make_turtle()

forward(50)
left(90)

forward(50)
left(90)

forward(50)
left(90)

forward(50)
left(90)
```



```
make_turtle()
for i in range(4):
    forward(50)
    left(90)
```



Data Structures

•List	Ordered,	changeable,	duplicates
•Tuple	Ordered,	unchangeable,	duplicates
•Set	Unordered,	addable/removable	no duplicates
•Dictionary	Unordered,	changeable,	no duplicate

List	Tuple	Set	Dictionary
L= [12, "banana", 5.3]	T= (12, "banana", 5.3)	S= {12, "banana", 5.3}	D={"Val":12,"name":"Ban"}
L[1]	T[2]	X in S	D["Val"]
L = L + ["game"] L[2] = "orange"	Immutable T3 = T1+T2	S.add("new Item") S.update({"more","items"})	D["Val"] = newValue D["newkey"] = "newVal"
del L[1] del L	Immutable del T	S.Remove("banana") del S	del D["Val"] del D
L2 = L.copy()	T2 = T	S2 = S.copy()	D2 = D.copy()
....	....	....	....

List

You can't do direct arithmetic operations on lists in Python like you would with numbers. Using standard operators like +, -, *, or / on lists will not perform element-wise calculations. Instead, they will either concatenate repeat, or raise a TypeError.

Tuple

- **Ordered:** The elements in a tuple have a defined order, and this order will not change. Each element has an index, starting from 0.
- **Immutable:** This is the most significant characteristic. Once a tuple is created, its contents cannot be modified. You cannot change an element, add new elements, or remove existing ones.

Sets

- **Unordered:** Elements within a set do not have a defined order, and their position can change. This means you cannot access elements by index.
- **Unique Elements:** Sets automatically handle duplicate entries, ensuring that each element appears only once. If you attempt to add a duplicate, it will simply be ignored.
- **Immutable Elements:** While sets themselves are mutable (elements can be added or removed), the elements they contain must be immutable data types (e.g., numbers, strings, tuples). Mutable types like lists or dictionaries cannot be direct elements of a set.
- **Mutable:** You can add or remove elements from a set after it has been created.

```
1 arr = ['test', 13, 1.131]
2 arr += ["game"]
3 arr
```

```
['test', 13, 1.131, 'game']
```

```
1 del arr[0]
2 arr
```

```
[13, 1.131, 'game']
```

```
1 arr2 = arr.copy()
2 arr2
```

```
[13, 1.131, 'game']
```

```
1 del arr
2 arr
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_36/612514380.py in <cell line: 0>()
      1 del arr
----> 2 arr

NameError: name 'arr' is not defined
```

```
1 # Shape (4, 2, 3, 4)
2 arr = list([
3     [
4       [[0,1,2,3],[4,5,6,7],[8,9,10,11]
5       ],
6       [[0,1,2,3],[4,5,6,7],[8,9,10,11]
```

```
      ]
            ],
    [
      [[0,1,2,3],[4,5,6,7],[8,9,10,11]
      ],
      [[0,1,2,3],[4,5,6,7],[8,9,10,11]
      ]
            ],
    [
      [[0,1,2,3],[4,5,6,7],[8,9,10,11]
      ],
      [[0,1,2,3],[4,5,6,7],[8,9,10,11]
      ]
            ],
    [
      [[0,1,2,3],[4,5,6,7],[8,9,10,11]
      ],
      [[0,1,2,3],[4,5,6,7],[8,9,10,11]
      ]
            ]
           ])

arr
```

```
[[[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]],
  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]],
 [[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]],
  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]],
 [[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]],
  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]],
 [[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]],
  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]]]
```

```
# Dictionary from lists
keys = ['id', 'name', 'age']
values = [1520, 'John', 20]

arr_dict = {
    keys[0]: values[0],
    keys[1]: values[1],
    keys[2]: values[2]
}

print(arr_dict)
```

```
{'id': 1520, 'name': 'John', 'age': 20}
```

```
arr = [2,4,6,8]
result = arr * arr # This only works with numpy arrays
print(result)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-734027852.py in <cell line: 0>()
      1 arr = [2,4,6,8]
----> 2 result = arr * arr
      3 print(result)

TypeError: can't multiply sequence by non-int of type 'list'
```

```
# An empty tuple
empty_tuple = ()

# A tuple with mixed data types
my_tuple = (1, "hello", 3.14, True)

# A tuple with a single element (note the comma)
single_element_tuple = ("apple",)
print(single_element_tuple, len(single_element_tuple))

# Creating a tuple from an iterable using the tuple() constructor
my_list = [1, 2, 3]
tuple_from_list = tuple(my_list)
```

```
('apple',) 1
```

Code	Type	Explanation
`('A',)`	`tuple`	The comma makes it a tuple.
`('A')`	`str`	The parentheses are ignored; it's just a string.
`42,`	`tuple`	Comma alone creates a tuple (parentheses are optional).
`(42)`	`int`	Parentheses are ignored; it's just an integer.

```
my_tuple = (10, 20, 30, 40)

print(my_tuple[0])  # Output: 10
print(my_tuple[-1]) # Output: 40 (last element)
print(my_tuple[1:3]) # Output: (20, 30) (slicing)

tuple_copy = my_tuple
print(tuple_copy)
```

```
10
40
(20, 30)
(10, 20, 30, 40)
```

```
my_set = {1, 2, 3, "apple", "banana"}
my_set
```

```
{1, 2, 3, 'apple', 'banana'}
```

```
another_set = set([1, 2, 2, 3, "orange"]) # Duplicates are removed automatically
empty_set = set() # To create an empty set, use set(), not {} (which creates an empty dictionary)
another_set
```

```
{1, 2, 3, 'orange'}
```

```
my_set.add(4)
my_set.update([5, 6]) # Add multiple elements from an iterable
my_set
```

```
{1, 2, 3, 4, 5, 6, 'apple', 'banana'}
```

```
my_set.remove(1) # Raises KeyError if element not found
print(my_set)
my_set.discard("apple") # Does not raise an error if element not found
print(my_set)
my_set.pop() # Removes and returns the first element
print(my_set)
my_set.clear() # Removes all elements
print(my_set)
```

```
{2, 3, 4, 5, 6, 'apple', 'banana'}
{2, 3, 4, 5, 6, 'banana'}
{3, 4, 5, 6, 'banana'}
set()
```

```
my_set = {"apple", "banana"}
print(my_set)

if "banana" in my_set:
    print("Banana is in the set.")
```

```
{'apple', 'banana'}
Banana is in the set.
```

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

union_set = set1.union(set2)         # {1, 2, 3, 4, 5}
intersection_set = set1.intersection(set2) # {3}
difference_set = set1.difference(set2)   # {1, 2} (elements in set1 but not in set2)
symmetric_difference_set = set1.symmetric_difference(set2) # {1, 2, 4, 5} (elements in either
# set but not both)
```