

Introduction to Python Programming

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

August 22, 2023

Python Programming Language

- Python is a high-level, general-purpose programming language
- Created by Guido van Rossum
 - First public release in 1991 (version 0.9.0)
 - Python 3.0 came in late 2008, latest stable release is 3.11
 - Python 3.x is backwards-incompatible (with Python 2.x)
- Python is an interpreted language

Python Programming Language

- Python is a high-level, general-purpose programming language
- Created by Guido van Rossum
 - First public release in 1991 (version 0.9.0)
 - Python 3.0 came in late 2008, latest stable release is 3.11
 - Python 3.x is backwards-incompatible (with Python 2.x)
- Python is an interpreted language
- Pros: free & open source, portable, large library support, ...

Python Programming Language

- Python is a high-level, general-purpose programming language
- Created by Guido van Rossum
 - First public release in 1991 (version 0.9.0)
 - Python 3.0 came in late 2008, latest stable release is 3.11
 - Python 3.x is backwards-incompatible (with Python 2.x)
- Python is an interpreted language
- Pros: free & open source, portable, large library support, ...
 - Typical usage: scientific calculations, AI/ML, data science, web development, database access, network programming, game Development etc.

Python Programming Language

- Python is a high-level, general-purpose programming language
- Created by Guido van Rossum
 - First public release in 1991 (version 0.9.0)
 - Python 3.0 came in late 2008, latest stable release is 3.11
 - Python 3.x is backwards-incompatible (with Python 2.x)
- Python is an interpreted language
- Pros: free & open source, portable, large library support, ...
 - Typical usage: scientific calculations, AI/ML, data science, web development, database access, network programming, game Development etc.
- Cons: slow speed and heavy memory usage

Using Python as a Calculator

- Access the Python interpreter
 - Open IDLE (built-in Integrated Development and Learning Environment) or
 - run `python3` command in a terminal

```
rathin@laptop:~$ python3
Python 3.10.6 (main, May 29 2023, 11:10:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Using Python as a Calculator

- Access the Python interpreter
 - Open IDLE (built-in Integrated Development and Learning Environment) or
 - run `python3` command in a terminal

```
rathin@laptop:~$ python3
Python 3.10.6 (main, May 29 2023, 11:10:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

- Try these out:
 - $((2 + 4)/3) * (5 - 8)$
 - $(2/4) + (3//2) + (10\%4)$
 - $3 ** 2$

Using Python as a Calculator

- Access the Python interpreter
 - Open IDLE (built-in Integrated Development and Learning Environment) or
 - run `python3` command in a terminal

```
rathin@laptop:~$ python3
Python 3.10.6 (main, May 29 2023, 11:10:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

- Try these out:
 - $((2 + 4)/3) * (5 - 8)$
 - $(2/4) + (3//2) + (10\%4)$
 - $3 ** 2$
- Determine the results:
 - $3 + 4 * 5 - 6 / 3$
 - $- 3 ** 2 ** 3$

Using Python as a Calculator

- Access the Python interpreter
 - Open IDLE (built-in Integrated Development and Learning Environment) or
 - run `python3` command in a terminal

```
rathin@laptop:~$ python3
Python 3.10.6 (main, May 29 2023, 11:10:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

- Try these out:
 - $((2 + 4)/3) * (5 - 8)$
 - $(2/4) + (3//2) + (10\%4)$
 - $3 ** 2$
- Determine the results:
 - $3 + 4 * 5 - 6 / 3$
 - $- 3 ** 2 ** 3$
- Refer to the precedence and associativity of the operators

Variables

- Symbolic name, refers to a memory location containing some value

Variables

- Symbolic name, refers to a memory location containing some value
- Python variables are dynamically typed
 - Appropriate type is determined by the value:
 - Variables in C/C++/Java etc. are static typed and requires a declaration

Variables

- Symbolic name, refers to a memory location containing some value
- Python variables are dynamically typed
 - Appropriate type is determined by the value: use `type()` to inspect
 - Variables in C/C++/Java etc. are static typed and requires a declaration

Variables

- Symbolic name, refers to a memory location containing some value
- Python variables are dynamically typed
 - Appropriate type is determined by the value: use `type()` to inspect
 - Variables in C/C++/Java etc. are static typed and requires a declaration

- Try out the following:

```
a = 3
b = 2
c = a / b
type(c)
d = a // b
type(d)
```

- Common built-in types: `bool`, `int`, `float`, `str`, `list`, `dict` etc.

Object Identity

- `id()` returns the identity of an object
 - 'identity' is a unique integer value at any given time
 - (loosely) it refers to memory address of the stored value
 - objects with non-overlapping lifetimes may have the same identity

Object Identity

- `id()` returns the identity of an object
 - 'identity' is a unique integer value at any given time
 - (loosely) it refers to memory address of the stored value
 - objects with non-overlapping lifetimes may have the same identity
- Operators `is` and `is not` are used to test object identity

Object Identity

- `id()` returns the identity of an object
 - 'identity' is a unique integer value at any given time
 - (loosely) it refers to memory address of the stored value
 - objects with non-overlapping lifetimes may have the same identity
- Operators `is` and `is not` are used to test object identity
 - Execute this:

```
a = 4 / 2
b = 4 // 2
a == b
a is b
c = 3 - 1
b is c
```

- Use `type()` and `id()` to explain the output

Strings

- String literals can be defined using either single quote or double quote (but don't mix them)
- Special characters can be escaped (like in C/C++/Java)

```
a = 'hi'  
b = "there"
```

Strings

- String literals can be defined using either single quote or double quote (but don't mix them)
- Special characters can be escaped (like in C/C++/Java)
- Python has many built-in string methods (see the documentation)
- `len()` returns the length (number of characters) of a string

```
a = 'hi'  
b = "there"  
len(a)  
len(b)
```

Strings

- String literals can be defined using either single quote or double quote (but don't mix them)
- Special characters can be escaped (like in C/C++/Java)
- Python has many built-in string methods (see the documentation)
- `len()` returns the length (number of characters) of a string
- The `+` operator can be used to concatenate two strings

```
a = 'hi'  
b = "there"  
len(a)  
len(b)  
a + ' ' + b
```

Strings

- String literals can be defined using either single quote or double quote (but don't mix them)
- Special characters can be escaped (like in C/C++/Java)
- Python has many built-in string methods (see the documentation)
- `len()` returns the length (number of characters) of a string
- The `+` operator can be used to concatenate two strings
- String comparison can be done simply using the relational operators: `==`, `!=`, `>`, `>=`, `<`, `<=`

```
a = 'hi'  
b = "there"  
len(a)  
len(b)  
a + ' ' + b  
'rathin' > 'Rathin'
```

Assignments in Python

- Operator = assigns RHS value (evaluates if an expression is given) to the variable in the LHS

Assignments in Python

- Operator = assigns RHS value (evaluates if an expression is given) to the variable in the LHS
- Chained assignments: `a = b = 3+2`

Assignments in Python

- Operator = assigns RHS value (evaluates if an expression is given) to the variable in the LHS
- Chained assignments: `a = b = 3+2`
- Multiple assignments: `a, b, c = 1, 2, 3`

Assignments in Python

- Operator = assigns RHS value (evaluates if an expression is given) to the variable in the LHS
- Chained assignments: `a = b = 3+2`
- Multiple assignments: `a, b, c = 1, 2, 3`
- Swap between two variables: `a, b = b, a`

Assignments in Python

- Operator = assigns RHS value (evaluates if an expression is given) to the variable in the LHS
- Chained assignments: `a = b = 3+2`
- Multiple assignments: `a, b, c = 1, 2, 3`
- Swap between two variables: `a, b = b, a`
- Shorthand operators like `+=`, `*=` etc. are also available

Assignments in Python

- Operator = assigns RHS value (evaluates if an expression is given) to the variable in the LHS
- Chained assignments: `a = b = 3+2`
- Multiple assignments: `a, b, c = 1, 2, 3`
- Swap between two variables: `a, b = b, a`
- Shorthand operators like `+=`, `*=` etc. are also available
- There is no increment/decrement operator in Python
- Instead we use something like: `i += 1`

Python Lists

- A built-in datatype to hold an ordered collection of items
`a = [1, 2, 3]` # *square brackets denotes a list*

Python Lists

- A built-in datatype to hold an ordered collection of items
`a = [1, 2, 3]` # *square brackets denotes a list*
- Values can be of any type: `a = [1, 'hi', 3.0, 1]`
- A list can be empty as well: `a = []` or `a = list()`

Python Lists

- A built-in datatype to hold an ordered collection of items
`a = [1, 2, 3]` # *square brackets denotes a list*
- Values can be of any type: `a = [1, 'hi', 3.0, 1]`
- A list can be empty as well: `a = []` or `a = list()`
- Items are indexed from 0 to $n - 1$
- Accessing items via (valid) index: `a[2]` # *3rd item*

Python Lists

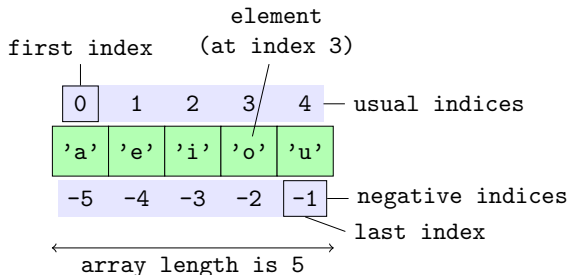
- A built-in datatype to hold an ordered collection of items
`a = [1, 2, 3]` # *square brackets denotes a list*
- Values can be of any type: `a = [1, 'hi', 3.0, 1]`
- A list can be empty as well: `a = []` or `a = list()`
- Items are indexed from 0 to $n - 1$
- Accessing items via (valid) index: `a[2]` # *3rd item*
- Index can be negative ($-n$ to -1): `a[-1]` # *last item*

Python Lists

- A built-in datatype to hold an ordered collection of items
`a = [1, 2, 3]` *# square brackets denotes a list*
- Values can be of any type: `a = [1, 'hi', 3.0, 1]`
- A list can be empty as well: `a = []` or `a = list()`
- Items are indexed from 0 to $n - 1$
- Accessing items via (valid) index: `a[2]` *# 3rd item*
- Index can be negative ($-n$ to -1): `a[-1]` *# last item*
- `len()` method returns the size of a list: `len([1,2,4])` *#gives 3*

Python Lists

- A built-in datatype to hold an ordered collection of items
`a = [1, 2, 3]` # *square brackets denotes a list*
- Values can be of any type: `a = [1, 'hi', 3.0, 1]`
- A list can be empty as well: `a = []` or `a = list()`
- Items are indexed from 0 to $n - 1$
- Accessing items via (valid) index: `a[2]` # *3rd item*
- Index can be negative ($-n$ to -1): `a[-1]` # *last item*
- `len()` method returns the size of a list: `len([1,2,4])` #*gives 3*



Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items

Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items
 - Modify an item: `a[2] = 30`

Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items

- Modify an item: `a[2] = 30`

- Add a single item:

```
a.append(22) # adds item to the end
```

```
a.insert(2, 'apple') # adds item to the specific index
```

Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items
 - Modify an item: `a[2] = 30`
 - Add a single item:
 - `a.append(22)` *# adds item to the end*
 - `a.insert(2, 'apple')` *# adds item to the specific index*
 - Add multiple items: `a.extend([25,30,33])`

Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items

- Modify an item: `a[2] = 30`

- Add a single item:

- `a.append(22)` # adds item to the end

- `a.insert(2, 'apple')` # adds item to the specific index

- Add multiple items: `a.extend([25,30,33])`

- Remove a specific item:

- `a.remove(30)` # removes first occurrence of 30

Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items
 - Modify an item: `a[2] = 30`
 - Add a single item:
 - `a.append(22)` # adds item to the end
 - `a.insert(2, 'apple')` # adds item to the specific index
 - Add multiple items: `a.extend([25,30,33])`
 - Remove a specific item:
 - `a.remove(30)` # removes first occurrence of 30
 - Remove from a specific index: `del a[1]` # deletes 2nd item

Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items

- Modify an item: `a[2] = 30`

- Add a single item:

```
a.append(22) # adds item to the end
```

```
a.insert(2, 'apple') # adds item to the specific index
```

- Add multiple items: `a.extend([25,30,33])`

- Remove a specific item:

```
a.remove(30) # removes first occurrence of 30
```

- Remove from a specific index: `del a[1]` # deletes 2nd item

- Remove and get a specific item:

```
a.pop(0) # deletes and returns the first item
```

```
a.pop() # pops the last item, default index is -1
```

Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items
 - Modify an item: `a[2] = 30`
 - Add a single item:
 - `a.append(22)` # adds item to the end
 - `a.insert(2, 'apple')` # adds item to the specific index
 - Add multiple items: `a.extend([25,30,33])`
 - Remove a specific item:
 - `a.remove(30)` # removes first occurrence of 30
 - Remove from a specific index: `del a[1]` # deletes 2nd item
 - Remove and get a specific item:
 - `a.pop(0)` # deletes and returns the first item
 - `a.pop()` # pops the last item, default index is -1
- Two lists can be joined with + operator: `[1,2] + [3,4,5]`
 - `extend()` adds items into same list while joining creates a new one

Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items
 - Modify an item: `a[2] = 30`
 - Add a single item:
 - `a.append(22)` # adds item to the end
 - `a.insert(2, 'apple')` # adds item to the specific index
 - Add multiple items: `a.extend([25,30,33])`
 - Remove a specific item:
 - `a.remove(30)` # removes first occurrence of 30
 - Remove from a specific index: `del a[1]` # deletes 2nd item
 - Remove and get a specific item:
 - `a.pop(0)` # deletes and returns the first item
 - `a.pop()` # pops the last item, default index is -1
- Two lists can be joined with + operator: `[1,2] + [3,4,5]`
 - `extend()` adds items into same list while joining creates a new one
- * operator repeats the list: `[1,2,3]*3` # `[1,2,3,1,2,3,1,2,3]`

Python Lists (contd.)

- Lists are mutable; we can modify/add/delete items
 - Modify an item: `a[2] = 30`
 - Add a single item:

```
a.append(22) # adds item to the end
a.insert(2, 'apple') # adds item to the specific index
```
 - Add multiple items: `a.extend([25,30,33])`
 - Remove a specific item:

```
a.remove(30) # removes first occurrence of 30
```
 - Remove from a specific index: `del a[1]` # deletes 2nd item
 - Remove and get a specific item:

```
a.pop(0) # deletes and returns the first item
a.pop() # pops the last item, default index is -1
```
- Two lists can be joined with + operator: `[1,2] + [3,4,5]`
 - `extend()` adds items into same list while joining creates a new one
- * operator repeats the list: `[1,2,3]*3` # `[1,2,3,1,2,3,1,2,3]`
- Membership test: `item in list1` or `item not in list1`

Python Lists (contd.)

- Lists can be sliced: `a[start:end:step]`

Python Lists (contd.)

- Lists can be sliced: `a[start:end:step]`
- Try the following:

```
a = [10, 20, 30, 40, 50, 60]
a[1:3]
a[:4]
a[3:]
a[:]
a[2::2]
a[::-1]
```

Python Lists (contd.)

- Lists can be sliced: `a[start:end:step]`
- Try the following:

```
a = [10, 20, 30, 40, 50, 60]
a[1:3]
a[:4]
a[3:]
a[:]
a[2::2]
a[::-1]
```

- Updating a slice: `a[2:5] = [1, 2, 3]`
- Deleting a slice: `del a[1:4]`

Python Lists (contd.)

- Lists can be sliced: `a[start:end:step]`
- Try the following:

```
a = [10, 20, 30, 40, 50, 60]
a[1:3]
a[:4]
a[3:]
a[:]
a[2::2]
a[::-1]
```

- Updating a slice: `a[2:5] = [1, 2, 3]`
- Deleting a slice: `del a[1:4]`
- Copying a list: `list2 = list1.copy()` or `list2 = list1[:]`

Python Lists (contd.)

- Lists can be sliced: `a[start:end:step]`
- Try the following:

```
a = [10, 20, 30, 40, 50, 60]
a[1:3]
a[:4]
a[3:]
a[:]
a[2::2]
a[::-1]
```

- Updating a slice: `a[2:5] = [1, 2, 3]`
- Deleting a slice: `del a[1:4]`
- Copying a list: `list2 = list1.copy()` or `list2 = list1[:]`
- Reversing a list (in place):

Python Lists (contd.)

- Lists can be sliced: `a[start:end:step]`
- Try the following:

```
a = [10, 20, 30, 40, 50, 60]
a[1:3]
a[:4]
a[3:]
a[:]
a[2::2]
a[::-1]
```

- Updating a slice: `a[2:5] = [1, 2, 3]`
- Deleting a slice: `del a[1:4]`
- Copying a list: `list2 = list1.copy()` or `list2 = list1[:]`
- Reversing a list (in place): `list1.reverse()`

Python Lists (contd.)

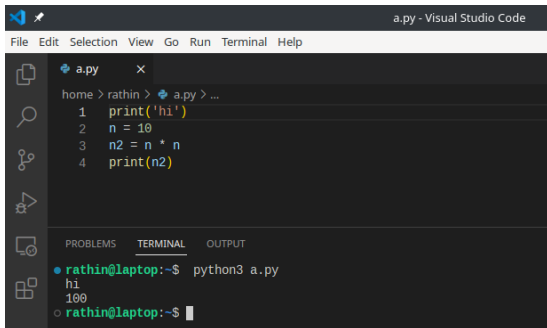
- Lists can be sliced: `a[start:end:step]`
- Try the following:

```
a = [10, 20, 30, 40, 50, 60]
a[1:3]
a[:4]
a[3:]
a[:]
a[2::2]
a[::-1]
```

- Updating a slice: `a[2:5] = [1, 2, 3]`
- Deleting a slice: `del a[1:4]`
- Copying a list: `list2 = list1.copy()` or `list2 = list1[:]`
- Reversing a list (in place): `list1.reverse()`
- Clearing a list: `list1.clear()` or `del list1[:]`

Switching to Python Scripts

- We write Python scripts in some file and save it with `.py` extension
- You can use a simple text editor to write Python programs: gedit, vim, notepad++, ...
- Or use a powerful IDE: VS Code, Spyder, ...
- Executing Python scripts from terminal: `python3 filename.py`



The screenshot shows the Visual Studio Code interface. The editor window displays a Python script named `a.py` with the following code:

```
1 print('hi')
2 n = 10
3 n2 = n * n
4 print(n2)
```

The terminal window at the bottom shows the command `python3 a.py` being executed, resulting in the output:

```
rathin@laptop:~$ python3 a.py
hi
100
rathin@laptop:~$
```

Basic I/O

- Read user input: `input(prompt)`

Save the following into a file and execute the file:

```
n = input("enter a number: ")
```

```
n + 2
```

Basic I/O

- Read user input: `input(prompt)`

Save the following into a file and execute the file:

```
n = input("enter a number: ")
type(n)
n + 2
```

Basic I/O

- Read user input: `input(prompt)`

Save the following into a file and execute the file:

```
n = int( input("enter a number: ") ) # type casting
type(n)
n + 2
```

Basic I/O

- Read user input: `input(prompt)`

- Display output:

```
print(*objects, sep=' ', end='\n', file=None, flush=False)
```

Save the following into a file and execute the file:

```
n = int( input("enter a number: ") ) # type casting
# n + 2
# type(n)
print('you have entered', n, 'as n', end=' ')
print('and n+2 is', n+2)
```

Conditionals

- `if` statement is similar to C/C++/Java

```
if condition:  
    stmt_1  
    ⋮  
    stmt_n
```

Conditionals

- `if` statement is similar to C/C++/Java

```
if condition:  
    stmt_1  
    ⋮  
    stmt_n
```

- A block in Python is defined only using its indentation
 - A colon is there to declare the start of an indented block

Conditionals

- `if` statement is similar to C/C++/Java

```
if condition:  
    stmt_1  
    :  
    stmt_n
```

- A block in Python is defined only using its indentation
 - A colon is there to declare the start of an indented block
 - A block containing a single statement must also be indented

Conditionals

- `if` statement is similar to C/C++/Java

```
if condition:  
    stmt_1  
    ⋮  
    stmt_n
```

- A block in Python is defined only using its indentation
 - A colon is there to declare the start of an indented block
 - A block containing a single statement must also be indented
 - All statements in a block must have same indentation
(do not mix tab and spaces)

Conditionals

- `if` statement is similar to C/C++/Java

```
if condition:  
    stmt_1  
    :  
    stmt_n
```

- A block in Python is defined only using its indentation
 - A colon is there to declare the start of an indented block
 - A block containing a single statement must also be indented
 - All statements in a block must have same indentation
(do not mix tab and spaces)
 - Indentation implies start of a new block
thus we can not arbitrarily indent statements

Conditionals

- `if` statement is similar to C/C++/Java

```
if condition:
    stmt_1
    :
    stmt_n
```

- A block in Python is defined only using its indentation
 - A colon is there to declare the start of an indented block
 - A block containing a single statement must also be indented
 - All statements in a block must have same indentation (do not mix tab and spaces)
 - Indentation implies start of a new block
thus we can not arbitrarily indent statements

```
if b > a:
    print("b is greater than a")
```

Conditionals (contd.)

- Python also has `if...else` construct

```
if condition:
    stmt_1
    :
    stmt_n
else: # must have same indentation level as its 'if'
    stmt_1 # block may have different indentation
    :
    stmt_n
```

Conditionals (contd.)

- Python also has `if...else` construct

```
if condition:
    stmt_1
    :
    stmt_n
else: # must have same indentation level as its 'if'
    stmt_1 # block may have different indentation
    :
    stmt_n
```

- Python's way of doing ternary expressions:
`true_expr if condition else false_expr`

```
max_squared = a*a if a > b else b*b
```

Conditionals (contd.)

- Use `elif` statement to create a chain

```
if marks > 90:
    print('grade A')
elif marks > 80: # 90 >= marks 80
    print('grade B')
elif marks > 70: # 80 >= marks 70
    print('grade C')
elif marks > 60: # 70 >= marks 60
    print('grade D')
elif marks > 50: # 60 >= marks 50
    print('grade E')
else: # marks <= 50
    print('grade F')
```

- Conditionals can be nested as well

Conditionals (contd.)

- Use `elif` statement to create a chain

```
if marks > 90:
    print('grade A')
elif marks > 80: # 90 >= marks 80
    print('grade B')
elif marks > 70: # 80 >= marks 70
    print('grade C')
elif marks > 60: # 70 >= marks 60
    print('grade D')
elif marks > 50: # 60 >= marks 50
    print('grade E')
else: # marks <= 50
    print('grade F')
```

- Conditionals can be nested as well

Exercise: Write a program to detect leap-year

A Brain Teaser

Write a program to print the 'even'/'odd' status of a given integer without using any conditional statements

A Brain Teaser

Write a program to print the 'even'/'odd' status of a given integer without using any conditional statements

hint: use `n&1` as an index

Loops

- Python has two kinds of loop constructs
 - `while` loop to repeat a block based on some (termination) condition
 - `for` loop to iterate over a collection/iterable

```
while condition:
```

```
    stmt_1
```

```
    :
```

```
    stmt_n
```

```
for loop_var in iterable:
```

```
    stmt_1
```

```
    :
```

```
    stmt_n
```

- Loops can be nested
- There are `break` and `continue` statements as well

Example of Loops

```
n = int( input("enter an integer: ") )
a, b = 0, 1
while a < n:
    print(a, end=', ')
    a, b = b, a+b
print() # print a newline
```

Example of Loops

```
n = int( input("enter an integer: ") )
a, b = 0, 1
while a < n:
    print(a, end=', ')
    a, b = b, a+b
print() # print a newline
```

```
words = ['rathin', 'hello', 'book', 'the', 'fifth']
for w in words:
    if 'th' in w: # substring test
        print(w, len(w))
```

The `range()` function

```
range(stop) # [0, stop)
range(start, stop[, step])
```

The range() function

```
range(stop) # [0, stop)
range(start, stop[, step])
```

Print the followings:

```
list( range(10) )
list( range(1, 11) )
list( range(2, 11, 3) )
list( range(0, -10, -2) )
```

The range() function

```
range(stop) # [0, stop)
range(start, stop[, step])
```

Print the followings:

```
list( range(10) )
list( range(1, 11) )
list( range(2, 11, 3) )
list( range(0, -10, -2) )
```

Using range() function with for loop

```
for i in range(5):
    print('#' * i)
```


Using `else` with Loops

- Most often or not, we terminate a loop early (using `break`)
- And immediately outside the loop we test whether the loop has completed or terminated early

Using `else` with Loops

- Most often or not, we terminate a loop early (using `break`)
- And immediately outside the loop we test whether the loop has completed or terminated early
- The typical way of doing this is by introducing a *flag* variable or directly probe the iteration variable once again with respect to the loop (termination) condition

Using `else` with Loops

- Most often or not, we terminate a loop early (using `break`)
- And immediately outside the loop we test whether the loop has completed or terminated early
- The typical way of doing this is by introducing a *flag* variable or directly probe the iteration variable once again with respect to the loop (termination) condition
- The Python provides `else` block for (both) loops for this purpose
- `else` block is executed only if the loop has terminated normally

```
for i in range(10):
    print(i)
    # if i==6: # uncomment this
    #     break
else: # indented at the same level as for
    print('loop terminated normally')
```

An example

```
for i in range(2,n):
    if n % i == 0:
        print(n, 'equals', i, '*', n//i)
        break
    else: # loop fully executed
        print(n, 'is a prime number')
```

An example

```
for n in range(2,10):
    for i in range(2,n):
        if n % i == 0:
            print(n, 'equals', i, '*', n//i)
            break
        else: # loop fully executed
            print(n, 'is a prime number')
```

Defining Functions

```
def function_name(parameters):  
    stmt_1  
    :  
    stmt_n
```

Defining Functions

```
def function_name(parameters):  
    stmt_1  
    :  
    stmt_n
```

- There is no return type specified

Defining Functions

```
def function_name(parameters):  
    stmt_1  
    :  
    stmt_n
```

- There is no return type specified

```
def add(a, b):  
    return a + b  
  
.  
.  
.  
x = add(10, 20)
```


Defining Functions

```
def function_name(parameters):  
    stmt_1  
    :  
    stmt_n
```

- There is no return type specified

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

. . .

```
x = add(10, 20)
```

- Multiple values can be returned together

```
def add_sub(a, b):  
    return a + b, a - b
```

. . .

```
x, y = add_sub(10, 20)
```

Function Examples

```
import math
...
def dist(x1, y1, x2, y2):
    return math.sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) )
...
print( dist(1, 2, 1, 4) )
```

Function Examples

```
import math
...
def dist(x1, y1, x2, y2):
    return math.sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) )
...
print( dist(1, 2, 1, 4) )
```

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
...
print( is_prime(24) )
print( is_prime(49) )
```

Default Valued Arguments

- The function arguments can have some default values
- Default value is assumed when no value is given for that parameter

```
def foo(x = 'nothing'):  
    print('you have passed', x)  
  
...  
print( foo() )  
print( foo(123) )  
print( foo('abcd') )
```

Default Valued Arguments

- The function arguments can have some default values
- Default value is assumed when no value is given for that parameter

```
def foo(x = 'nothing'):  
    print('you have passed', x)  
  
...  
print( foo() )  
print( foo(123) )  
print( foo('abcd') )
```

- When there are multiple arguments the order is important

```
def add(x, y, z=0):  
    # print(x, y ,z) # uncomment this  
    return x + y + z  
  
...  
print( add(10, 20) )  
print( add(5, 6, 7) )
```