Python Basic

Chapter-1

Introduction of Python, Python Operators

Python Introduction

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.
- **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python's features

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.
-

Apart from the above-mentioned features, Python has a big list of good features, few are listed below −

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.

- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Data Types

List

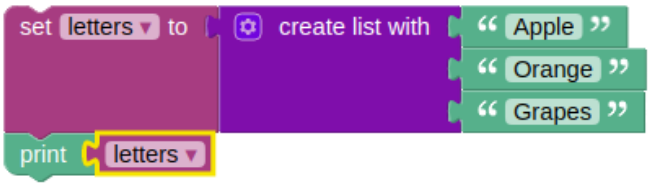
List Block & Text Programming

1. Create empty list

The simplest list is the empty list, which is created with the **create empty list Block:**

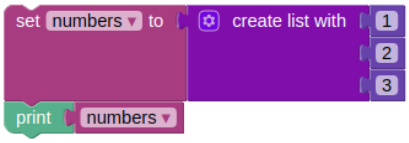
2. Create list with basic usage

The **create list with** block allows one to specify the initial values in a new list. In this example, a list of words is being created and placed in a variable named **letters**:
letters = ['Apple', 'Orange', 'Grapes']
print(letters)
Output : - ['Apple', 'Orange', 'Grapes']



3. This shows the creation of a list of numbers:

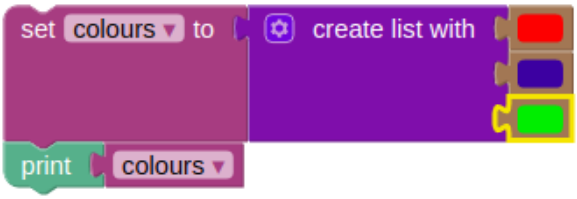
numbers = [1, 2, 3]
print(numbers)
Output: - [1, 2, 3]



4. This creates a list of colours:

colours = ['#ff0000', '#000099', '#33cc00']
print(colours)

Output: - ['#ff0000', '#000099', '#33cc00']



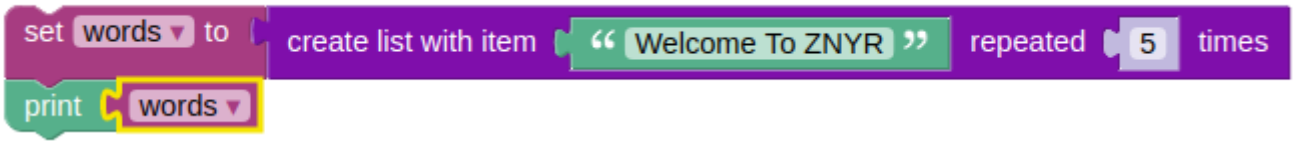
5. It is less common, but possible, to create a list with values of different types:

['ZNYR', 2021, '#33cc00']

6. **Create list with item**

The **create list with item** block lets you create a list that has the specified number of copies of an item. For example, the following blocks set the variable **words** to the list containing ['Welcome To ZNYR', 'Welcome To ZNYR', 'Welcome To ZNYR', 'Welcome To ZNYR', 'Welcome To ZNYR'].
words = ['Welcome To ZNYR'] * 5
print(words)

Output: - ['Welcome To ZNYR', 'Welcome To ZNYR', 'Welcome To ZNYR', 'Welcome To ZNYR', 'Welcome To ZNYR']



7. CHECKING A LIST'S LENGTH

is empty

The value of an **is empty** block is **true** if its input is the empty list and **false** if it is anything else (including a non-list). IS THIS TRUE? The value of the following blocks would be **false** because the variable **colours** is not empty: it has three items.

Note the similarity to the "is empty" block for text.

8. Length of

The value of the **length of** block is the number of elements in the list used as an input. For example, the value of the following blocks would be 3 because colour has three items.

words = [123, 'ZNYR', 123]
print(len(words))

Output:- 3

9. FINDING ITEMS IN A LIST

These blocks find the position of an item in a list. For example, the following has a value of 1 because the first appearance of "very" is as the beginning of the **words** list ("very", "very", "very")

words = ['very', 'very', 'very']

The result of the following is 3 because the last appearance of "very" in **words** is in position 3.

If the item is nowhere in the list, the result is in the value 0, as in this example:

These blocks are analogous to the ones for finding letters in text.

10. GETTING ITEMS FROM A LIST

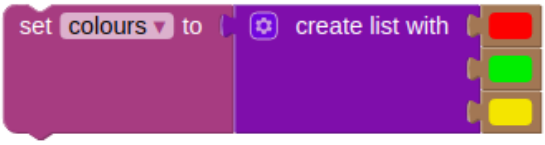
Getting a single item

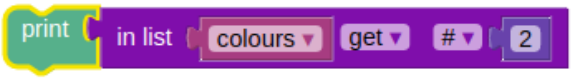
Recall the definition of the list **colours**:

colours = ['#ff0000', '#33cc00', '#ffcc00']

print(colours[1])

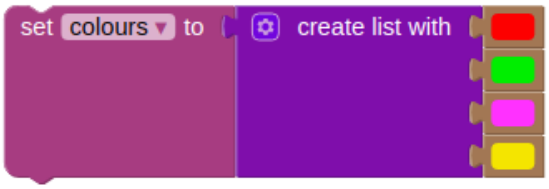
Output :- #33cc00





This gets green because it is the second element counting from the right end:

colours = ['#ff0000', '#33cc00', '#ff99ff', '#ffcc00']
print(colours[-2])
Output: - #ff99ff



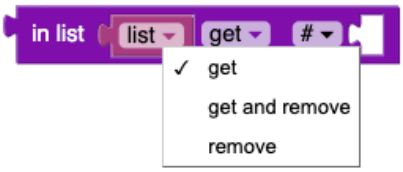
This gets the first element, red:

This gets the last element, yellow:

This randomly selects an item from the list, returning any of red, blue, green, or yellow with equal likelihood.

Getting and removing an item

A dropdown menu on the **in list ... get** block changes it to **in list ... get and remove**, which provides the same otuput but a modifies the original list:



This example sets the variable **first letter** to "alpha" and leaves **letters** as: ["beta", "gamma"].

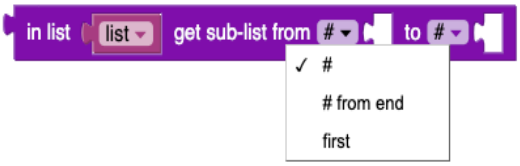
Removing an item

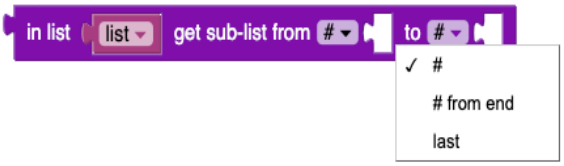
Selecting "remove" on the dropdown causes the plug on the left of the block to disappear:

This removes the first item from **letters**.

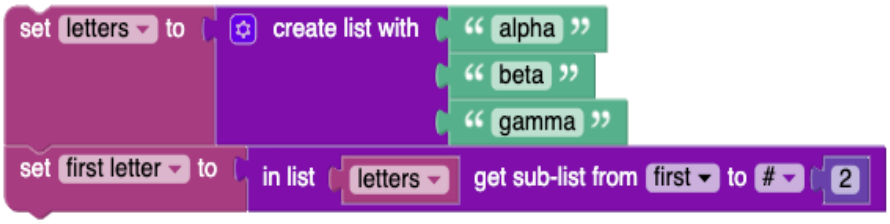
Getting a sublist

The **in list ... get sublist** block is similar to the **in list ... get** block except that it extracts a sublist, rather than an individual item. There are several options for how the start and end of the sublist can be specified:





In this example, a new list **first letters** is created. This new list has two elements: ["alpha", "beta"].

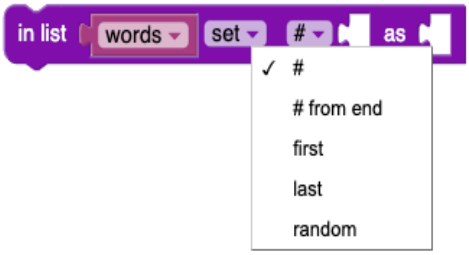


Note that this block does not modify the original list.

Adding Items to a List

in list ... set

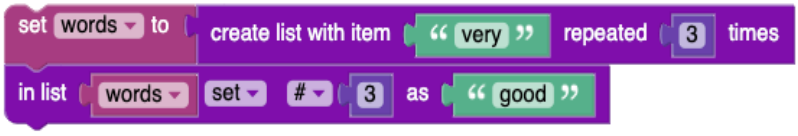
The **in list ... set** block replaces the item at a specified location in a list with a different item.



For the meaning of each of the dropdown options, see the previous section.

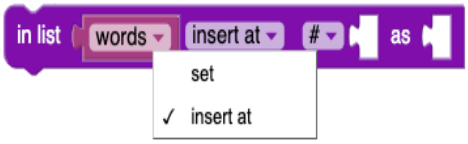
The following example does two things:

1. The list **words** is created with 3 items: ["very", "very", "very"].
2. The third item in the list is replaced by "good". The new value of **words** is ["very", "very", "good"].



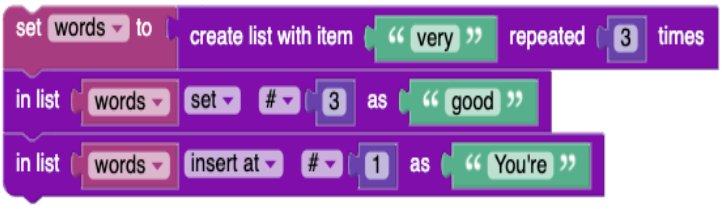
in list ... insert at

The **in list ... insert at** block is obtained by using the dropdown menu on the **in list ... set** block:



It inserts a new item into the list at the specified location, before the item previously at that location. The following example (built on an earlier one) does three things:

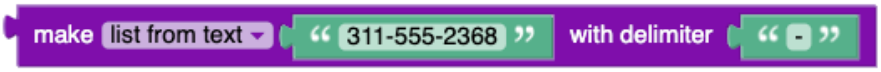
1. The list **words** is created with 3 items: ["very", "very", "very"].
2. The third item in the list is replaced by "good". The new value of **words** is ["very", "very", "good"].
3. The word "you're" is inserted at the beginning of the list. The final value of **words** is ["You're", "very", "very", "good"].



Splitting strings and joining lists

make list from text

The **make list from text** block splits the given text into pieces using a delimiter:



In the above example, a new list is returned containing three pieces of text: "311", "555", and "2368".

make text from list

The **make text from list** block joins a list into a single text using a delimiter:

In the above example, a new text is returned with the value: "311-555-2368".

JOIN LISTS

Join Two Lists
There are several ways to join, or concatenate, two or more lists in Python.
One of the easiest ways are by using the + operator.

Example

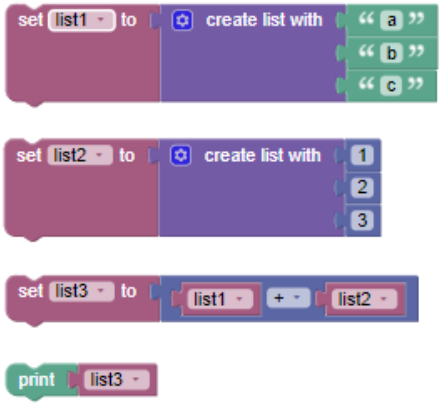
list1 = ["a", "b", "c"]

list2 = [1, 2, 3]

list3 = list1 + list2

print(list3)

Block **Output**



['a', 'b', 'c', 1, 2, 3]

List Methods

Python has a set of built-in methods that you can use on lists.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Variables

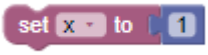
Variables and Text creations block & Programming

We use the term *variable* the same as it is used in mathematics and in other programming languages: a named value that can be changed (varies). Variables can be created in several different ways.

- Some blocks such as **count with** and **for each** use a variable and defines its values. A traditional computer science term for these are **loop variables**.
- User-defined functions (also known as "procedures") can define inputs, which creates variables that can be used only within the function. These are traditionally called "parameters" or "arguments".
- Users may create variables at any time through the "set" block. These are traditionally called "**global variables**". Blockly does not support **local variables**.

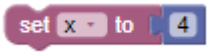
Create Variable

In Snap, we created variables by using the "Make a Variable" button. In Python, we don't explicitly declare a variable in code. Rather, variables are created when they are assigned to a certain value:



is the same as x = 1

We can change the value of x by setting it equal to a different value:



is the same as x = 4

Or by using different operators:

is the same as x = 5 + 4

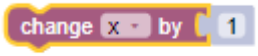
is the same as x = 5 - 4

is the same as x = 5 * 4

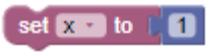
is the same as x = 5 % 4

is the same as x = 5 ^ 4

Another important block we used with variables was the

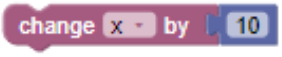


block. In Python, it is done a little differently. Notice that



are equivalent. Python follows the structure of this second block to change the value of a variable.

and

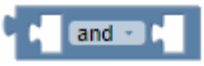
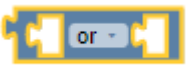


are the same as x = x + 10

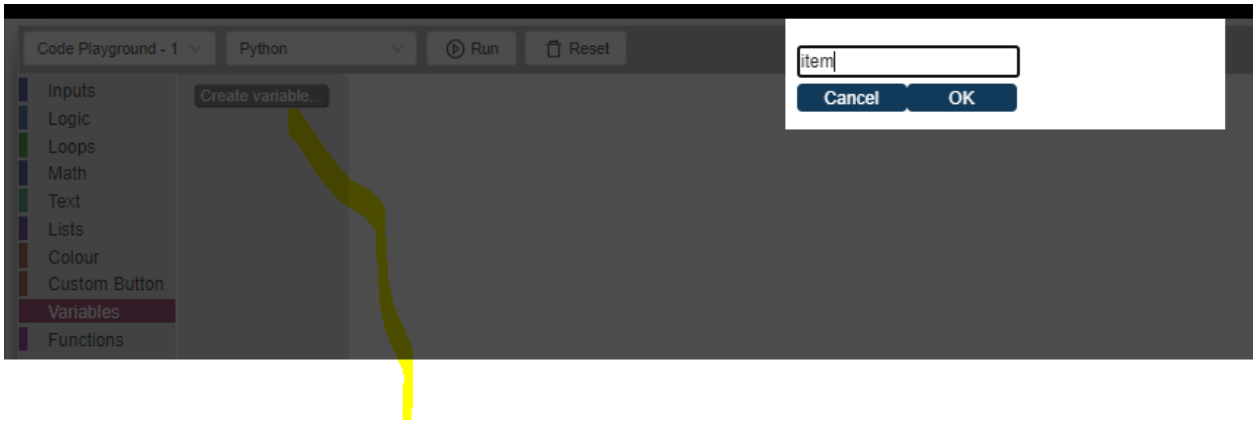
Here's a quick summary of many of the useful operators in Python shown side by side with their Snap equivalent. We can see that Python operators like greater than or equal to >= can save us a lot of time when writing our code, since in Snap we would have needed to drag out multiple blocks.

Function	**Snap**	**Python**
Addition		x + y
Subtraction		x - y
Multiplication		x * y
Modulo		x % y

Learn to Code Test Plan

Less Than		x < y
Greater Than		x > y
Equals		x = y
Not		not x
True and False		True and False
True or False		True or False

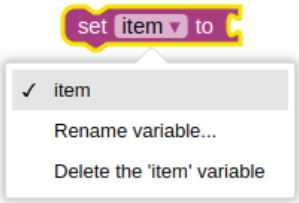
Below screen shown create a variable in block



Click and create variable

Dropdown menu

Clicking on a variable's dropdown symbol (triangle) gives the following menu:
item = 0



The menu provides the following options.

- the names of all existing variables defined in the program.
- "Rename variable...", changes the name of this variable wherever it appears in the program. Selecting this option opens a prompt for the new name.
- "Delete the variable...", deletes all blocks that reference this variable wherever it appears in the program.

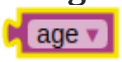
BLOCKS

SET

The **set** block assigns a value to a variable, creating the variable if it doesn't already exist. For example, this sets the value of the variable named "age" to 12.
age = 12

Get

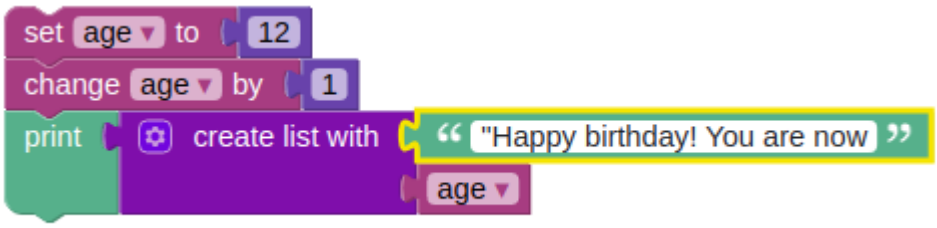
The **get** block provides the value stored in a variable, without changing it.



It is possible, but a bad idea, to write a program in which a **get** appears without a corresponding **set**.

Change

The **change** block adds a number to a variable.
age = 12
age = (age if isinstance(age, Number) else 0) + 1
print(['"Happy birthday! You are now', age])



Output:- ['"Happy birthday! You are now', 13]
The first row of blocks creates a variable named "age" and sets its initial value to the number 12. The second row of blocks gets the value 12, adds 1 to it, and stores the sum (13) into the variable. The final row displays the message: "Happy birthday! You are now 13"

Text creation

The following block creates the piece of text "hello" and stores it in the variable named greeting.

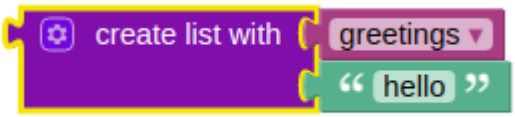
Example:-

greetings = 'hello'

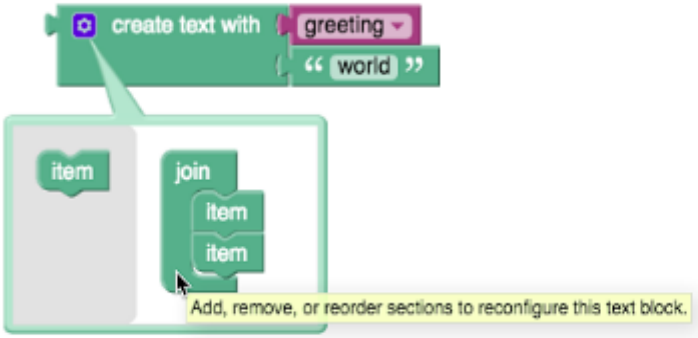
The **create text with** block combines (concatenates) the value of the greeting variable and the new text "world" to create the text "helloworld". Note that there is no space between them, since none was in either original text.

Example:-

[greetings, 'hello']



To increase the number of text inputs, click on the gear icon, which changes the view to:

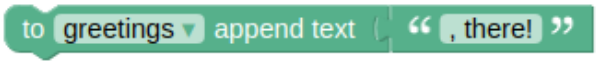


Additional inputs are added by dragging an "item" block from the gray toolbox on the left into the "join" block.

Text modification

The **to...append text** block adds the given text to the specified variable. In this case, it changes the value of the variable greeting from "hello" to "hello, there!"
Example :-

greetings = str(greetings) + ', there!'



Text length
The **length of** blocks count the number of letters, numbers, etc., in each text. The length of "ZNYR" is 9, and the length of the empty text is 0.
Example :-

len('ZNYR')

print(len('ZNYR'))

Output:- 4

Checking for empty text
The **is empty** block checks whether the given text is empty (has length 0). The result is **true** in the first case and **false** in the second.
Example:-

not len('')

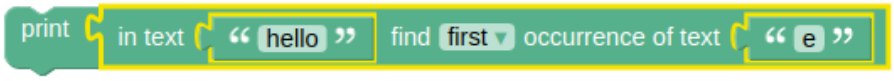
not len(greetings)

Finding text

These blocks can be used to check whether a piece of text is in another piece of text and, if so, where it appears. For example, this asks for the first occurrence of "e" in "hello". The result is 2.

Example:- 1

print('hello'.find('e') + 1)

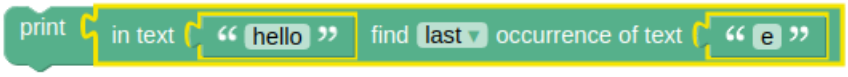


Output: - 2

Example: - 2

This asks for the *last* occurrence of "e" in "hello", which, is also 2.

print('hello'.rfind('e') + 1)



Output:- 2

Whether **first** or **last** is selected, this block will give the result 0, since "hello" does not contain "z".

Extracting text

Extracting a single character

This gets "N", the second letter in "ZNYR":

Example:- 1

print('ZNYR'[1])

Output:- N

This gets "Y", the second *to last* letter in "ZNYR":

print('ZNYR'[-2])

Output :- Y
This gets "a", the first letter in "ZNYR":

print('ZNYR'[0])

Output :- Z
This gets "e", the last letter in "ZNYR":

print('ZNYR'[-1])

Output :- R
This gets any of the 5 letters in "abcde" with equal probability:

Example:-
import random

```
def text_random_letter(text):

  x = int(random.random() * len(text))

  return text[x];

print(text_random_letter('ZNYR'))
```

Output:- Y

None of these modify the text on which the extraction is performed.

Extracting a region of text

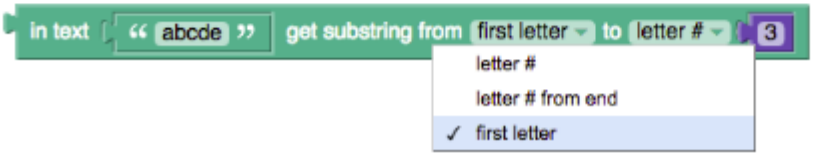
The **in text...get substring** block allows a region of text to be extracted, starting with either:

- letter #
- letter # from end
- the first letter

and ending with:

- letter #
- letter # from end
- the last letter

In the following example, "abc" is extracted.

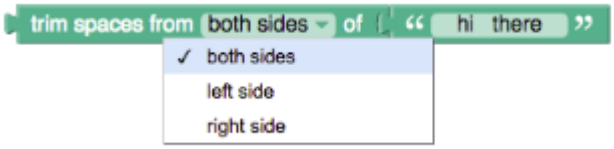


Trimming (removing) spaces

The following block removes space characters from:

- the beginning of the text
- the end of the text
- both sides of the text

The result of the following block is "hi there". (Spaces in the middle of the text are not affected.)



Adjusting text case

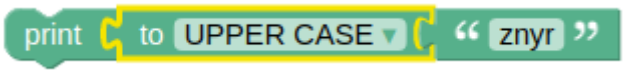
This block creates a version of the input text that is either:

- UPPER CASE (all letters upper-case)
- lower case
- Title Case (first letters upper-case, other letters lower-case)

The result of the following block is "ZNYR".

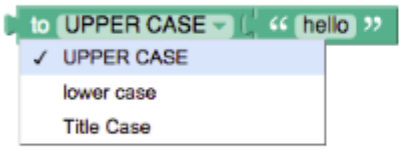
Example:-

print('znyr'.upper()



Output : - ZNYR

The result of the following block is "HELLO".



Non-alphabetic characters are not affected. Note that this block on text in languages without case, such as Chinese.

Printing text

The **print** block causes the input value to be displayed in a pop-up window, as shown:
print('hello znyr')

Output: - hello znyr

Getting input from the user

The following block creates a pop-up window that prompts the user to enter a name. The result is stored in the variable **name**:
name = None

```
def text_prompt(msg):
  try:
    return raw_input(msg)
  except NameError:
    return input(msg)
name = text_prompt('Enter your name:')
```

Output : - Enter your name:

Conditional Statements

Conditional Statement in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python. Conditional Statement in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

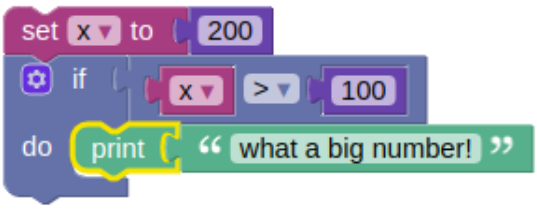
Python if Statement is used for decision-making operations. It contains a body of code which runs only when the condition given in the if statement is true. If the condition is false, then the optional else statement runs which contains some code for the else condition.

If blocks

The simplest conditional statement is an **if** block, as shown:

```
x = 200
if x > 100:

  print('what a big number!')
```

Output: - what a big number!



When run, this will compare the value of the variable **x** to 100. If it is larger, "What a big number!" will be printed. Otherwise, nothing happens.

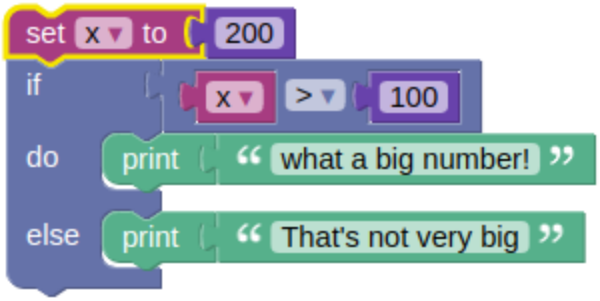
If-Else blocks

It is also possible to specify that something should happen if the condition is *not* true, as shown in this example:

```
x = 200
if x > 100:
```

```
  print('what a big number!')
else:
  print("That's not very big")
```

Output : - what a big number!

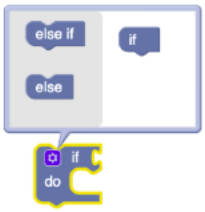


As with the previous block, "What a big number!" will be printed if **x** > 100; otherwise, "That's not very big." will be printed.
An **if** block may have zero or one **else** sections but not more than one.

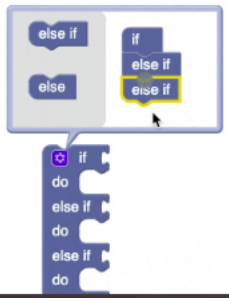
Block Modification

Only the plain **if** block appears in the toolbox:

To add **else if** and **else** clauses, the click on the gear icon, which opens a new window:



Drag **else if** and **else** clauses under the **if** block, as well as reordering and removing them. When finished, click on the gear icon, which closes the window, as shown here:



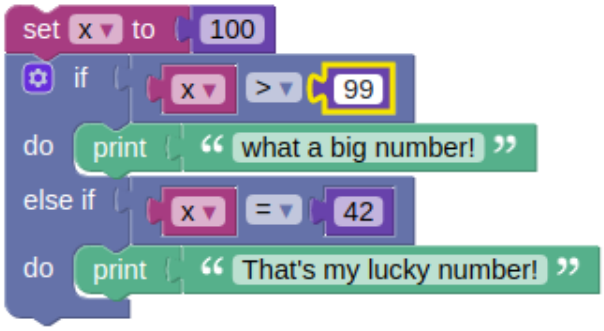
If-Else-If blocks

It is also possible to test multiple conditions with a single **if** block by adding **else if** clauses:

Example: - 1

```
x = 100
if x > 99:
  print('what a big number!')
elif x == 42:
  print("That's my lucky number!")
```

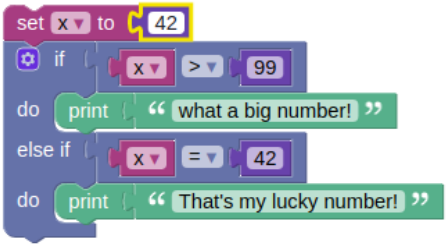
Output : - what a big number!



Example :- 2

```
x = 42

if x > 99:

  print('what a big number!')

elif x == 42:

  print("That's my lucky number!")
```

Output :- That's my lucky number!



The block first checks if **x** > 100, printing "What a big number!" if it is. If it is not, it goes on to check if **x** = 42. If so, it prints "That's my lucky number." Otherwise, nothing happens.

An **if** block may have any number of **else if** sections. Conditions are evaluated top to bottom until one is satisfied, or until no more conditions are left.

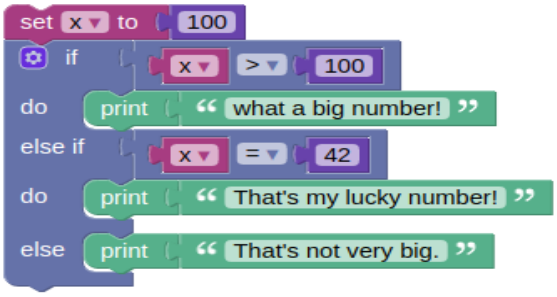
If-Else-If-Else blocks

As shown here, **if** blocks may have both **else if** and **else** sections:

Example :-

```
x = 100
if x > 100:
  print('what a big number!')
elif x == 42:
  print("That's my lucky number!")
else:
  print("That's not very big.")
```

Output :- That's not very big.



The **else** section guarantees that some action is performed, even if none of the prior conditions are true.
An **else** section may occur after any number of **else if** sections, including zero.

Chapter-2

Python Basics

LOOPS

Python has two primitive loop commands:

Loop Through a List

You can loop through the list items by using a for loop:

Example

Print all items in the list, one by one:
thislist = ["apple", "banana", "cherry"]

for x in thislist:

  print(x)

Block **Output**

apple
banana
cherry

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

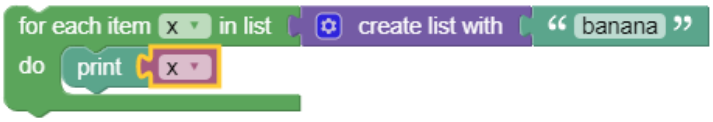
Example

Loop through the letters in the word "banana":

for x in ['banana']:

  print(x)

Block **Output**



banana

The count range() Function

To loop through a set of code a specified number of times, we can use the count range() function,

The count range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the count range() function:

```
for count in range(5):

  print('ZNYR')
```

Block

Output

```
ZNYR
ZNYR
ZNYR
ZNYR
ZNYR
```

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(1, 11), which means values from 1 to 11 (but not including 11):

Example1:

```
for i in range(1, 11):
  print(i)
```

Block

Output

1

2

3

4

5

6

7

8

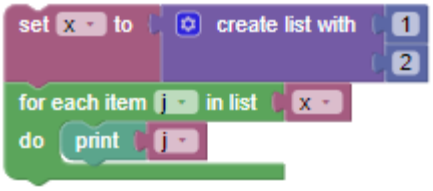
9

10

Example:2

```
x = [1, 2]
for j in x:
  print(j)
```

Block



Output

1
2

Functions

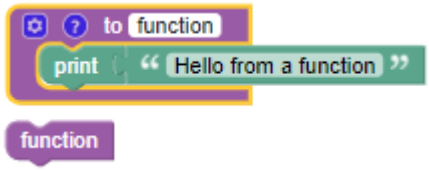
Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
# Describe this function...

def function():

  print('Hello from a function')

function()
```

Block



Output

Hello from a function

Return Values

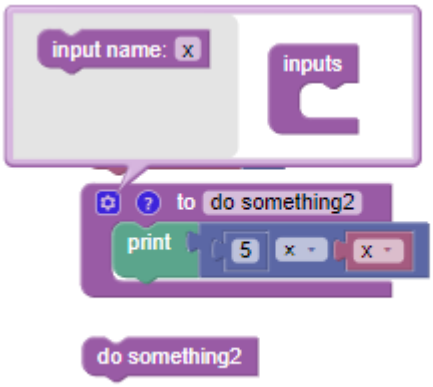
To let a function return a value, use the return statement:

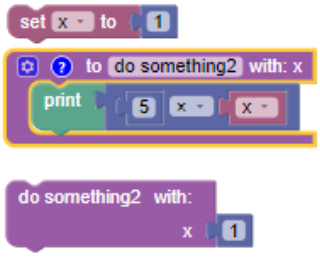
Example

```
# Describe this function...
def do_something2(x):
  print(5 * x)
x = 1
do_something2(1)
```

Block

Pass input values click setting


input name: x
inputs
to do something2
print
5
x
x
do something2


set x to 1
to do something2 with: x
print
5
x
x
do something2 with:
x
1

Output

5