**PSP**

**WEEK 2 &3 : Variables / Types / Control Statements**

**Hello, World!**

Python is a very simple language, and has a very straightforward syntax. It encourages programmers to program without boilerplate (prepared) code. The simplest directive in Python is the "print" directive - it simply prints out a line (and also includes a newline, unlike in C).

# print()

To print a string in Python 3, just write:

In

[1]:

In

[2]:

This line will be printed.

1

print

(

"This line will be printed."

)

print

(

1

)

1

1

# Indentation

Python uses indentation for blocks, instead of curly braces.

Both tabs and spaces are supported, but the standard indentation requires standard Python code to use four spaces. For example:

In

[3]:

x

**=**

1

**if**

x

**==**

2

:

*# indented four spaces*

print

(

"x is 1."

)

1

2

3

4

# Keywords

In [4]:

**import**

keyword

print

(

keyword

.

kwlist

)

print

(

len

(

keyword

.

kwlist

))

1

2

3

['False', 'None', 'True', '\_\_peg\_parser\_\_', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'de l', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'o r', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield'] 36

# input()

In

[5]:

In

[6]:

In

[7]:

Enter a number: 5

Out[6]:

int

5

my\_val

**=**

int

(

input

(

"Enter a number: "

)

)

type

(

my\_val

)

print

(

my\_val

)

1

1

1

# Variables and Types

In

[8]:

25

student\_marks

student\_marks

**=**

25

print

(

student\_marks

)

print

(

"student\_marks"

)

1

2

3

4

Python is completely object oriented, and not "statically typed". You do not need to declare variables before using them, or declare their type. Every variable in Python is an object.

This tutorial will go over a few basic types of variables.

**Native Data types:**

1. Numeric - Integers / Floating point / Complex Numbers / Boolean (True / False ) 2. Non numeric - String

## Numbers

Python supports two types of numbers - integers and floating point numbers. (It also supports complex numbers, which will not be explained in this tutorial).

To define an integer, use the following syntax:

In [9]:

myint

**=**

7

print

(

myint

)

print

(

type

(

myint

))

1

2

3

7

<class 'int'>

In [10]:

x

**=**

int

(

5.0

)

print

(

x

)

1

2

5

To define a floating point number, you may use one of the following notations:

In [11]:

myfloat

**=**

7.5

print

(

type

(

myfloat

))

print

(

myfloat

)

1

2

3

<class 'float'>

In

[12]:

In

[13]:

7.5

5.0

7

myfloat

**=**

float

(

5

)

print

(

myfloat

)

myvar

**=**

'7'

print

(

myvar

)

type

(

myvar

)

1

2

1

2

3

In

[14]:

In

[15]:

Out[13]:

str

Out[14]:

int

Out[15]:

2

y

**=**

int

(

myvar

)

type

(

y

)

a

**=**

4

b

**=**

2

c

**=**

a

**/**

b

int

(

c

)

1

2

1

2

3

4

[16]:

my\_value

**=**

5

print

(

my\_value

)

1

2

5

## Constants

[17]:

NUM

**=**

5

NUM

1

2

Out[17]: 5

Variables in Python are defined using snake case, whereas constants are defined using the uppercase!

Examples of Cases:

camelCase snake\_case UPPERCASE

lowercase Title Case

## Strings

In [18]:

hello

'hello'

mystring

**=**

'hello'

print

(

mystring

)

mystring

**=**

"hello"

mystring

1

2

1

2

In [19]:

Out[19]:

The difference between the two is that using double quotes makes it easy to include apostrophes (whereas these would terminate the string if using single quotes)

In [20]:

mystring

**=**

"Don't worry about apostrophes!"

print

(

mystring

)

1

2

Don't worry about apostrophes!

There are additional variations on defining strings that make it easier to include things such as carriage returns, backslashes and Unicode characters. These will be covered later.

Simple operators can be executed on numbers and strings:

In [21]: In [22]:

3

hello world

one

**=**

1

two

**=**

2

three

**=**

one

**+**

two

print

(

three

)

a

**=**

"hello "

b

**=**

"world"

c

**=**

a

**+**

b

print

(

c

)

1

2

3

4

1

2

3

4

Assignments can be done on more than one variable "simultaneously" on the same line like this

In [23]:

a

,

b

,

c

**=**

3

,

4

,

7

print

(

a

,

b

)

print

(

c

)

1

2

3

3 4

7

Mixing operators between numbers and strings is not supported:

[

]:

*# This will not work!*

one

**=**

1

two

**=**

2

hello

**=**

"hello"

print

(

one

**+**

two

**+**

hello

)

1

2

3

4

5

6

# Exercise

The target of this exercise is to create a string, an integer, and a floating point number. The string should be named mystring and should contain the word "hello". The floating point number should be named myfloat and should contain the number 10.0, and the integer should be named myint and should contain the number 20.

In

[25]:

*# change this code*

mystring

**=**

**None**

myfloat

**=**

**None**

myint

**=**

**None**

*# testing code*

**if**

mystring

**==**

"hello"

:

print

(

"String: %s"

**%**

mystring

)

**if**

isinstance

(

myfloat

,

float

)

**and**

myfloat

**==**

10.0

:

print

(

"Float: %f"

**%**

myfloat

)

**if**

isinstance

(

myint

,

int

)

**and**

myint

**==**

20

:

print

(

"Integer: %d"

**%**

myint

)

1

2

3

4

5

6

7

8

9

10

11

12

## Lists

Lists are very similar to arrays. They can contain any type of variable, and they can contain as many variables as you wish. Lists can also be iterated over in a very simple manner. Here is an example of how to build a list.

In [26]:

mylist

**=**

[

1

,

3

,

4

,

6

,

7

,

9

,

"hi"

,

"hello"

]

print

(

mylist

)

1

2

[1, 3, 4, 6, 7, 9, 'hi', 'hello']

Lists in Python have the following properties:

1. Ordered
2. Supports Heterogeneous values (although recommended to use where values are Homogeneous)
3. Iterable
4. Sequential
5. Size Mutable

## Accessing Items in a List

In

[27]:

1

3

print

(

mylist

[

0

])

print

(

mylist

[

1

])

print

(

mylist

[

2

])

print

(

mylist

[

3

])

print

(

mylist

[

4

])

print

(

mylist

[

5

])

1

2

3

4

5

6

4

6

7

9

In [28]:

value

**=**

mylist

[

1

]

print

(

value

)

1

2

3

### Some useful list methods

Here are some of the mostly used list methods. The remaining methods will be discussed in details in the upcoming sessions.

[29]:

new\_list

**=**

[

21

,

33

,

45

,

75

,

91

,

99

]

print

(

new\_list

)

1

2

[21, 33, 45, 75, 91, 99]

### append

[30]: [21, 33, 45, 75, 91, 99, 101]

new\_list

.

append

(

101

)

print

(

new\_list

)

1

2

### insert

In [31]:

new\_list

.

insert

(

1

,

107

)

print

(

new\_list

)

1

2

[21, 107, 33, 45, 75, 91, 99, 101]

### extend

In [32]:

new\_list

.

extend

([

200

,

201

,

202

])

print

(

new\_list

)

1

2

[21, 107, 33, 45, 75, 91, 99, 101, 200, 201, 202]

### pop

In [33]:

removed\_value

**=**

new\_list

.

pop

()

print

(

new\_list

)

print

(

removed\_value

)

1

2

3

[21, 107, 33, 45, 75, 91, 99, 101, 200, 201]

202

In [34]:

new\_list

.

pop

(

1

)

print

(

new\_list

)

1

2

[21, 33, 45, 75, 91, 99, 101, 200, 201]

### remove

In [35]:

new\_list

.

remove

(

101

)

print

(

new\_list

)

1

2

[21, 33, 45, 75, 91, 99, 200, 201]

### del

In

[36]:

In

[37]:

[1

, 4, 6, 7, 9, 'hi', 'hello'

]

**del**

mylist

[

1

]

print

(

mylist

)

1

1

## Indexing and Slicing

In [38]:

new\_list

**=**

[

21

,

33

,

45

,

75

,

91

,

99

]

1

**Indexing**

In [39]:

new\_list

[

2

]

1

Out[39]: 45

[40]: new\_list[**-**1]

Out[40]: 99

**Slicing**

Out[43]: [45, 75]

In

[41]:

In

[42]:

In

[43]:

In

[44]:

In

[45]:

In

[46]:

Out[41]:

[21

, 33,

45]

Out[42]:

, 45, 75, 91,

99]

[33

Out[45]:

99]

, 91,

[75

Out[46]:

21]

, 91, 75, 45, 33,

[99

new\_list

[:

3

]

new\_list

[

1

:]

new\_list

[

2

:

4

]

new\_list

[

1

:

5

:

2

]

*# slicing with a step size of two*

new\_list

[

**-**

3

:]

*# specifying position from the last*

new\_list

[::

**-**

1

]

*# reverse the list*

1

1

1

1

1

1

Out[44]: [33, 75]

# Exercise

**Exercise A**

In this exercise, you will need to add numbers and strings to the correct lists using the "append" list method. You must add the numbers 1,2, and 3 to the "numbers" list, and the words 'hello' and 'world' to the strings list.

You will also have to fill in the variable second\_name with the second name in the names list, using the brackets operator [] Note that the index is zerobased, so if you want to access the second item in the list, its index will be 1.

In

[47]:

[]

[]

numbers

**=**

[]

strings

**=**

[]

names

**=**

[

"John"

,

"Eric"

,

"Jessica"

]

*# write your code here*

second\_name

**=**

**None**

*# this code should write out the filled arrays and the second name in the names list (Eric).*

print

(

numbers

)

print

(

strings

)

print

(

"The second name on the names list is %s"

**%**

second\_name

)

1

2

3

4

5

6

7

8

9

10

11

The second name on the names list is None

**Exercise B**

1. Take input from keyboard (first input - Name, second input- Roll number)
2. Create a list, add the name and roll number to the list
3. Print the resulting list

# Basic Operators

## Arithmetic Operators

Just as any other programming languages, the addition, subtraction, multiplication, and division operators can be used with numbers.

Try to predict what the answer will be. Does python follow order of operations?

[48]:

number

**=**

(

1

**+**

2

)

**\***

3

**/**

4.0

**-**

2

print

(

number

)

1

2

0.25

Another operator available is the modulo (%) operator, which returns the integer remainder of the division. dividend % divisor = remainder .

In [49]:

remainder

**=**

11

**%**

3

print

(

remainder

)

1

2

2

Another equally useful operator is the floor division (//) operator, which returns an integer value of the resulting operation. Remember, it just truncates the value after the decimal point.

In [50]:

x

**=**

11

**//**

4

x

1

2

Out[50]: 2

Using two multiplication symbols makes a power relationship.

In

[51]:

In

[52]:

49

8

squared

**=**

7

**\*\***

2

cubed

**=**

2

**\*\***

3

print

(

squared

)

print

(

cubed

)

1

2

1

2

# Using Operators with Strings

Python supports concatenating strings using the addition operator:

In [53]:

x

**=**

"hello "

**+**

"world"

print

(

x

)

1

2

hello world

Python also supports multiplying strings to form a string with a repeating sequence:

In [54]:

lotsofhellos

**=**

"hello "

**\***

10

print

(

lotsofhellos

)

1

2

hello hello hello hello hello hello hello hello hello hello

# Using Operators with Lists

Lists can be joined with the addition operators:

In

[55]:

, 3, 5, 7, 2, 4, 6,

[1

8]

even\_numbers

**=**

[

2

,

4

,

6

,

8

]

odd\_numbers

**=**

[

1

,

3

,

5

,

7

]

all\_numbers

**=**

odd\_numbers

**+**

even\_numbers

print

(

all\_numbers

)

1

2

3

4

Just as in strings, Python supports forming new lists with a repeating sequence using the multiplication operator:

[56]: print([1,2,3] **\*** 3)

[1, 2, 3, 1, 2, 3, 1, 2, 3]

# Exercise

The target of this exercise is to create two lists called x\_list and y\_list , which contain 10 instances of the variables x and y , respectively. You are also required to create a list called big\_list , which contains the variables x and y , 10 times each, by concatenating the two lists you have created.

In

[57]:

x\_list contains 0 objects

y\_list contains 0 objects

x

**=**

7

y

**=**

5

*# TODO: change this code*

x\_list

**=**

[]

y\_list

**=**

[]

big\_list

**=**

[]

print

(

"x\_list contains %d objects"

**%**

len

(

x\_list

))

print

(

"y\_list contains %d objects"

**%**

len

(

y\_list

))

print

(

"big\_list contains %d objects"

**%**

len

(

big\_list

))

*# testing code*

**if**

x\_list

.

count

(

x

)

**==**

10

**and**

y\_list

.

count

(

y

)

**==**

10

:

print

(

"Almost there..."

)

**if**

big\_list

.

count

(

x

)

**==**

10

**and**

big\_list

.

count

(

y

)

**==**

10

:

print

(

"Great!"

)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

big\_list contains 0 objects

# String Formatting

Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d". Let's say you have a variable called "name" with your user name in it, and you would then like to print(out a greeting to that user.)

In

[58]:

In

[59]:

Hello, John!

Enter your name: Joe

Hello, Joe! Welcome to TBC! You scored 7.889

*# This prints out "Hello, John!"*

name

**=**

"John"

print

(

"Hello, %s!"

**%**

name

)

*# This prints out "Hello, John!"*

score

**=**

7.8886564567

name

**=**

input

(

"Enter your name: "

)

print

(

"Hello, %s! Welcome to TBC! You scored %.3f"

**%**

(

name

,

score

))

1

2

3

1

2

3

4

To use two or more argument specifiers, use a tuple (parentheses):

In

[60]:

John is 23 years old.

*# This prints out "John is 23 years old."*

name

**=**

"John"

age

**=**

23

print

(

"%s is %d years old."

**%**

(

name

,

age

))

1

2

3

4

Any object which is not a string can be formatted using the %s operator as well. The string which returns from the "repr" method of that object is formatted as the string. For example:

In [61]:

*# This prints out: A list: [1, 2, 3]*

mylist

**=**

[

1

,

2

,

3

]

print

(

"A list: %s"

**%**

mylist

)

1

2

3

A list: [1, 2, 3]

Here are some basic argument specifiers you should know:

%s - String (or any object with a string representation, like numbers)

%d - Integers

%f - Floating point numbers %.<number of digits>f - Floating point numbers with a fixed amount of digits to the right of the dot.

%x/%X - Integers in hex representation (lowercase/uppercase)

## f-string literals and the format method

In addition to C-style formatting, Python supports its own ways to format strings. A quick example is shown below. The details of these methods will be discussed in the later sessions.

In

[62]:

name

**=**

"Daniel"

college

**=**

"The British College"

classtime

**=**

1100

*# Older C-style formatting*

print

(

"Hello %s. Welcome to %s. The class starts at %d hrs."

**%**

(

name

,

college

,

classtime

))

*# Using Python's format method*

print

(

"Hello {}. Welcome to {}. The class starts at {} hrs."

.

format

(

name

,

college

,

classtime

))

*# Using Python's f-string literal*

print

(

f"Hello

{

name

}

. Welcome to

{

college

}

. The class starts at

{

classtime

}

hrs."

)

1

2

3

4

5

6

7

8

9

10

11

12

Hello Daniel. Welcome to The British College. The class starts at 1100 hrs.

Hello Daniel. Welcome to The British College. The class starts at 1100 hrs. Hello Daniel. Welcome to The British College. The class starts at 1100 hrs.

# Exercise

You will need to write a format string which prints out the data using the following syntax: Hello John Doe. Your current balance is $53.44.

**Option A**

In

[63]:

Hello John Doe.Your current balance is $53.44.

data

**=**

(

"John"

,

"Doe"

,

53.44

)

format\_string

**=**

"Hello %s %s.Your current balance is $%.2f."

**%**

data

print

(

format\_string

)

1

2

3

4

**Option B**

In [64]:

data

**=**

(

"John"

,

"Doe"

,

53.44

)

format\_string

**=**

f"Hello

{

data

[

0

]}

{

data

[

1

]}

. Your current balance is $

{

data

[

2

]}

."

print

(

format\_string

)

1

2

3

Hello John Doe. Your current balance is $53.44.

**Option C**

In

[65]:

Hello John Doe. Your current balance is $53.44.

data

**=**

(

"John"

,

"Doe"

,

53.44

)

format\_string

**=**

"Hello {} {}. Your current balance is ${}."

.

format

(

**\***

data

)

print

(

format\_string

)

1

2

3

4

# Basic String Operations

Strings are bits of text. They can be defined as anything between quotes:

|  |  |
| --- | --- |
| [66]:  In [67]:  Out[67]:  In [68]: | astring **=** "Hello world!"  str  As you can see, the first thing you learned was printing a simple sentence. This sentence was stored by Python as a string. However, instead of immediately printing strings out, we will explore the various things you can do to them. You can also use single quotes to assign a string. However, you will face problems if the value to be assigned itself contains single quotes.For example to assign the string in these bracket(single quotes are ' ') you need to use double quotes only like this  12  That prints out 12, because "Hello world!" is 12 characters long, including punctuation and spaces. |

In [69]:

type

(

astring

)

1

print

(

len

(

astring

))

1

astring

**=**

"hello world!"

print

(

astring

.

index

(

"o"

))

1

2

4

That prints out 4, because the location of the first occurrence of the letter "o" is 4 characters away from the first character. Notice how there are actually two o's in the phrase - this method only recognizes the first.

But why didn't it print out 5? Isn't "o" the fifth character in the string? To make things more simple, Python (and most other programming languages) start things at 0 instead of 1. So the index of "o" is 4.

In [70]:

astring

**=**

"Hello world!"

print

(

astring

.

count

(

"l"

))

1

2

3

For those of you using silly fonts, that is a lowercase L, not a number one. This counts the number of l's in the string. Therefore, it should print 3.

In

[71]:

In

[72]:

In

[73]:

Out[71]:

'tHIS name!'

Out[72]:

'this name!'

mystr

**=**

"This NAME!"

mystr

.

swapcase

()

mystr

.

lower

()

astring

**=**

"Hello world!"

print

(

astring

)

print

(

astring

[

3

:

7

])

1

2

1

1

2

3

Hello world! lo w

This prints a slice of the string, starting at index 3, and ending at index 6. But why 6 and not 7? Again, most programming languages do this - it makes doing math inside those brackets easier.

If you just have one number in the brackets, it will give you the single character at that index. If you leave out the first number but keep the colon, it will give you a slice from the start to the number you left in. If you leave out the second number, if will give you a slice from the first number to the end.

You can even put negative numbers inside the brackets. They are an easy way of starting at the end of the string instead of the beginning. This way, -3 means "3rd character from the end".

In [74]:

astring

**=**

"Hello world!"

print

(

astring

[

**-**

3

::

2

])

1

2

l!

This prints the characters of string from 3 to 7 skipping one character. This is extended slice syntax. The general form is [start:stop:step].

In [75]:

astring

**=**

"Hello world!"

print

(

astring

[::

**-**

1

])

1

2

!dlrow olleH

There is no function like strrev in C to reverse a string. But with the above mentioned type of slice syntax you can easily reverse a string like this.

[76]:

astring

**=**

"Hello world!"

print

(

astring

.

upper

())

print

(

astring

.

lower

())

1

2

3

HELLO WORLD! hello world!

These make a new string with all letters converted to uppercase and lowercase, respectively.

In [77]:

astring

**=**

"Hello world!"

print

(

astring

.

startswith

(

"Hello"

))

print

(

astring

.

endswith

(

"asdfasdfasdf"

))

1

2

3

True

False

This is used to determine whether the string starts with something or ends with something, respectively. The first one will print True, as the string starts with "Hello". The second one will print False, as the string certainly does not end with "asdfasdfasdf".

In [78]:

astring

**=**

"Hello world!"

afewwords

**=**

astring

.

split

(

" "

)

print

(

afewwords

)

1

2

3

['Hello', 'world!']

This splits the string into a bunch of strings grouped together in a list. Since this example splits at a space, the first item in the list will be "Hello", and the second will be "world!".

# Conditions

Python uses boolean variables to evaluate conditions. The boolean values True and False are returned when an expression is compared or evaluated. For example:

In

[79]:

True

False

if condition:

do sth

elif condition2:

do sth else

else:

do sth

x

**=**

2

print

(

x

**==**

2

)

*# prints out True*

print

(

x

**==**

3

)

*# prints out False*

print

(

x

**<**

3

)

*# prints out True*

1

2

3

4

5

6

1

2

3

4

True

In [80]:

**if**

(

x

**==**

2

):

print

(

"Yay!"

)

1

2

Yay!

Notice that variable assignment is done using a single equals operator "=", whereas comparison between two variables is done using the double equals operator "==". The "not equals" operator is marked as "!=".

In [81]:

(

1

**>=**

3

)

**and**

(

4

**>**

2

)

1

Out[81]: False

# Boolean operators / logical operators

The "and" and "or" boolean operators allow building complex boolean expressions, for example:

[82]:

Your name is John, and you are also 23 years old.

Your name is either John or Rick.

name

**=**

"John"

age

**=**

23

**if**

name

**==**

"John"

**and**

age

**==**

23

:

print

(

"Your name is John, and you are also 23 years old."

)

**if**

name

**==**

"John"

**or**

name

**==**

"Rick"

:

print

(

"Your name is either John or Rick."

)

1

2

3

4

5

6

7

# The "in" operator

The "in" operator could be used to check if a specified object exists within an iterable object container, such as a list:

In [83]:

Out[83]: True In [84]:

Your name is either John or Rick.

x

**=**

"abc"

x

**in**

"abcdef"

name

**=**

"John"

**if**

name

**in**

[

"John"

,

"Rick"

]:

print

(

"Your name is either John or Rick."

)

1

2

1

2

3

Python uses indentation to define code blocks, instead of brackets. The standard Python indentation is 4 spaces, although tabs and any other space size will work, as long as it is consistent. Notice that code blocks do not need any termination.

Here is an example for using Python's "if" statement using code blocks:

if <statement is="" true="">:

<do something="">

....

....

elif <another statement="" is="" true="">: # else if

<do something="" else="">

....

....

else:

<do another="" thing="">

....

....

<

/do></do></another></do></statement

>

1

2

3

4

5

6

7

8

9

10

11

12

13

For example:

In [85]:

x equals two!

x

**=**

4

**\***

0.5

**if**

x

**==**

2

:

print

(

"x equals two!"

)

**else**

:

print

(

"x does not equal to two."

)

1

2

3

4

5

6

A statement is evaulated as true if one of the following is correct:

1. The "True" boolean variable is given, or calculated using an expression, such as an arithmetic comparison.
2. An object which is not considered "empty" is passed.

Here are some examples for objects which are considered as empty:

1. An empty string: ""
2. An empty list: []
3. The number zero: 0
4. The false boolean variable: False

[86]:

5]

, 2, 3, 4,

[1

odd\_list

**=**

[

1

,

2

,

3

,

4

,

5

]

even\_list

**=**

[]

**if**

odd\_list

:

print

(

odd\_list

)

**if**

even\_list

:

print

(

even\_list

)

1

2

3

4

5

6

7

## Ternary Operator

In

[87]:

In

[88]:

even

odd

number\_type

**=**

'odd'

**if**

x

**%**

2

**!=**

0

**else**

'even'

print

(

number\_type

)

number\_type

**=**

'odd'

**if**

x

**%**

2

**==**

0

**else**

'even'

print

(

number\_type

)

1

2

1

2

# The 'is' operator

Unlike the double equals operator "==", the "is" operator does not match the values of the variables, but the instances themselves. For example:

In

[89]:

True

False

x

**=**

[

1

,

2

,

3

]

y

**=**

[

1

,

2

,

3

]

print

(

x

**==**

y

)

*# Prints out True*

print

(

x

**is**

y

)

*# Prints out False*

1

2

3

4

In

[90]:

In

[91]:

In

[92]:

In

[93]:

In

[94]:

Out[91]:

7]

, 2, 3, 5,

[1

Out[92]:

, 5, 3, 2,

[7

1]

Out[93]:

7]

, 2, 3, 5,

[1

Out[94]:

False

num

**=**

[

1

,

2

,

3

,

5

,

7

]

num\_copy

**=**

num

.

copy

()

num\_copy

num\_copy

.

sort

(

reverse

**=**

**True**

)

num\_copy

num

num

**is**

num\_copy

1

2

1

1

2

1

1

# The "not" operator

Using "not" before a boolean expression inverts it:

In [95]:

print

(

**not**

**False**

)

*# Prints out True*

print

((

**not**

**False**

)

**==**

(

**False**

))

*# Prints out False*

1

2

True

False

# Exercise

Change the variables in the first section, so that each if statement resolves as True.

[96]:

*# change this code*

number

**=**

1

second\_number

**=**

5

first\_array

**=**

[]

second\_array

**=**

[]

**if**

number

**>**

15

:

print

(

"1"

)

**if**

first\_array

:

print

(

"2"

)

**if**

len

(

second\_array

)

**==**

2

:

print

(

"3"

)

**if**

len

(

first\_array

)

**+**

len

(

second\_array

)

**==**

5

:

print

(

"4"

)

**if**

first\_array

**and**

first\_array

[

0

]

**==**

1

:

print

(

"5"

)

**if**

**not**

second\_number

:

print

(

"6"

)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

# Loops

There are two types of loops in Python, for and while.

## The "for" loop

For loops iterate over a given sequence. Here is an example:

In

[97]:

2

3

primes

**=**

[

2

,

3

,

5

,

7

,

9

]

**for**

x

**in**

primes

:

print

(

x

)

1

2

3

4

5

7

9

In [98]:

**for**

x

**in**

range

(

len

(

primes

)):

print

(

primes

[

x

])

1

2

2

3

5

7

9

For loops can iterate over a sequence of numbers using the "range" functions. Note that the range function is zero based.

In [99]:

*# Prints out the numbers 0,1,2,3,4*

**for**

x

**in**

range

(

5

):

print

(

x

)

1

2

3

0

1

2

3

4

In [100]:

*# Prints out 3,4,5*

**for**

x

**in**

range

(

3

,

6

):

print

(

x

)

1

2

3

3

4

5

[101]:

*# Prints out 3,5,7*

**for**

x

**in**

range

(

3

,

8

,

2

):

print

(

x

)

1

2

3

3

5

7

## "while" loops

While loops repeat as long as a certain boolean condition is met. For example:

In

[102]:

0

1

*# Prints out 0,1,2,3,4*

marks

**=**

[

30

,

25

,

28

,

35

,

32

]

count

**=**

0

**while**

count

**<**

len

(

marks

):

print

(

count

)

count

**+=**

1

*# This is the same as count = count + 1*

1

2

3

4

5

6

7

2

3

In

[103]:

4

30

25

*# Prints out the actual marks*

marks

**=**

[

30

,

25

,

28

,

35

,

32

]

count

**=**

0

**while**

**True**

:

**if**

count

**>=**

len

(

marks

):

**break**

print

(

marks

[

count

])

count

**+=**

1

*# This is the same as count = count + 1*

1

2

3

4

5

6

7

8

28

35

32

## "break" and "continue" statements

**break** is used to exit a for loop or a while loop, whereas **continue** is used to skip the current block, and return to the "for" or "while" statement. A few examples:

In

[104]:

0

1

*# Prints out 0,1,2,3,4*

count

**=**

0

**while**

**True**

:

**if**

count

**>=**

5

:

**break**

print

(

count

)

count

**+=**

1

1

2

3

4

5

6

7

8

2

3

In

[105]:

4

50

55

marks

**=**

[

50

,

55

,

70

,

"F"

,

33

]

**for**

i

**in**

marks

:

**if**

i

**==**

"F"

:

**break**

print

(

i

)

1

2

3

4

5

70

[106]:

1

3

**for**

x

**in**

range

(

10

):

*# Check if x is even*

**if**

x

**%**

2

**==**

0

:

**continue**

print

(

x

)

1

2

3

4

5

5

7

In

[107]:

9

1

Even

**for**

x

**in**

range

(

1

,

11

):

*# Check if x is even*

**if**

x

**%**

2

**==**

0

:

print

(

"Even"

)

**continue**

print

(

x

)

1

2

3

4

5

6

7

3

Even

5

Even

7

Even

9

Even

**can we use "else" clause for loops?**

Unlike languages like C,CPP.. we can use **else** for loops. When the loop condition of "for" or "while" statement fails then code part in "else" is executed. If **break** statement is executed inside for loop then the "else" part is skipped. Note that "else" part is executed even if there is a **continue** statement.

Here are a few examples:

In

[108]:

0

1

count

**=**

0

**while**

(

count

**<**

5

):

print

(

count

)

count

**+=**

1

**else**

:

print

(

"count value reached %d"

**%**

(

count

))

1

2

3

4

5

6

7

2

3

4

In

[109]:

count value reached 5

1

2

**for**

i

**in**

range

(

1

,

10

):

*# print("input: ",i)*

**if**

(

i

**%**

5

**==**

0

):

**break**

print

(

i

)

**else**

:

print

(

"this is not printed because for loop is terminated because of break but not due to fail in condition"

)

1

2

3

4

5

6

7

3

4