python™

By: Ahmed El-Mahdy

FB: bit.ly/2Utbjkz

LNKD: bitlylink.com/ooLxn

LECTURE 1

- Python history
- Installation
- Basic Syntax
- Variables
- Numbers
- Strings
- Operators
- Control Flow
- Loop

PYTHON HISTORY

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in the December 1989 by Guido Van Rossum at CWI in Netherland.
- In February 1991, van Rossum published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like: lambda, map, filter, and reduce.
- Python 2.0 added new features like: list comprehensions, garbage collection system.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.
- Python is influenced by following programming languages:
  - ABC language.
  - Modula-3

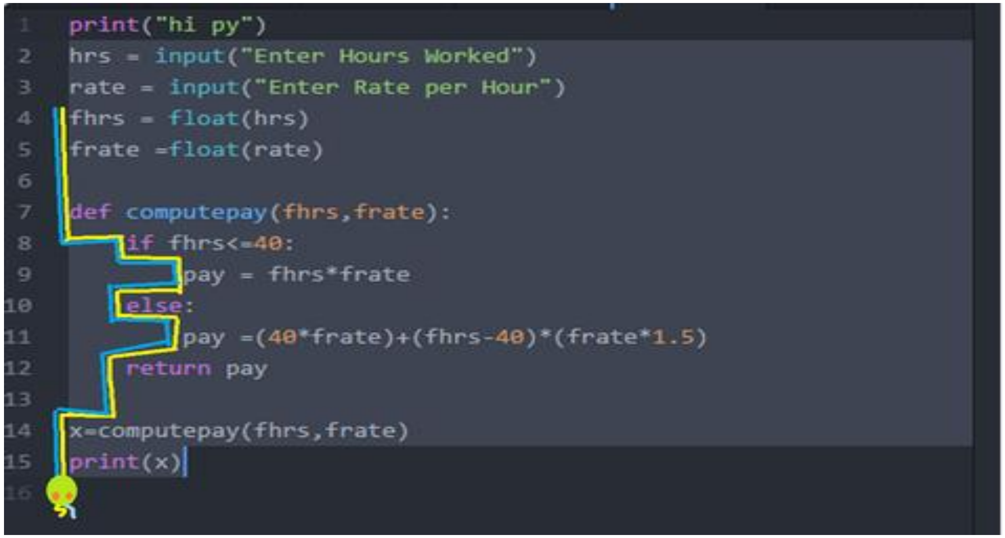
PYTHON HISTORY

- Amoeba is a distributed operating system developed by Andrew S. Tanenbaum and others at the Vrije Universiteit Amsterdam.
- The aim of the Amoeba project was to build a timesharing system that makes an entire network of computers appear to the user as a single machine.
- Development at the Vrije Universiteit was stopped: the source code of the latest version (5.3) was last modified on 30 July 1996.

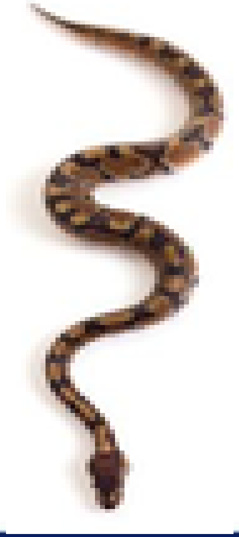
PYTHON HISTORY

Why python called python ?!

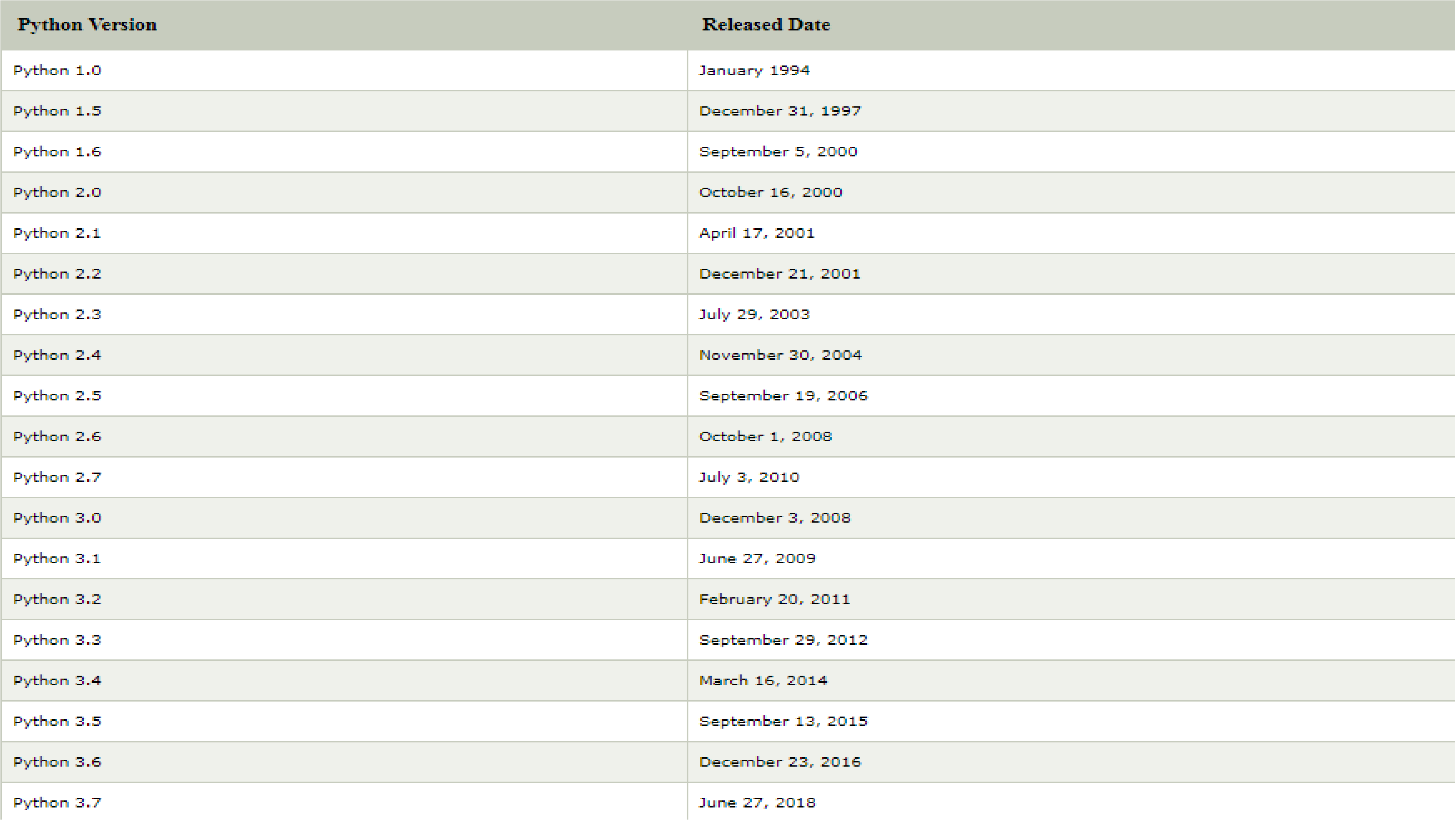
- At the time when he began implementing Python, Guido van Rossum was also reading the published scripts from "Monty Python's Flying Circus" (a BBC comedy series from the seventies, in the unlikely case you didn't know). It occurred to him that he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python."



```
print("hi py")
hrs = input("Enter Hours Worked")
rate = input("Enter Rate per Hour")
fhrs = float(hrs)
frate =float(rate)

def computepay(fhrs,frate):
    if fhrs<=40:
        pay = fhrs*frate
    else:
        pay =(40*frate)+(fhrs-40)*(frate*1.5)
    return pay

x=computepay(fhrs,frate)
print(x)
```



PYTHON HISTORY



Python Version	Released Date
Python 1.0	January 1994
Python 1.5	December 31, 1997
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000
Python 2.1	April 17, 2001
Python 2.2	December 21, 2001
Python 2.3	July 29, 2003
Python 2.4	November 30, 2004
Python 2.5	September 19, 2006
Python 2.6	October 1, 2008
Python 2.7	July 3, 2010
Python 3.0	December 3, 2008
Python 3.1	June 27, 2009
Python 3.2	February 20, 2011
Python 3.3	September 29, 2012
Python 3.4	March 16, 2014
Python 3.5	September 13, 2015
Python 3.6	December 23, 2016
Python 3.7	June 27, 2018

PYTHON HISTORY

Which python ?!

Python 3.7.3

Why python?!

- Simple syntax – easy for beginners – strong for professionals
- Clean code enforcement through indentations
- Cross platform running every where (Windows – Linux – Mac)
- Many libraries and modules to import and use
- Dynamic & Slow
- Large and supportive helpful community

PYTHON HISTORY

WHO IS USING PYTHON?!

- Instagram.
- Google.
- Spotify.
- Netflix.
- Reddit

- Uber.
- Dropbox.
- Pinterest.
- Instacar
- Twitter
- And countless others ….

INSTALLATION

Installation on Ubuntu

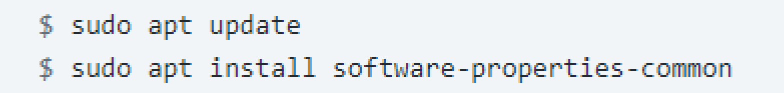
Python is already installed on Ubuntu

IF NOT !

INSTALLATION

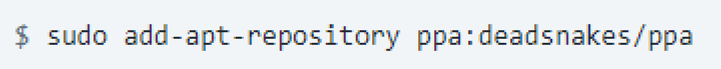
Installing Python 3.7 on Ubuntu with Apt

- Step 1: updating the packages list and installing the prerequisites

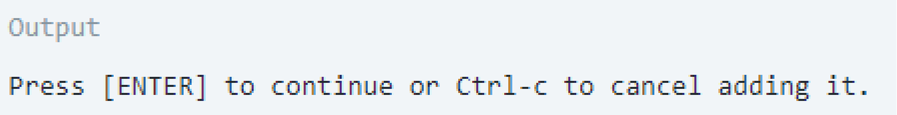


```
$ sudo apt update
$ sudo apt install software-properties-common
```

- Step 2: add the deadsnakes PPA to your sources list:

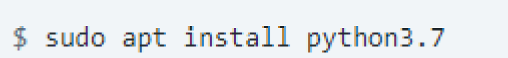


```
$ sudo add-apt-repository ppa:deadsnakes/ppa
```



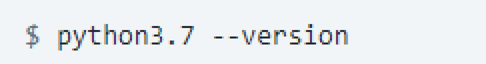
```
Output

Press [ENTER] to continue or Ctrl-c to cancel adding it.
```

- Step 3: Once the repository is enabled, install Python 3.7 with:

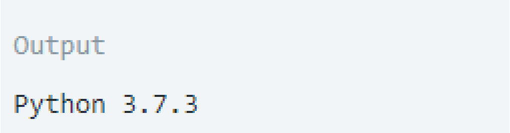


```
$ sudo apt install python3.7
```

- Step 4: verify it by typing:



```
$ python3.7 --version
```

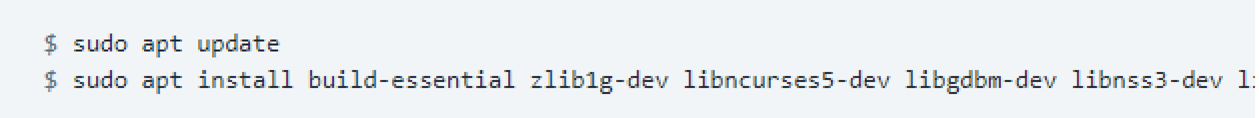


```
Output

Python 3.7.3
```

INSTALLATION

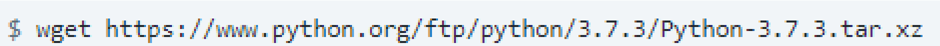
Installing Python 3.7 on Ubuntu from source

- Step 1: update the packages list and install the packages necessary to build Python source:



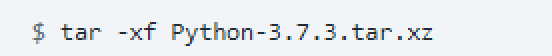
```
$ sudo apt update
$ sudo apt install build-essential zlib1g-dev libncurses5-dev libgdbm-dev libnss3-dev l
```

- Step 2: Download the latest release's source code from the Python download page using the following wget command:



```
$ wget https://www.python.org/ftp/python/3.7.3/Python-3.7.3.tar.xz
```

- Step 3:Once download is complete, extract the tarball:

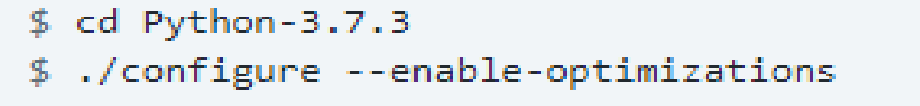


```
$ tar -xf Python-3.7.3.tar.xz
```

INSTALLATION

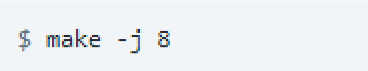
Installing Python 3.7 on Ubuntu from source

- Step 4: Next, navigate to the Python source directory and run the configure script which will perform a number of checks to make sure all of the dependencies on your system are present



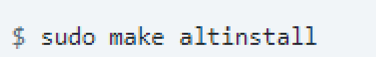
```
$ cd Python-3.7.3
$ ./configure --enable-optimizations
```

- Step 5: Start the Python build process using make



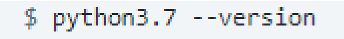
```
$ make -j 8
```

- Step 6: When the build is done install the Python binaries by typing

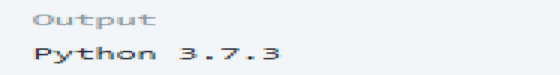


```
$ sudo make altinstall
```

Step 7: Python 3.7 is installed and ready to be used, verify it by typin



```
$ python3.7 --version
```



```
Output
Python 3.7.3
```

INSTALLATION

3 ways to Run Python :

- Using the shell by typing python on the terminal

- Runing python file
  - Creat my_file.py
  - On treminal python my_file.py

- Using the shebang way
  - Let the 1st line of your python file be # !/usr/bin/env python
  - Give your file permission execute sudo chmod +x my_file.py
  - Then on terminal ./my_file.py

BASIC SYNTAX

Interactive Mode Programming:
Invoking the interpreter without passing a script file as a parameter

```
$ python
```

Write print("Hello, Python!")

Script Mode Programming:
Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Write into script file (.py)----- print("Hello, Python!")
$ python test.py (test .py is name of script)

BASIC SYNTAX

Python Identifiers:

- A Python identifier is a name used to identify a variable, function, class, module or other object.

- An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

- Python does not allow punctuation characters such as @, $, and % within identifiers.

- Python is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in Python.

BASIC SYNTAX

Python Identifiers:

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

BASIC SYNTAX

Python Identifiers:

- Where in other programming languages the indentation in code is for readability only, in Python the indentation is very important.

- Python uses indentation to indicate a block of code.

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

BASIC SYNTAX

Python Identifiers:

Multi-Line Statements:

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

```
total = item_one + \
          item_two + \
          item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

```
days = ['Monday', 'Tuesday', 'Wednesday',
            'Thursday', 'Friday']
```

BASIC SYNTAX

Python Identifiers:

Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

BASIC SYNTAX

Comments

- Python has commenting capability for the purpose of in-code documentation.

- Comments start with a #, and Python will render the rest of the line as a comment:
  #This is a comment.
  print("Hello, World!")

- Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:
  '''

- This is a multiline comment.

  '''

BASIC SYNTAX

Waiting for the User:

```
#!/usr/bin/python
input("\n\nPress the enter key to exit.")
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line:

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon −

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

BASIC SYNTAX

Python Identifiers:

Docstrings

Python also has extended documentation capability, called docstrings.

Docstrings can be one line, or multiline.

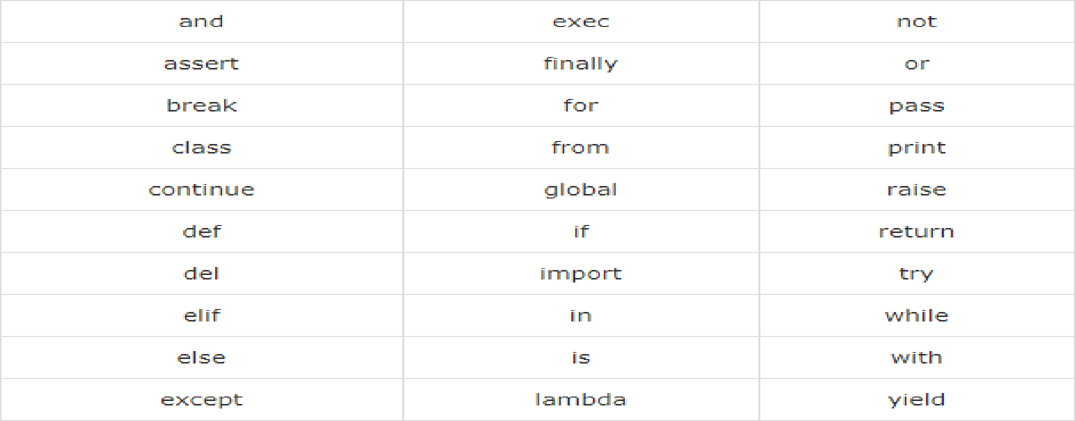
Python uses triple quotes at the beginning and end of the docstring:

```
"""This is a
multiline docstring."""
print("Hello, World!")
```

BASIC SYNTAX

Python Identifiers:

Reserved words



and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

PEP 8 -- Style Guide for Python Code:

https://www.python.org/dev/peps/pep-0008/?fbclid=IwAR1xO0Guh3l-LWpxNZSoo1fK0LBzE46K6Jm5QuLrKPion4a-pMwS1mwp-80

VARIABLES

Variables are nothing but reserved memory locations to store values.This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

VARIABLES

Assigning Values to Variables:

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
#!/usr/bin/env python
counter = 100 # An integer assignment
miles = 1000.0 # A floating point
name = “SyS" # A string
print(counter)
print(miles)
print(name)
```

VARIABLES

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example −

a = b = c = 1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

a,b,c = 1,2,"john“

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

VARIABLES

Variable Names:
A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

VARIABLES

Standard Data Types:

- Python has five standard data types −
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary

VARIABLES

Output Variables:

```
x = "awesome"
print("Python is " + x)
---------------------------------
x = "Python is "
y = "awesome"
z =  x + y
print(z)
---------------------------------
x = 5
y = 10
print(x + y)
--------------------------------
ERROR
x = 5
y = «sys"
print(x + y)
```

NUMBERS

Python supports four different numerical types –

- int (signed integers)
- long (long integers)
- float (floating point real values)
- complex (complex numbers) - (self study)

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
-----------------------------
print(type(x))
print(type(y))
print(type(z))
```

NUMBERS

Casting

Integers:

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

Floats:

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
```

Strings:

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

NUMBERS

Mathematical Functions

Examples :

pow()

abs()

max()

min()

round()

For more visit :

https://docs.python.org/3/library/math.html?fbclid=IwAR2mlfGCRe9epB7bkguchiQKMyn94VaXfY1PQnd3dNcEpPWvO9WCOhQrbT4

STRINGS

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example −

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

STRINGS

Accessing Values in Strings:

strings in Python are arrays of bytes representing unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

```
#!/usr/bin/python

var1 = 'Hello World!'

var2 = "Python Programming"

print( "var1[0]: ", var1[0])

print("var2[1:5]: ", var2[1:5])
```

STRINGS

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

```
#!/usr/bin/python

var1 = 'Hello World!'

print ("Updated String :- ", var1[:6] + 'Python‘)
```

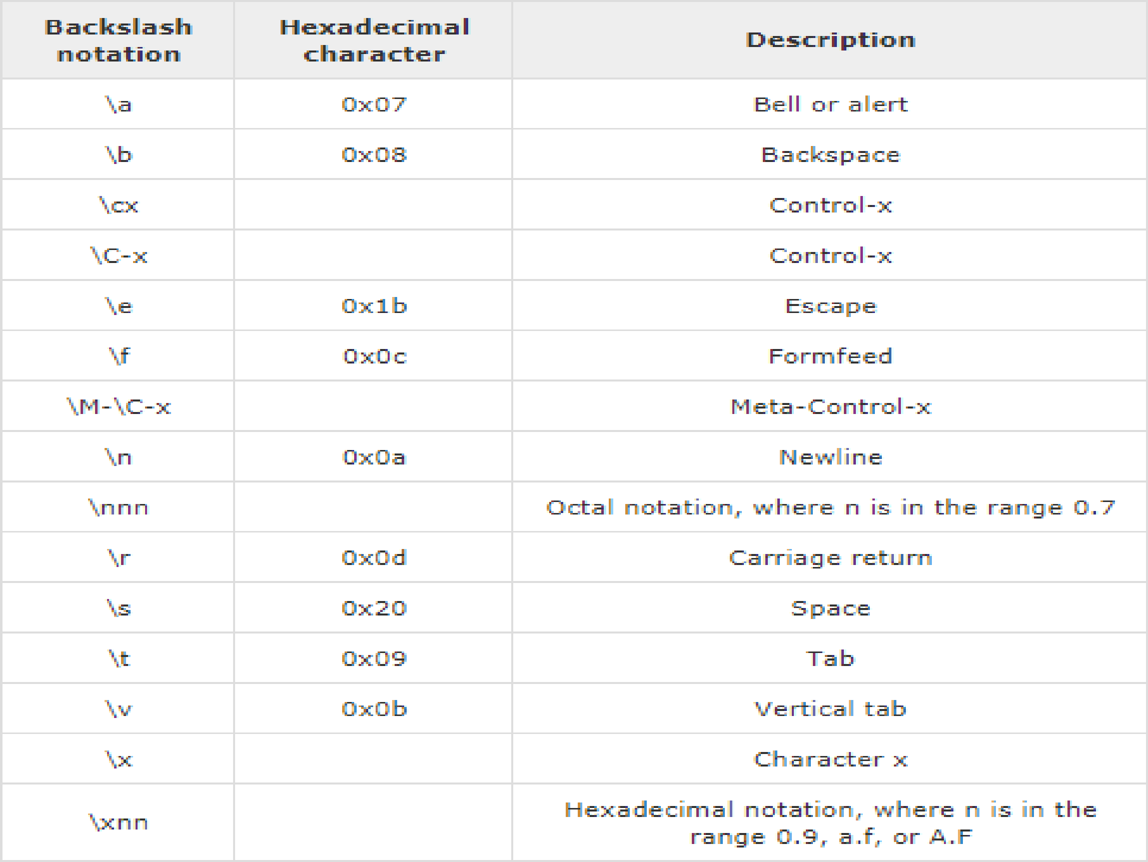
STRINGS
INPUT/OUTPUT: `print`

- used to **output** stuff to console
- keyword is `print`

```
x = 1
print(x)
x_str = str(x)
print("my fav num is", x, ".", "x =", x)
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

STRINGS

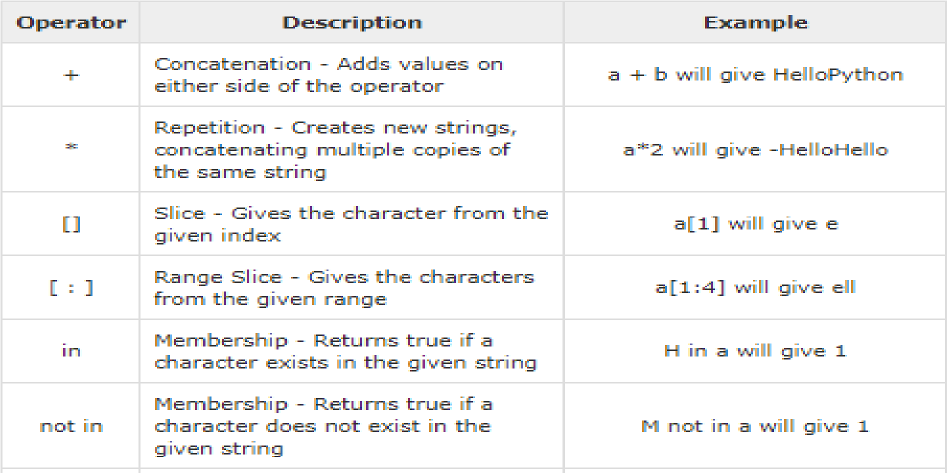
Escape Characters



Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

STRINGS

Some String Special Operators



Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1

- My_str * 3 #will print its value 3 times
- String coparison using == , is , in
- My_str = str (5) #to convert from int to string we use str () method

STRINGS

String Formating Operator

```
#!/usr/bin/python
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

set of symbols which can be used along with % − (self study)

More format (self study)

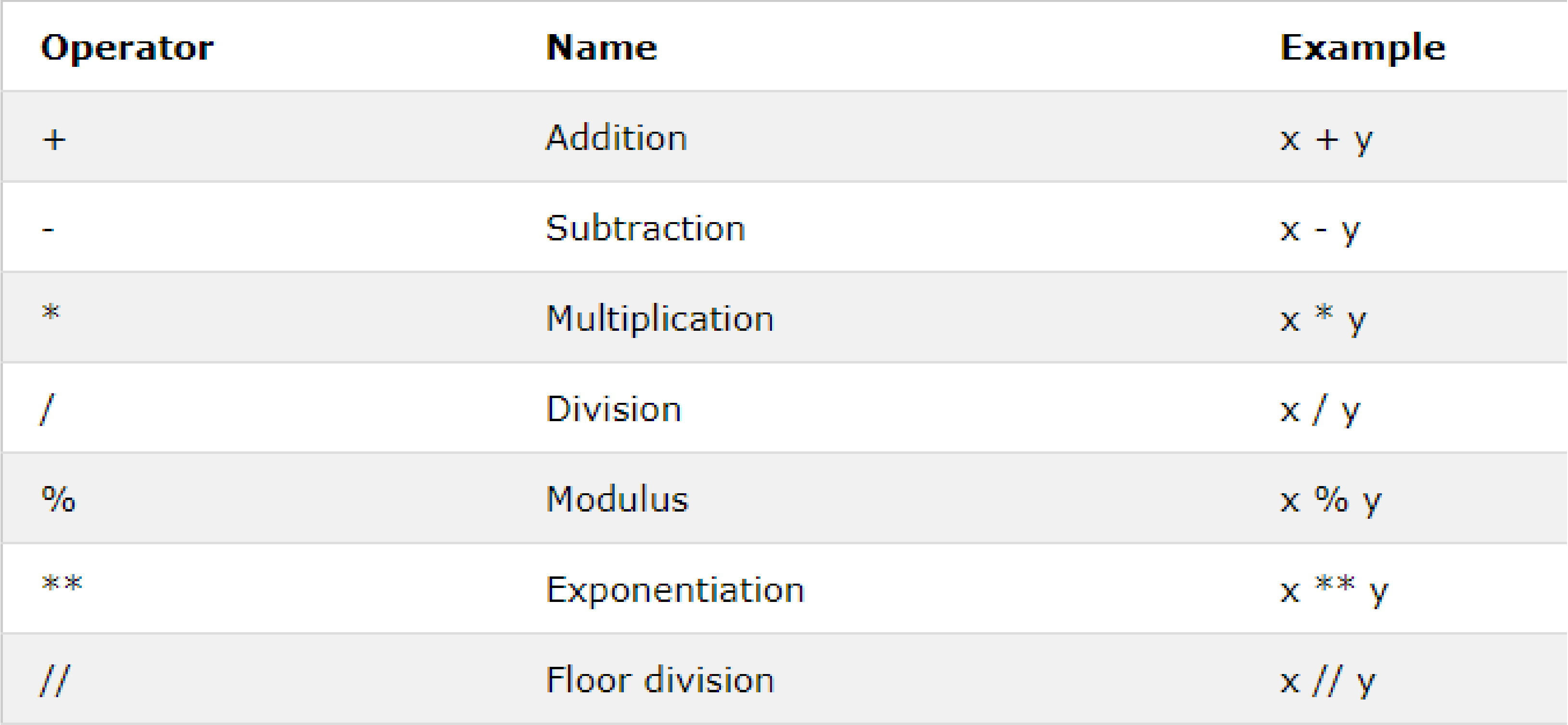
STRINGS

Triple Quotes:

```
#!/usr/bin/python

para_str = """this is a long string that is made up of several lines and
non-printable characters such as TAB ( \t ) and they will show up that
way when displayed. NEWLINEs within the string, whether explicitly
given like this within the brackets [ \n ], or just a NEWLINE within the
variable assignment will also show up.
"""
print (para_str)
```

OPERATORS

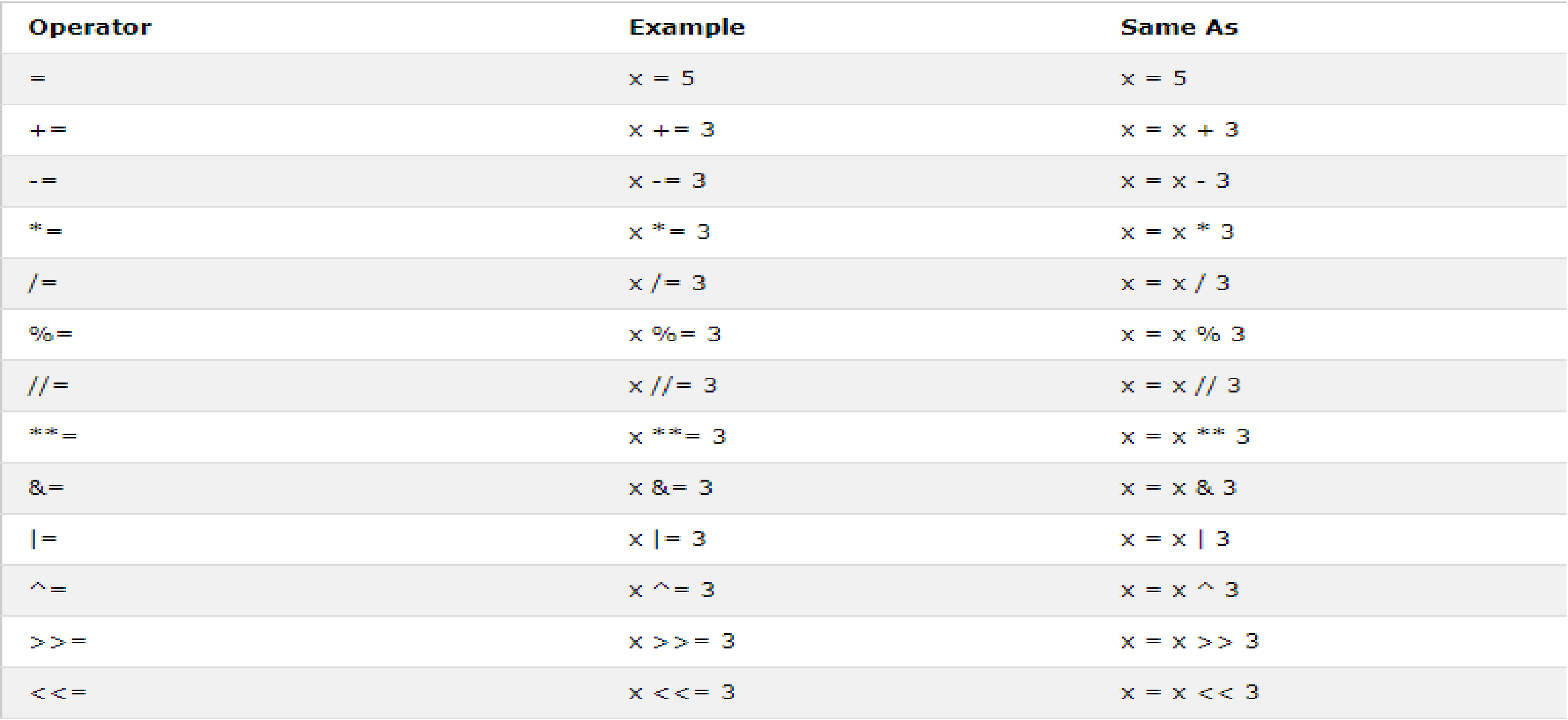
Python Arithmetic Operators



Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

OPERATORS

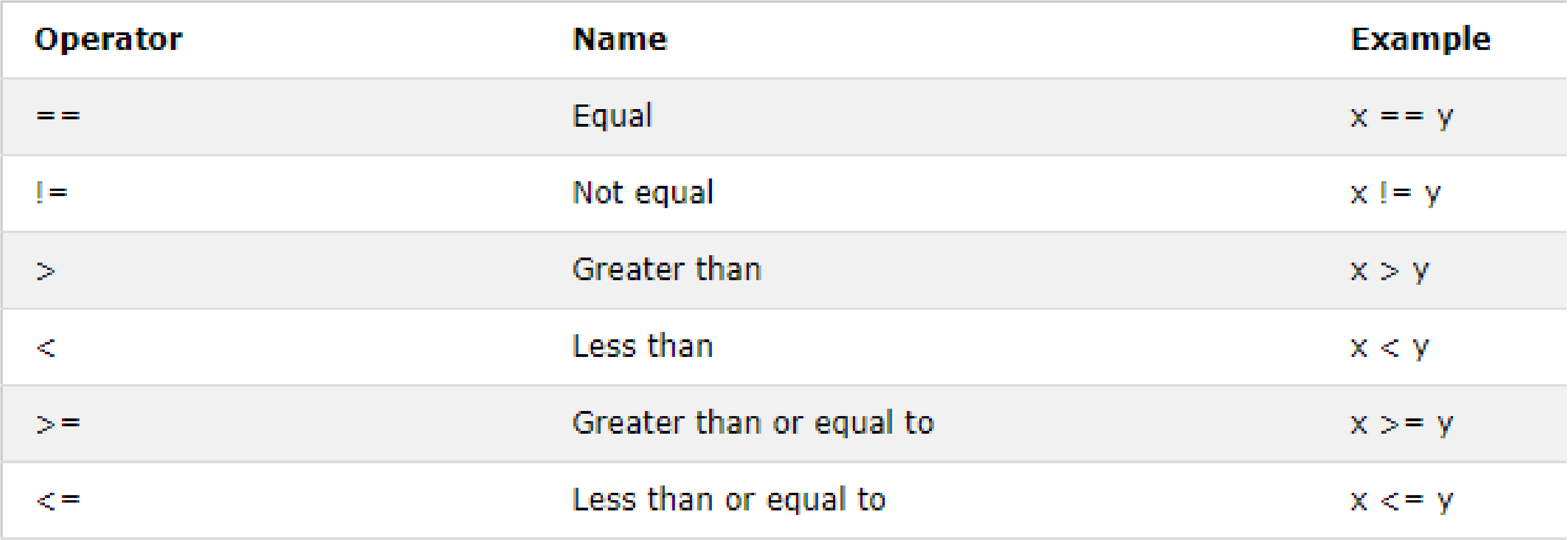
Python Assignment Operators



Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

OPERATORS

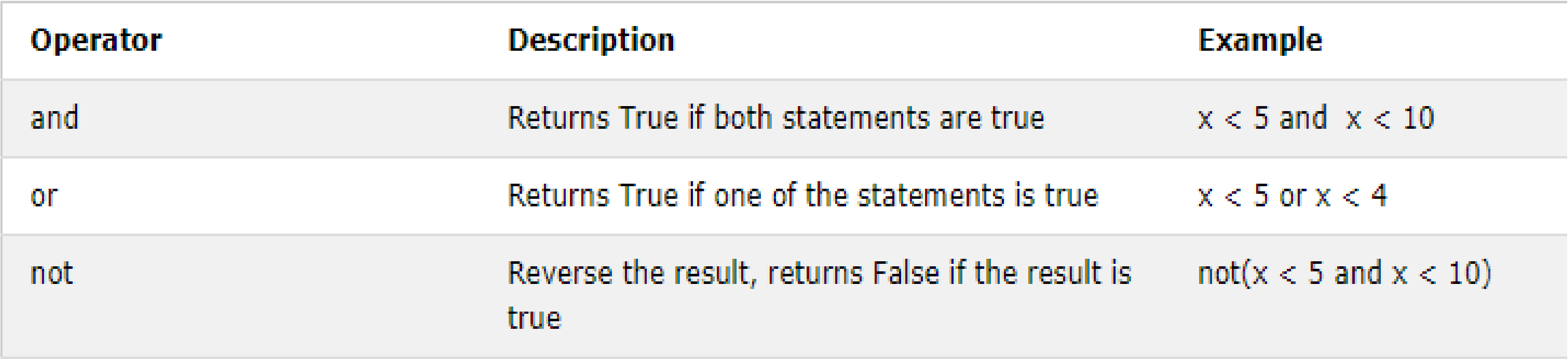
Python Comparison Operators



Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

OPERATORS

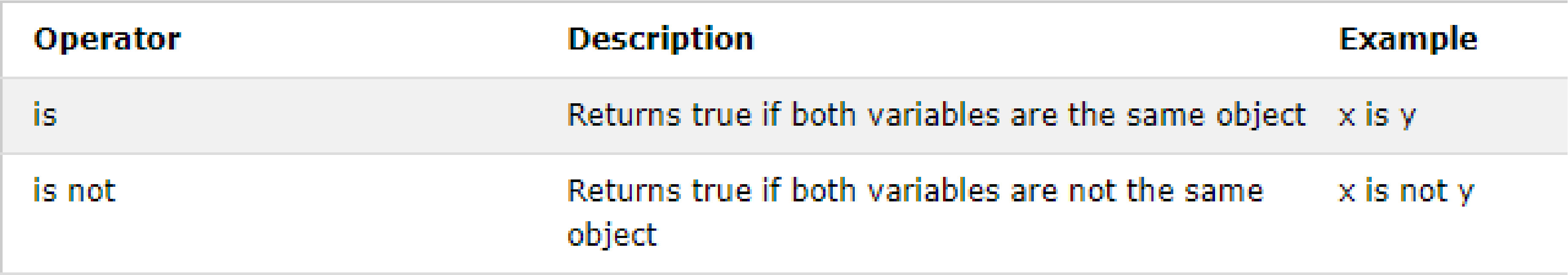
Python Logical Operators



Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

OPERATORS

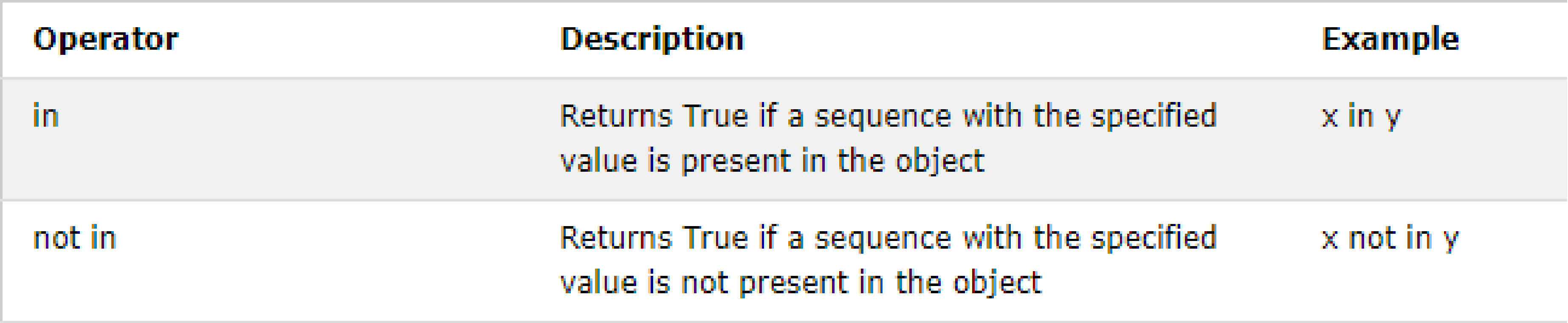
Python Identity Operators



Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

OPERATORS

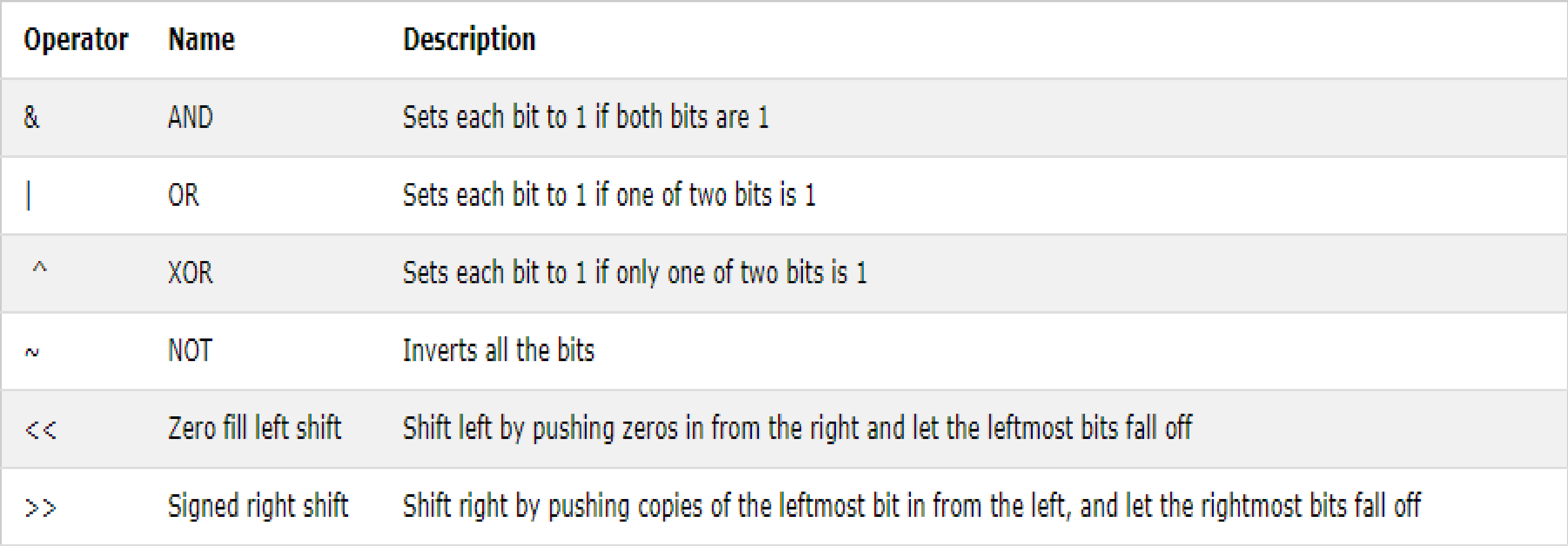
Python Membership Operators



Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

OPERATORS

Python Bitwise Operators



Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

COMPARISON OPERATORS ON `int, float, string`

- `i` and `j` are variable names
- comparisons below evaluate to a Boolean

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` • **equality** test, `True` if `i` is the same as `j`

`i != j` • **inequality** test, `True` if `i` not the same as `j`

INPUT/OUTPUT: `input("")`

- prints whatever is in the quotes
- user types in something and hits enter
- binds that value to a variable

```
text = input("Type anything... ")
print(5*text)
```

- `input` **gives you a string** so must cast if working with numbers

```
num = int(input("Type a number... "))
print(5*num)
```

LOGIC OPERATORS ON bools

- a and b are variable names (with Boolean values)

not a
- True if a is False
False if a is True

a and b
- True if both are True

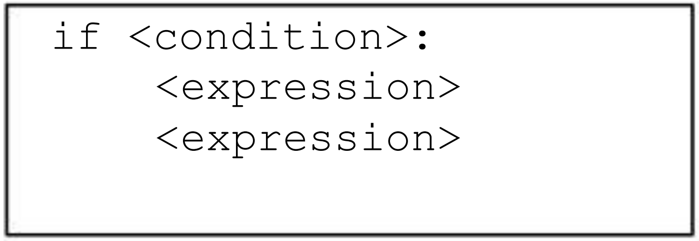
a or b
- True if either or both are True

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

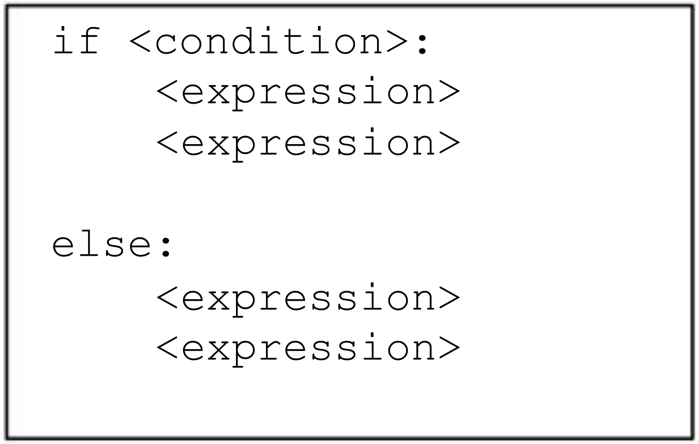
COMPARISON EXAMPLE

```
pset_time = 15
sleep_time = 8
print(sleep_time > pset_time)
derive = True
drink = False
both = drink and derive
print(both)
```

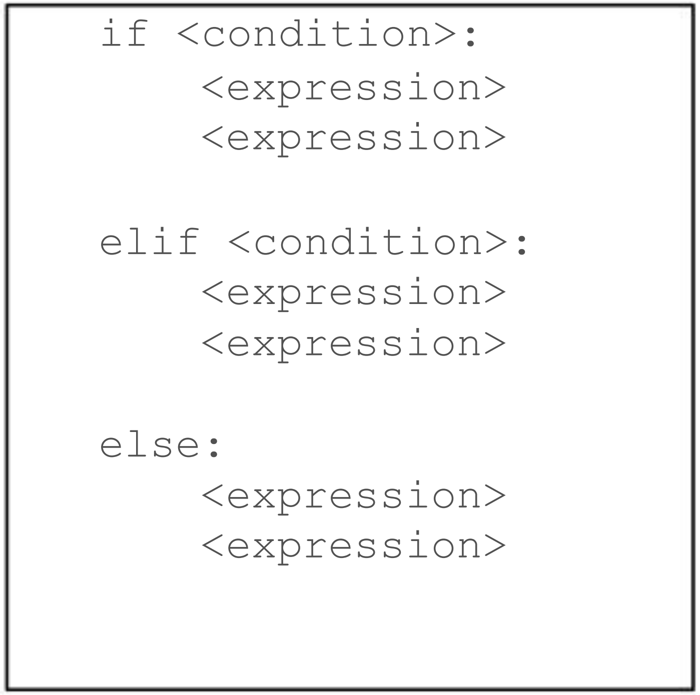
CONTROL FLOW - BRANCHING



```
if <condition>:
    <expression>
    <expression>
```



```
if <condition>:
    <expression>
    <expression>

else:
    <expression>
    <expression>
```



```
if <condition>:
    <expression>
    <expression>

elif <condition>:
    <expression>
    <expression>

else:
    <expression>
    <expression>
```

- `<condition>` has a value `True` or `False`
- evaluate expressions in that block if `<condition>` is `True`

CONTROL FLOW

Normal IF ……. Elif

```
if (condition): #condition can be between ()

          Statement

elif condition: #condition can be without ()

          Statement

else:
            Statement
```

CONTROL FLOW

Short Hand IF

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

Short Hand IF ….Else

```
print("A") if a > b else print("B")
```

One line if else statement, with 3 conditions:

```
print("A") if a > b else print("=") if a == b else print("B")
```

CONTROL FLOW

Using (And) , (OR)

```
if a > b and c > a:
        print("Both conditions are True")



if a > b or a > c:
        print("At least one of the conditions is
True")
```

LOOP

For Loop

```
for value in list:
            print(value) #for on list

for key, value in dict:
            print(key, value) #for on a dictionary

for value in range(2, 11, 2):
            print(value) #for on a range method result
```

Note: range(start, end, step)

LOOP range(start,stop,step)

- default values are `start= 0` and `step = 1` and optional
- loop until value is `stop- 1`

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```

LOOP

For Loop

Else in For Loop

```
for x in range(6):
        print(x)
else:
        print("Finally finished!")
```

Nested Loops

```
for x in adj:
    for y in fruits:
        print(x, y)
```

CONTROL FLOW: `while` LOOPS

```
while <condition>:
    <expression>
    <expression>
```

- `<condition>` evaluates to a Boolean
- if `<condition>` is `True`, do all the steps inside the while code block
- check `<condition>` again
- repeat until `<condition>` is `False`

LOOP

While Loop

```
while condition:
       Statement
        Condition change
```

Note: Python deosn't have ++ operator but += works

Loop interruption operators:

Break: get out the entire loop.

Continue: skip this loop and go for the next one.

Pass: null operation nothing happens on execution.

Python Iterators (Self Study)

`while` LOOP EXAMPLE

```
You are in the Lost Forest.
************
************
   .
************
************
Go left or right?
```

PROGRAM:

```
n = input("You're in the Lost Forest. Go left or right? ")
while n == "right":
    n = input("You're in the Lost Forest. Go left or right? ")
print("You got out of the Lost Forest!")
```

`for` VS `while` LOOPS

`for` loops

- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

`while` loops

- **unbounded** number of iterations
- can **end early** via `break`
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a `while` loop using a `for` loop