

MCA I

Python Overview

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

Python Overview

- **Python is Interpreted:** This means that it is processed at runtime by the interpreter and you do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** This means that you can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** This means that Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is Beginner's Language:** Python is a great language for the beginner 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 it's progress.

Python Features

- **Easy-to-learn:** Python has relatively few keywords, simple structure, and a clearly defined syntax.
- **Easy-to-read:** Python code is much more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's success is that its source code is fairly easy-to-maintain.
- **A broad standard library:** One of Python's greatest strengths is the bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode:** Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.

Python Features (cont'd)

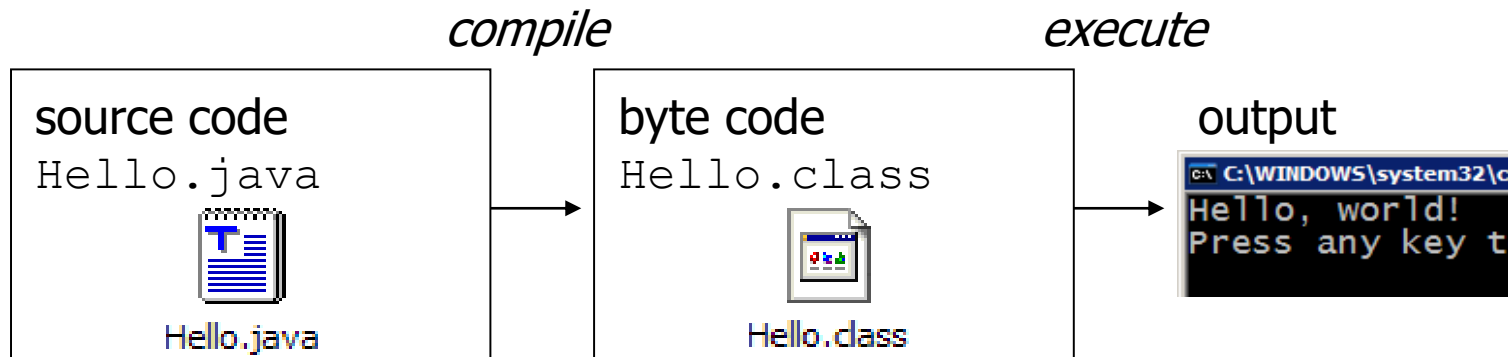
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python 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.

Python Environment

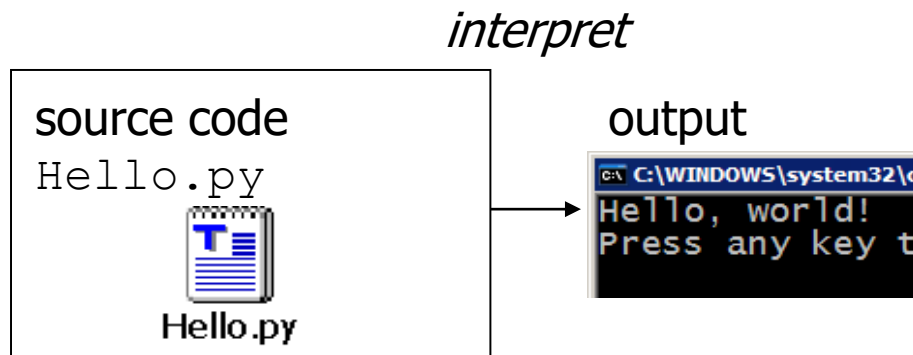
- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX etc.)
- Win 9x/NT/2000
- Macintosh (PPC, 68K)
- OS/2
- DOS (multiple versions)
- PalmOS
- Nokia mobile phones
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion
- Python has also been ported to the Java and .NET virtual machines.

Compiling and interpreting

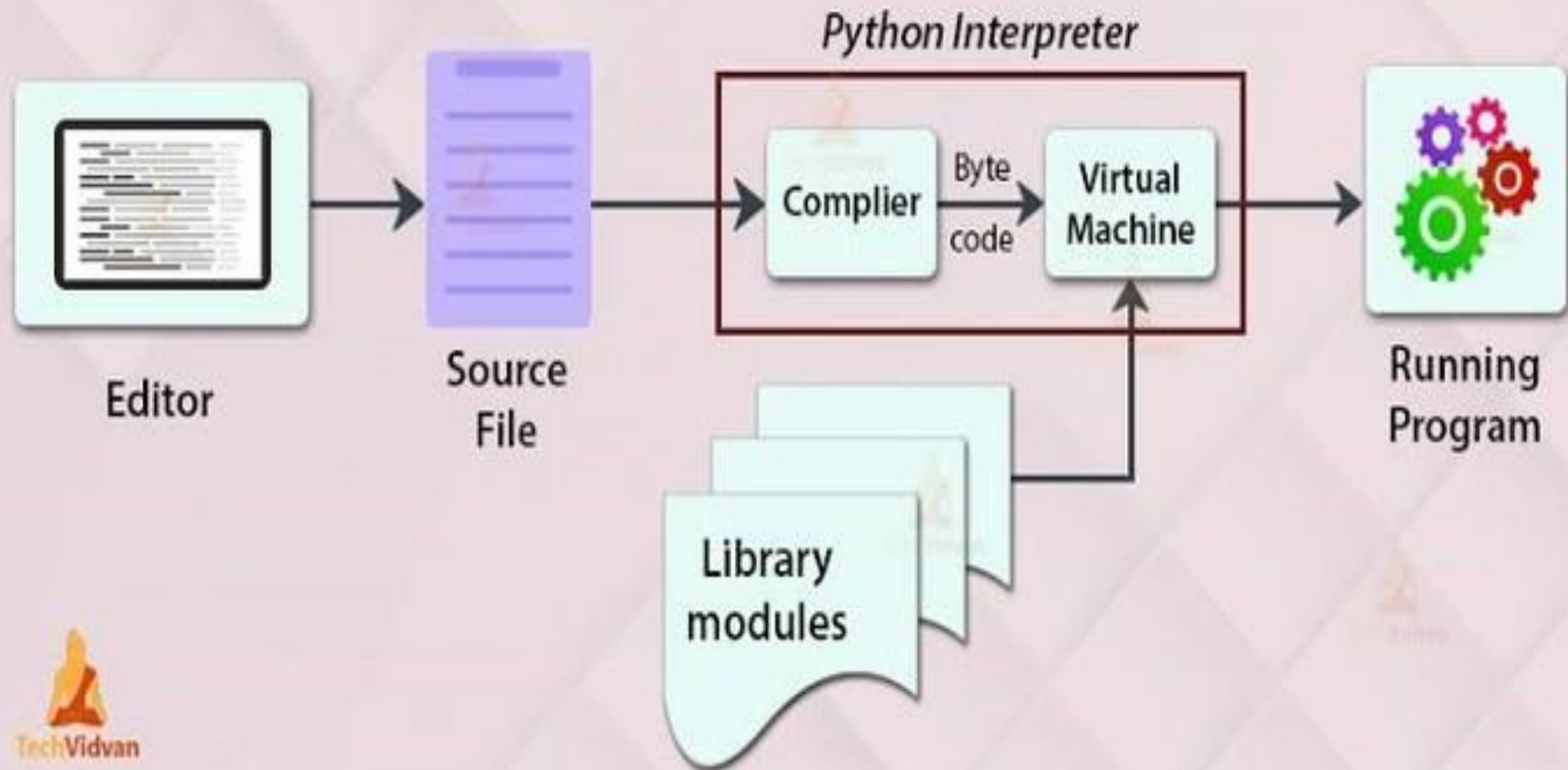
- Many languages require you to *compile* (translate) your program into a form that the machine understands.



- Python is instead directly *interpreted* into machine instructions.



How Python Interpreter Works?



Frozen Binaries

- .pyc files, related Python Libraries and PVM in single executable file .exe (True executable file)
- To create frozen binaries, third party tool is needed.
- Py2exe for Windows exe
- Pyinstaller and Freeze for Unix or Linux

2. Python - Basic Syntax

- **Interactive Mode Programming:**

```
>>> print "Hello, Python!";
```

```
Hello, Python!
```

```
>>> 3+4*5;
```

```
23
```

- **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.

For example, put the following in one test.py, and run,

```
print "Hello, Python!";  
print "I love COMP3050!";
```

The output will be:

```
Hello, Python!  
I love COMP3050!
```

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.

Python Identifiers (cont'd)

- Here are following identifier naming convention for Python:
 - Class names start with an uppercase letter and all other identifiers with a lowercase letter.
 - Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be 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.

Reserved Words:

Keywords contain lowercase letters only.

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

Lines and Indentation:

- One of the first caveats programmers encounter when learning Python is the fact that there are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Both blocks in this example are fine:

```
if True:
    print "Answer";
    print "True" ;
else:
    print "Answer";
    print "False"
```


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']
```

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 can be 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."""
```

Comments in Python:

- A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment, and the Python interpreter ignores them.

Using Blank Lines:

- A line containing only whitespace, possibly with a comment, is known as a blank line, and Python totally ignores it.
- In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

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')
```

Multiple Statement Groups as Suites:

- Groups of individual statements making up a single code block are called **suites** in Python.

Compound or complex statements, such as `if`, `while`, `def`, and `class`, are those which require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (`:`) and are followed by one or more lines which make up the suite.

```
if expression :
```

```
    suite
```

```
elif expression :
```

```
    suite
```

```
else :
```

```
    suite
```

Python - Variable Types

- 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.

Assigning Values to Variables:

- Python variables do not have to be explicitly declared 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.

```
counter = 100 # An integer assignment
```

```
miles = 1000.0 # A floating point
```

```
name = "John" # A string
```

```
print counter
```

```
print miles
```

```
print name
```


Multiple Assignment:

- You can also assign a single value to several variables simultaneously. For example:

```
a = b = c = 1
```

```
a, b, c = 1, 2, "john"
```

Standard Data Types:

Python has five standard data types:

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers:

- Number data types store numeric values. They are immutable data types, which means that changing the value of a number data type results in a newly allocated object.
- Number objects are created when you assign a value to them. For example:

```
var1 = 1  
var2 = 10
```

Python supports four different numerical types:

- int (signed integers)
- long (long integers [can also be represented in octal and hexadecimal])
- float (floating point real values)
- complex (complex numbers)

Number Examples:

int	long	float	complex
10	51924361L	0	3.14j
100	-0x19323L	15.2	45.j
-786	0122L	-21.9	9.322e-36j
80	0xDEFA BCECBDAECBFBAEI	32.3+e18	.876j
-490	535633629843L	-90	-.6545+0J
-0x260	-052318172735L	-3.25E+101	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

Python Strings:

- Strings in Python are identified as a contiguous set of characters in between quotation marks.
- Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator (`[]` and `[:]`) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus (`+`) sign is the string concatenation operator, and the asterisk (`*`) is the repetition operator.

Literals

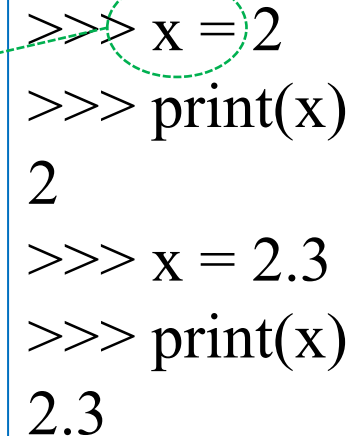
- In the following example, the parameter values passed to the print function are all technically called *literals*
 - More precisely, “Hello” and “Programming is fun!” are called *textual literals*, while 3 and 2.3 are called *numeric literals*

```
>>> print("Hello")
Hello
>>> print("Programming is fun!")
Programming is fun!
>>> print(3)
3
>>> print(2.3)
2.3
```

Simple Assignment Statements

- A literal is used to indicate a specific value, which can be *assigned* to a *variable*

- x is a variable and 2 is its value

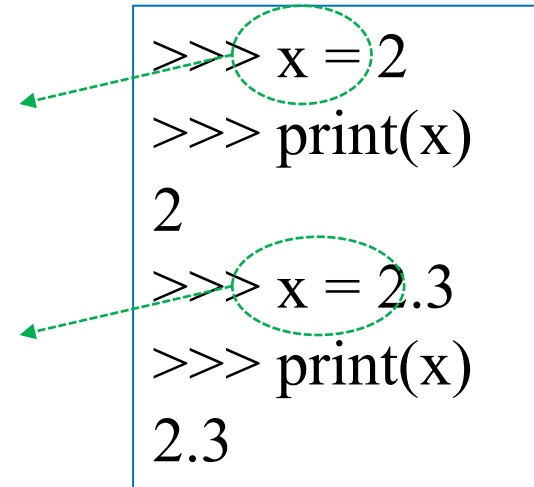


```
>>> x = 2
>>> print(x)
2
>>> x = 2.3
>>> print(x)
2.3
```

Simple Assignment Statements

- A literal is used to indicate a specific value, which can be *assigned* to a *variable*

- x is a variable and 2 is its value
- x can be assigned different values; hence, it is called a variable



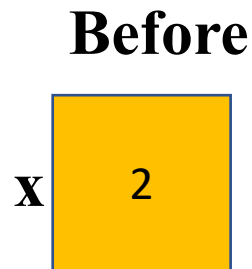
The image shows two snippets of Python code within a blue rectangular border. The first snippet consists of two lines: `>>> x = 2` and `>>> print(x)`. The second snippet also consists of two lines: `>>> x = 2.3` and `>>> print(x)`. In the first snippet, the value `2` is circled with a dashed green line, and a green arrow points from this circle to the text '2 is its value' in the list to the left. In the second snippet, the value `2.3` is circled with a dashed green line, and a green arrow points from this circle to the text 'x can be assigned different values; hence, it is called a variable' in the list to the left. The output of the first snippet is the number `2`, and the output of the second snippet is the number `2.3`.

```
>>> x = 2
>>> print(x)
2
>>> x = 2.3
>>> print(x)
2.3
```

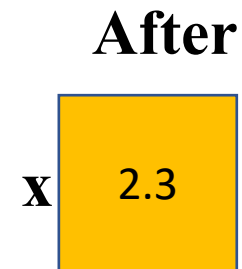

Simple Assignment Statements: Box View

- A simple way to view the effect of an assignment is to assume that when a variable changes, its old value is replaced

```
>>> x = 2
>>> print(x)
2
>>> x = 2.3
>>> print(x)
2.3
```

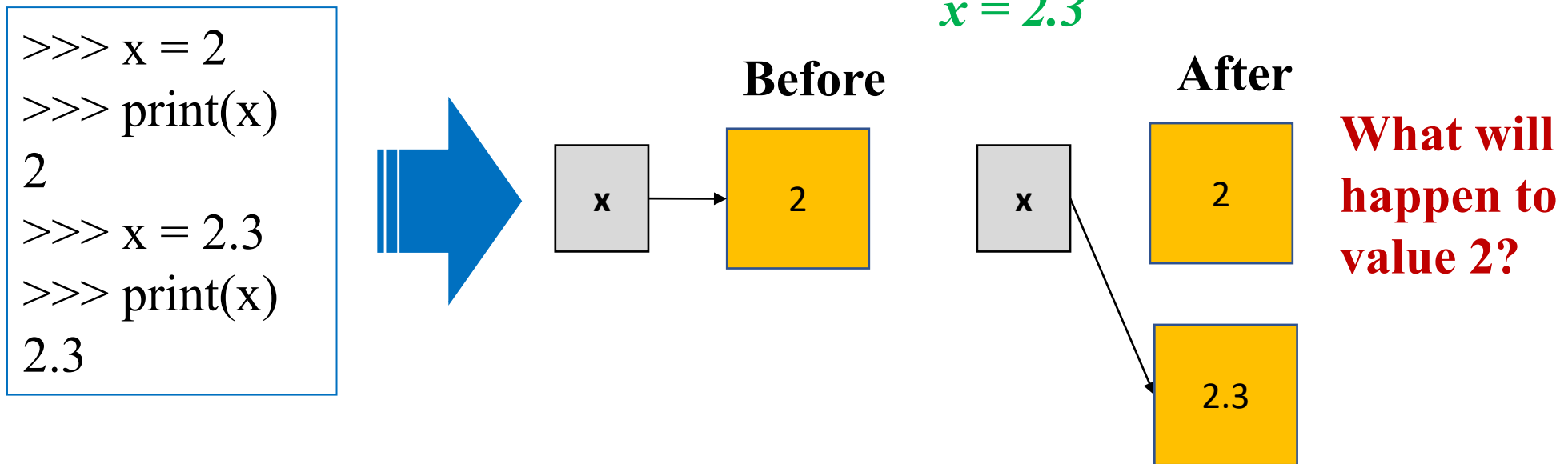


x = 2.3



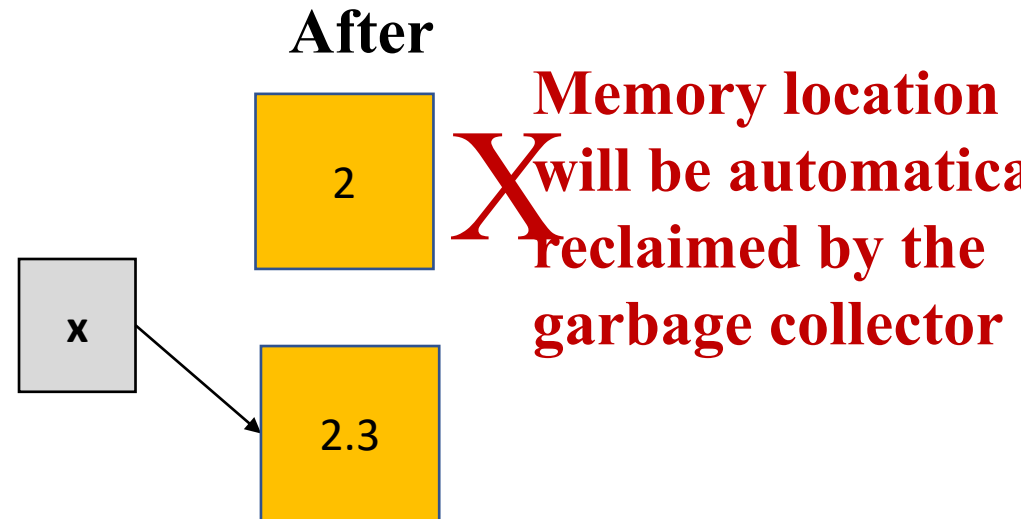
Simple Assignment Statements: Actual View

- Python assignment statements are actually slightly different from the “variable as a box” model
 - In Python, values may end up anywhere in memory, and variables are used to refer to them



Garbage Collection

- Interestingly, as a Python programmer you do not have to worry about computer memory getting filled up with old values when new values are assigned to variables
- Python will automatically clear old values out of memory in a process known as *garbage collection*



Assigning Input

- So far, we have been using values specified by programmers and printed or assigned to variables
 - How can we let users (not programmers) input values?
- In Python, input is accomplished via an assignment statement combined with a built-in function called *input*

<variable> = input(<prompt>)

- When Python encounters a call to *input*, it prints <prompt> (which is a string literal) then pauses and waits for the user to type some text and press the <Enter> key

Assigning Input

- Here is a sample interaction with the Python interpreter:

```
>>> name = input("Enter your  
name: ")  
Enter your name: Akshay Kumar  
>>> name  
'Akshay Kumar'  
>>>
```

- Notice that whatever the user types is then stored as a string
 - What happens if the user inputs a number?

Assigning Input

- Here is a sample interaction with the Python interpreter:

Still a string!

```
>>> number = input("Enter a  
number: ")  
Enter a number: 3  
>>> number  
'3'  
>>>
```

- How can we force an input number to be stored as a number and not as a string?
 - We can use the built-in *eval* function, which can be “wrapped around” the input function

Assigning Input

- Here is a sample interaction with the Python interpreter:

```
>>> number = eval(input("Enter a  
number: "))  
Enter a number: 3  
>>> number  
3  
>>>
```

**Now an int
(no single quotes)!**

Assigning Input

- Here is a sample interaction with the Python interpreter:

```
>>> number = eval(input("Enter a  
number: "))  
Enter a number: 3.7  
>>> number  
3.7  
>>>
```

**And now a float
(no single quotes)!**




3.7

Assigning Input

- Here is another sample interaction with the Python interpreter:

```
>>> number = eval(input("Enter an  
equation: "))  
Enter an equation: 3 + 2  
>>> number  
5  
>>>
```



The *eval* function will evaluate this formula and return a value, which is then assigned to the variable “number”

Datatype Conversion

- Besides, we can convert the string output of the *input* function into an integer or a float using the built-in *int* and *float* functions

```
>>> number = int(input("Enter a  
number: "))  
Enter a number: 3  
>>> number  
3  
>>>
```

**An integer
(no single quotes)!**



Datatype Conversion

- Besides, we can convert the string output of the *input* function into an integer or a float using the built-in *int* and *float* functions

```
>>> number = float(input("Enter a  
number: "))  
Enter a number: 3.7  
>>> number  
3.7  
>>>
```

**A float
(no single quotes)!**



Datatype Conversion

- As a matter of fact, we can do various kinds of conversions between strings, integers and floats using the built-in *int*, *float*, and *str* functions

```
>>> x =  
10  
>>>  
float(x)  
10.0  
>>> str(x)  
'10'  
>>>
```

integer → float
integer → string

```
>>> y =  
"20"  
>>>  
float(y)  
20.0  
>>> int(y)  
20  
>>>
```

string → float
string → integer

```
>>> z =  
30.0  
>>> int(z)  
30  
>>> str(z)  
'30.0'  
>>>
```

float → integer
float → string

Simultaneous Assignment

- Python allows us also to assign multiple values to multiple variables all at the same time

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
>>>
```

- This form of assignment might seem strange at first, but it can prove remarkably useful (e.g., for swapping values)

Simultaneous Assignment

- Suppose you have two variables *x* and *y*, and you want to swap their values (*i.e.*, you want the value stored in *x* to be in *y* and vice versa)

```
>>> x = 2
>>> y = 3
>>> x = y
>>> y = x
>>> x
3
>>> y
3
```

**X CANNOT be done with
two simple assignments**

Simultaneous Assignment

- Suppose you have two variables `x` and `y`, and you want to swap their values (*i.e.*, you want the value stored in `x` to be in `y` and vice versa)

Thus far, we have been using different *names* for variables. These names are technically called *identifiers*

```
>>> x = 2
>>> y = 3
>>> temp = x
>>> x = y
>>> y = temp
>>> x
3
>>> y
2
>>>
```

✓ **CAN be done with *three* simple assignments, but more efficiently with simultaneous assignment**

Identifiers

- Python has some rules about how identifiers can be formed
 - Every identifier must begin with a letter or underscore, which may be followed by any sequence of letters, digits, or underscores

```
>>> x1 = 10
>>> x2 = 20
>>> y_effect = 1.5
>>> celsius = 32
>>> 2celsius
File "<stdin>", line 1
    2celsius
      ^
SyntaxError: invalid syntax
```


Identifiers

- Python has some rules about how identifiers can be formed
 - Identifiers are *case-sensitive*

```
>>> x = 10
>>> X = 5.7
>>> print(x)
10
>>> print(X)
5.7
```

Identifiers

- Python has some rules about how identifiers can be formed
 - Some identifiers are part of Python itself (they are called *reserved words* or *keywords*) and cannot be used by programmers as ordinary identifiers

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Python Keywords

Identifiers

- Python has some rules about how identifiers can be formed
 - Some identifiers are part of Python itself (they are called *reserved words* or *keywords*) and cannot be used by programmers as ordinary identifiers

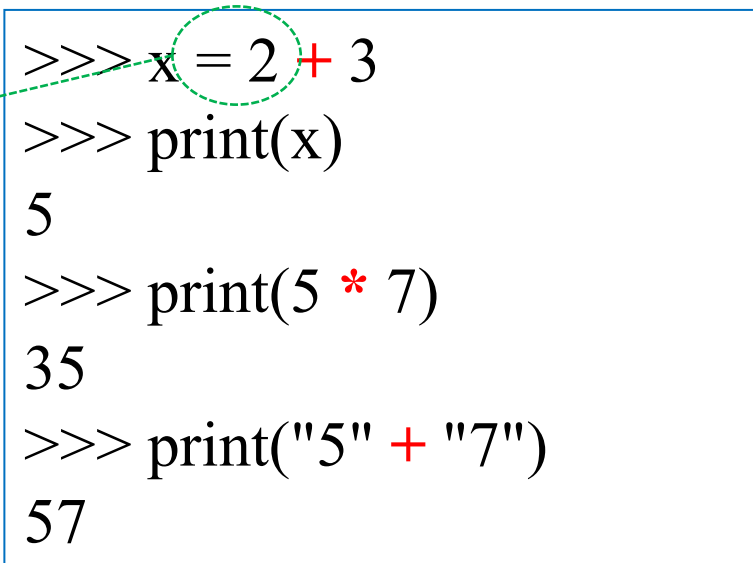
An example...

```
>>> for = 4
      File "<stdin>", line 1
        for = 4
          ^
      SyntaxError: invalid syntax
```

Expressions

- You can produce new data (numeric or text) values in your program using *expressions*

- This is an expression that uses the *addition operator*

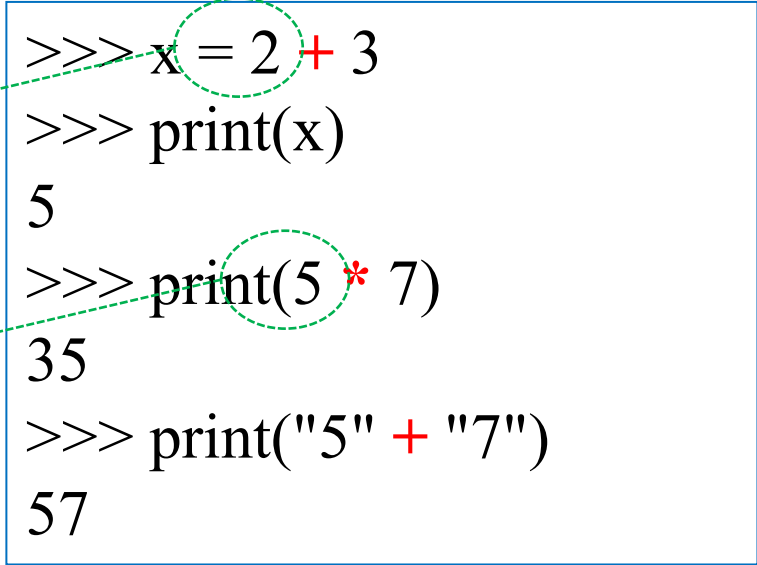


```
>>> x = 2 + 3
>>> print(x)
5
>>> print(5 * 7)
35
>>> print("5" + "7")
57
```

Expressions

- You can produce new data (numeric or text) values in your program using *expressions*

- This is an expression that uses the *addition operator*
- This is another expression that uses the *multiplication operator*



```
>>> x = 2 + 3
>>> print(x)
5
>>> print(5 * 7)
35
>>> print("5" + "7")
57
```

Expressions

- You can produce new data (numeric or text) values in your program using *expressions*

- This is an expression that uses the *addition operator*
- This is another expression that uses the *multiplication operator*
- This is yet another expression that uses the *addition operator* but to *concatenate* (or glue) strings together

```
>>> x = 2 + 3
>>> print(x)
5
>>> print(5 * 7)
35
>>> print("5" + "7")
57
```

Expressions

- You can produce new data (numeric or text) values in your program using *expressions*

*Another
example...*

```
>>> x = 6
>>> y = 2
>>> print(x - y)
4
>>> print(x/y)
3.0
>>> print(x//y)
3
```

*Yet another
example...*

```
>>> print(x*y)
12
>>> print(x**y)
36
>>> print(x%y)
0
>>> print(abs(-
x))
6
```

Expressions: Summary of Operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Float Division
**	Exponentiation
abs()	Absolute Value
//	Integer Division
%	Remainder

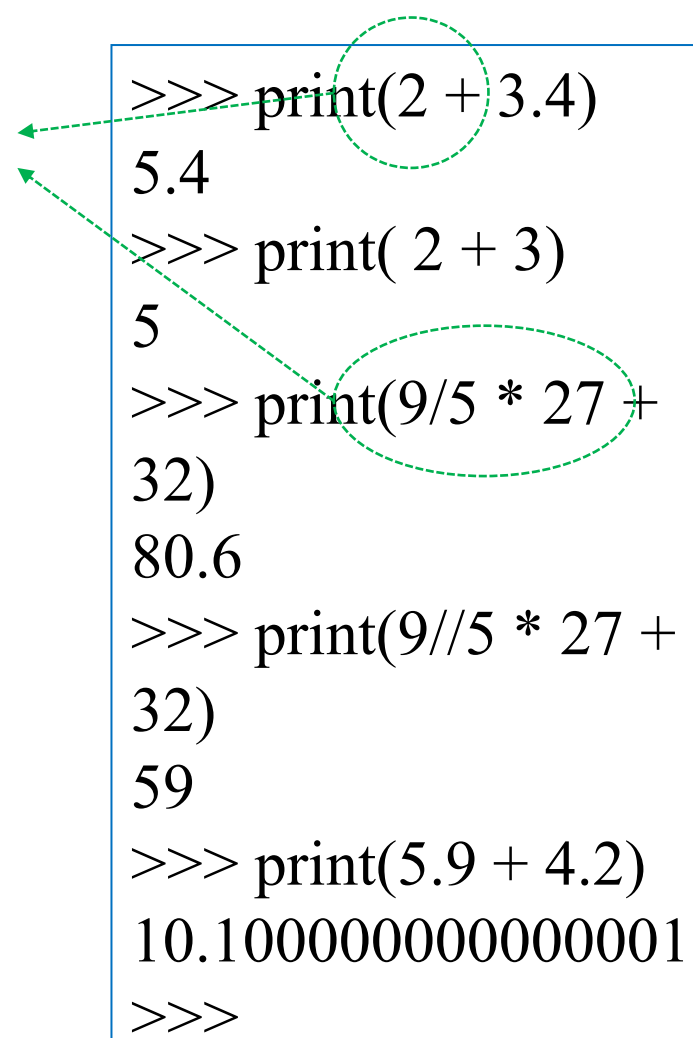
Python Built-In Numeric Operations

Explicit and Implicit Data Type Conversion

- Data conversion can happen in two ways in Python
 1. **Explicit Data Conversion** (we saw this earlier with the *int*, *float*, and *str* built-in functions)
 2. **Implicit Data Conversion**
 - Takes place *automatically* during run time between *ONLY* numeric values
 - E.g., Adding a float and an integer will automatically result in a float value
 - E.g., Adding a string and an integer (or a float) will result in an *error* since string is not numeric
 - Applies *type promotion* to avoid loss of information
 - Conversion goes from integer to float (e.g., upon adding a float and an integer) and not vice versa so as the fractional part of the float is not lost

Implicit Data Type Conversion: Examples

- The result of an expression that involves a float number alongside (an) integer number(s) is a float number



```
>>> print(2 + 3.4)
5.4
>>> print(2 + 3)
5
>>> print(9/5 * 27 + 32)
80.6
>>> print(9//5 * 27 + 32)
59
>>> print(5.9 + 4.2)
10.100000000000001
>>>
```

Implicit Data Type Conversion: Examples

- The result of an expression that involves a float number alongside (an) integer number(s) is a float number
- The result of an expression that involves values of the same data type will not result in any conversion

```
>>> print(2 + 3.4)
5.4
>>> print( 2 + 3)
5
>>> print(9/5 * 27 + 32)
80.6
>>> print(9//5 * 27 + 32)
59
>>> print(5.9 + 4.2)
10.100000000000000001
>>>
```

Data Type Summary

- **Integers:** 2323, 3234L
- **Floating Point:** 32.3, 3.1E2
- **Complex:** 3 + 2j, 1j
- **Lists:** l = [1,2,3]
- **Set:** thisset = {"apple", "banana", "cherry"}
- **Tuples:** t = (1,2,3)
- **Dictionaries:** d = {'hello' : 'there', 2 : 15}

Data Type Summary

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are **mutable**
- All variables are **references**

Modules

- One problem with entering code interactively into a Python shell is that the definitions are lost when we quit the shell
 - If we want to use these definitions again, we have to type them all over again!
- To this end, programs are usually created by typing definitions into a separate file called a *module* or *script*
 - This file is saved on disk so that it can be used over and over again
- A Python module file is just a text file with a *.py extension*, which can be created using any program for editing text (e.g., notepad or vim)

Programming Environments and IDLE

- A special type of software known as a *programming environment* simplifies the process of creating modules/programs
- A programming environment helps programmers write programs and includes features such as automatic indenting, color highlighting, and interactive development
- The standard Python distribution includes a programming environment called **IDLE** that you can use for working on the programs of this course

Summary

- Programs are composed of statements that are built from *identifiers* and *expressions*
- Identifiers are names
 - They begin with an underscore or letter which can be followed by a combination of letter, digit, and/or underscore characters
 - They are case sensitive
- Expressions are the fragments of a program that produce data
 - They can be composed of *literals*, *variables*, and *operators*

Summary

- A literal is a representation of a specific value (e.g., 3 is a literal representing the number three)
- A variable is an identifier that stores a value, which can change (hence, the name *variable*)
- Operators are used to form and combine expressions into more complex expressions (e.g., the expression $x + 3 * y$ combines two expressions together using the + and * operators)

Summary

- In Python, *assignment* of a value to a variable is done using the equal sign (i.e., =)
- Using assignments, programs can get inputs from users and manipulate them internally
- Python allows *simultaneous assignments*, which are useful for swapping values of variables
- *Datatype conversion* involves converting *implicitly* and *explicitly* between various datatypes, including integer, float, and string