ISYS90088
Introduction to Application Development

Week 3 –

Construct arithmetic expressions, evaluation, precedence, number rep, mixed type arithmetic, fundamentals of strings, Boolean and logical operations

Semester 2 , 2017

Dr. Thomas Christy

Room Change. 4.04 Doug McDonell

Objectives

After completing this Lecture, you will be able to:

- Construct arithmetic expressions and evaluate them
- Initialize and use variables with appropriate names
- Use strings for the terminal input and output of text
- Use mixed mode data types and evaluate them
- Use a few fundamental string operations
- Use boolean types and logical operators
- Construct a simple Python program that performs inputs, calculations, and outputs
- Import functions from library modules – Math

Number representation

- How do computers store data?
  - All data that is stored in the computer is converted to a sequence of binary digits – 0's and 1's
  - A computers memory is divided into tiny storage locations known as bytes.
  - In general, each byte is divided into eight smaller storage locations known as bits (binary digits).
  - When a piece of data is stored in a byte, the computer sets the eight bits to an on/off pattern that represents the data.

Storing numbers

In the binary numbering system, a number is represented as a sequence of 0's and 1's

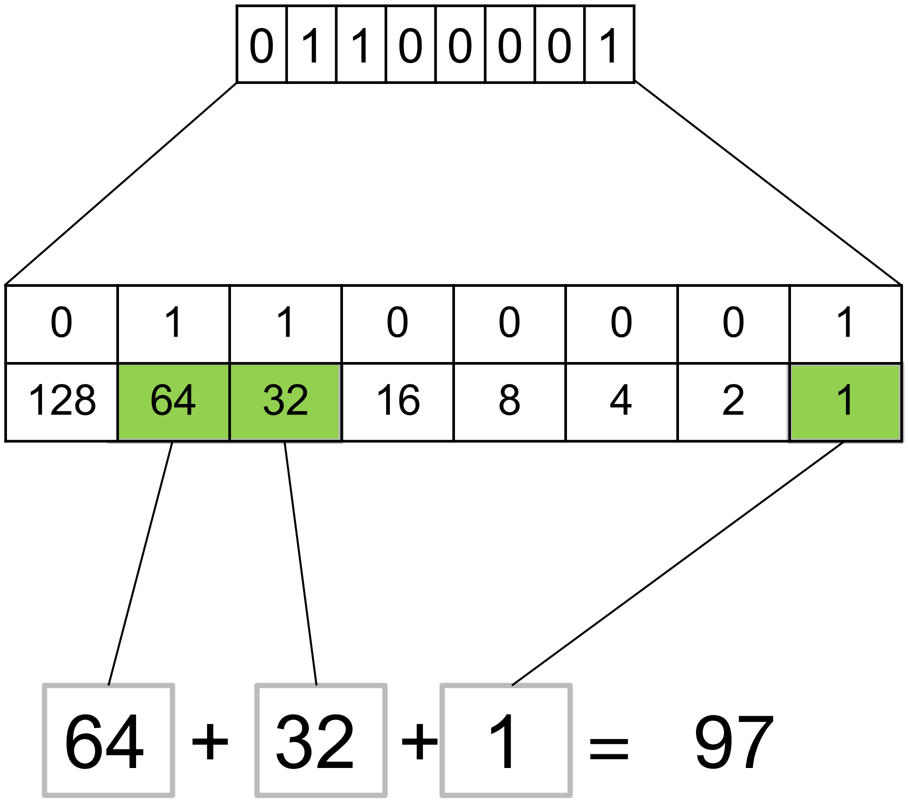
Example: The number 50 (a decimal number with base 10) is equivalent to:

128(=2^7) 64(=2^6) 32(=2^5) 16(=2^4) 8(=2^3) 4(=2^2) 2(=2^1) 1(=2^0)

That is 50 in binary is : 00110010

Example1: Try converting 97 into binary!!!!!

0 1 1 0 0 0 0 1



0	1	1	0	0	0	0	1

0	1	1	0	0	0	0	1
128	64	32	16	8	4	2	1

$\boxed{64} + \boxed{32} + \boxed{1} = 97$

Storing characters

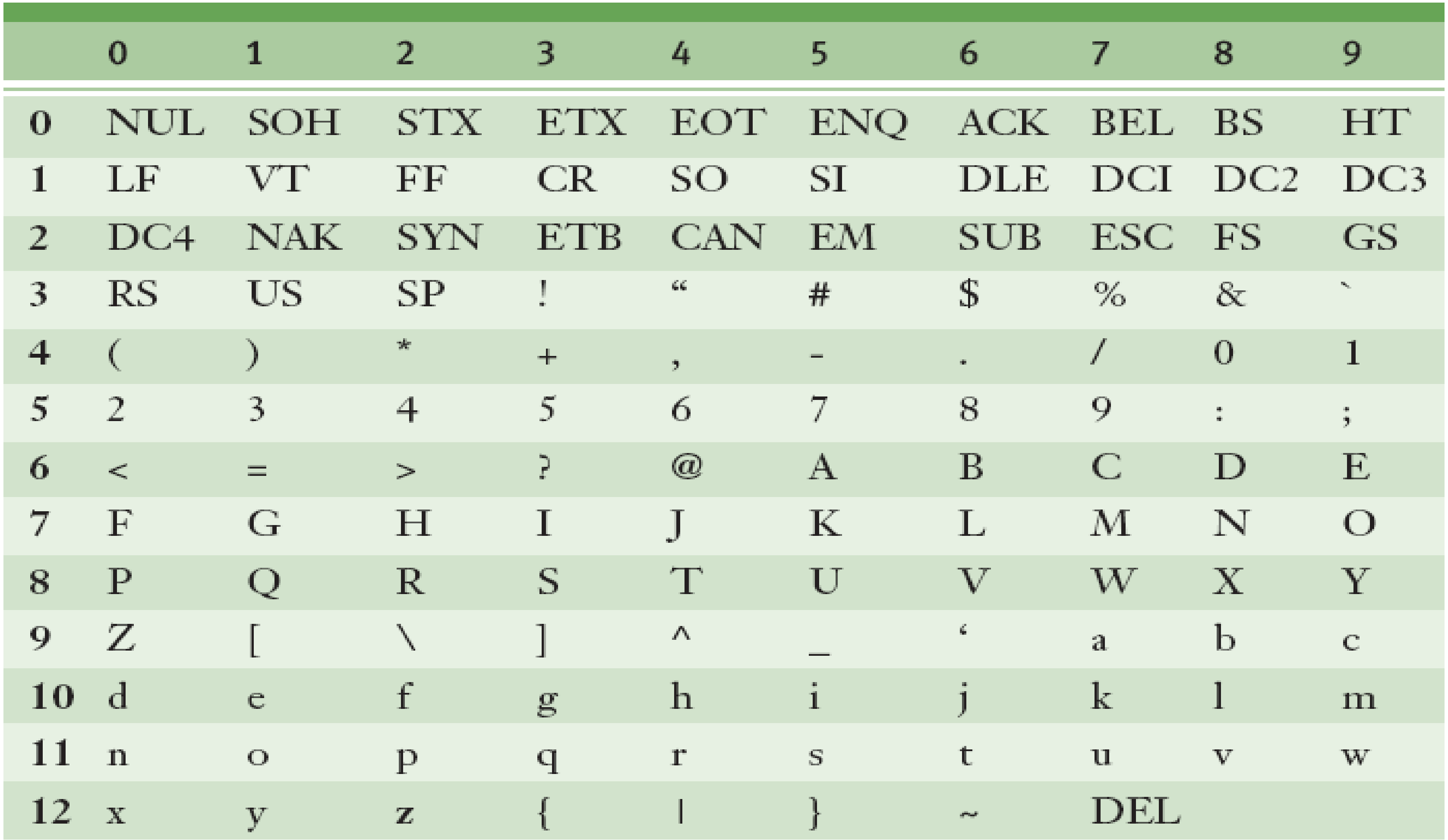
- Any data that is stored in a computers memory must be stored as a binary number. This includes characters, alphabets, symbols, punctuations etc….
- When a character is stored in a computer, it is first converted to a numeric code. The numeric code is then converted to a binary number.

Example: **A** ➔ numeric code ➔ binary number

Character Sets

- Character literals in python look like strings and are of string types
- They belong to character sets – ASCII set (128 codes)
- ASCII set encodes each keyboard characters
  - American Standard Code for Information Interchange
  - The digits in the left column represent the leftmost digits of the ASCII Code.
  - The digit in the top row are the rightmost digits.
  - ASCII code for ‘A’ = 65

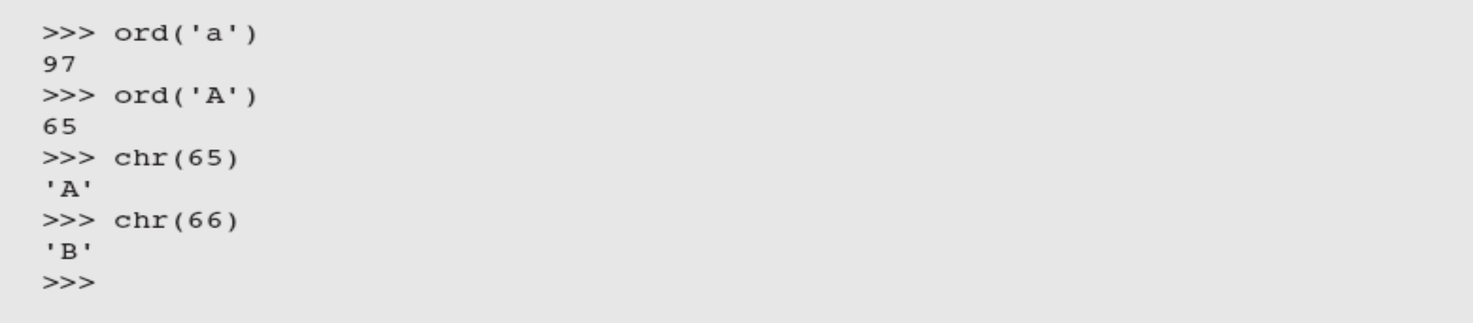
Character Sets



	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DCI	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	“	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Character Sets (continued)

- In Python, character literals look just like string literals and are of the string type
  - They belong to several different **character sets**, among them the **ASCII set**
  - ASCII character set maps to set of integers
- **ord** and **chr** convert characters to and from ASCII



```
>>> ord('a')
97
>>> ord('A')
65
>>> chr(65)
'A'
>>> chr(66)
'B'
>>>
```

Example: if you want to shift three places to the right of the letter ‘A’, simply write: chr(ord(‘A’) + 3)

Variables and the Assignment Statement – question !

#It is typical that you have a first name and a surname.

Write a python program that prints the first name followed by the surname, making sure that there is a blank space between the first name and the surname.

Your program should store the first name in a variable called firstName and the surname in a variable secondName.

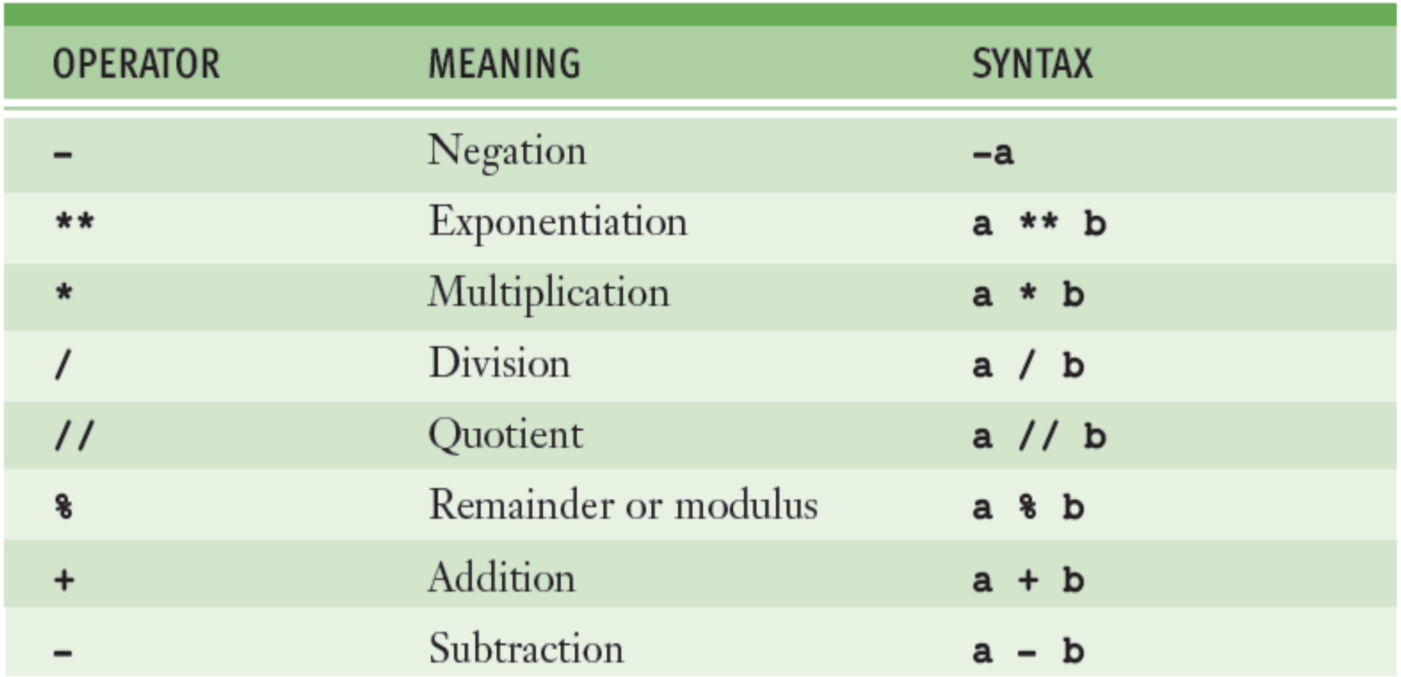
You may use additional variables for the task.

Expressions

- Variable reference evaluates to the variable’s current value
- **Expressions** provide easy way to perform operations on data values to produce other values
- When entered at Python shell prompt:
  - an expression’s operands are evaluated
  - its operator is then applied to these values to compute the value of the expression

Arithmetic Expressions

- An **arithmetic expression** consists of operands and operators combined in a manner that is already familiar to you from learning algebra

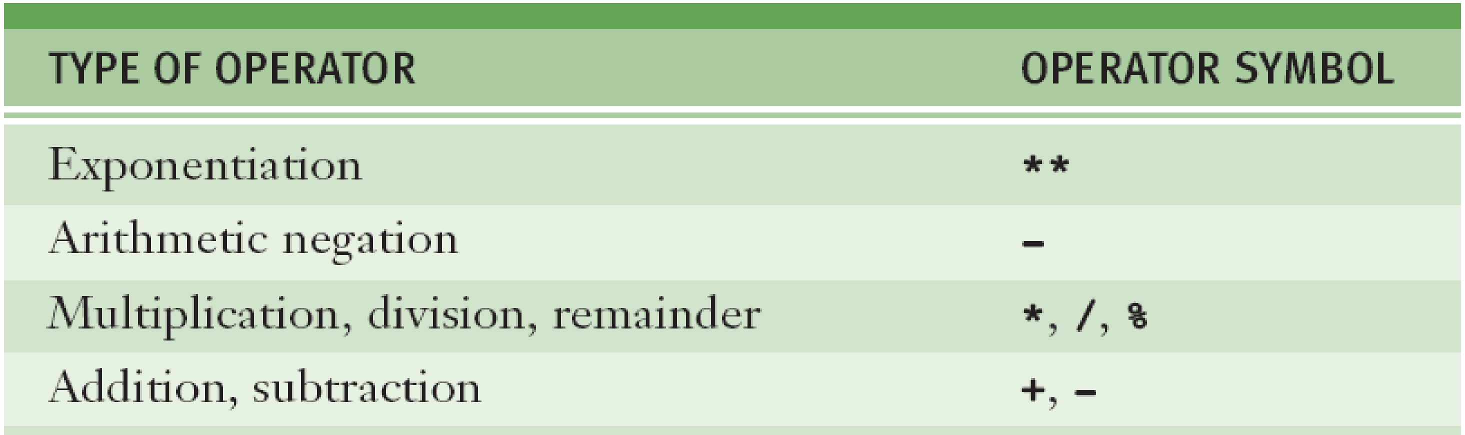


OPERATOR	MEANING	SYNTAX
`-`	Negation	`-a`
`**`	Exponentiation	`a ** b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`//`	Quotient	`a // b`
`%`	Remainder or modulus	`a % b`
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`

Arithmetic Expressions (continued)

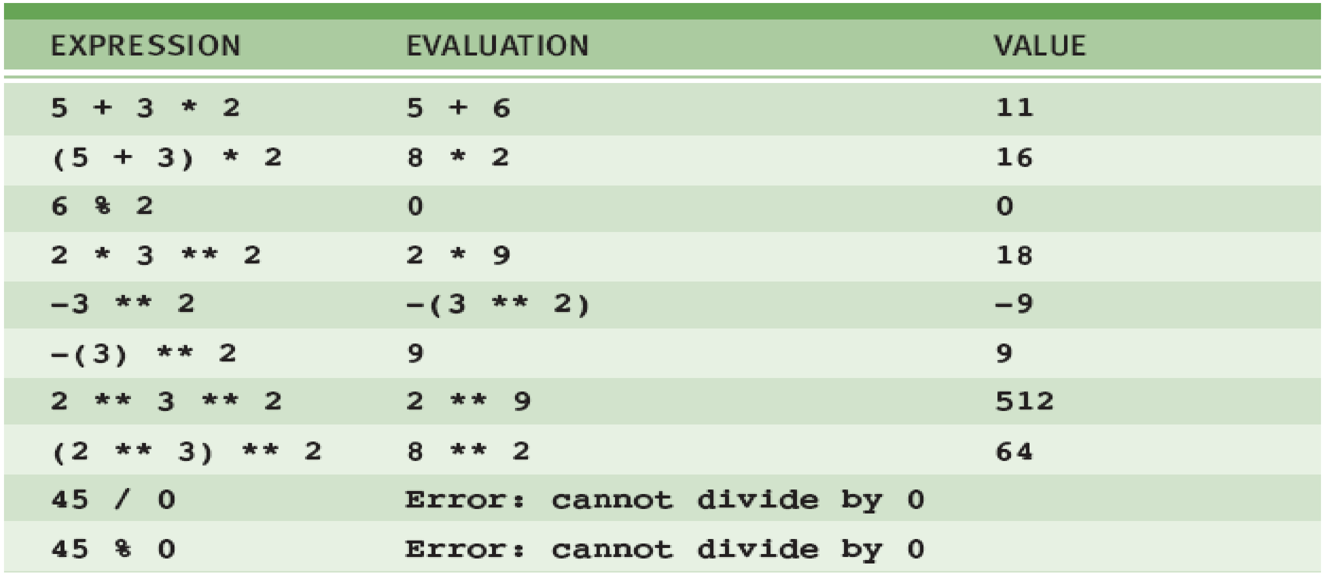
- **Precedence rules**:
  - ** has the highest precedence and is evaluated first
  - Unary negation is evaluated next
  - *, /, and % are evaluated before + and -
  - + and - are evaluated before equal to (=)
  - With two exceptions, operations of equal precedence are **left associative**, so they are evaluated from left to right
    - Exponentiation (**) and assignment (=) are **right associative**
  - You can use parenthesis ( ) to change the order of evaluation as parenthesis takes precedence.

Precedence rule for arithmetic operators (continued)



TYPE OF OPERATOR	OPERATOR SYMBOL
Exponentiation	**
Arithmetic negation	-
Multiplication, division, remainder	*, /, %
Addition, subtraction	+, -

Arithmetic Expressions (continued)



EXPRESSION	EVALUATION	VALUE
5 + 3 * 2	5 + 6	11
(5 + 3) * 2	8 * 2	16
6 % 2	0	0
2 * 3 ** 2	2 * 9	18
-3 ** 2	-(3 ** 2)	-9
-(3) ** 2	9	9
2 ** 3 ** 2	2 ** 9	512
(2 ** 3) ** 2	8 ** 2	64
45 / 0	Error: cannot divide by 0	
45 % 0	Error: cannot divide by 0	

Syntax error: set of rules for constructing well formed expressions in a language (error when an expression or sentence is not well formed).

Semantic error: detected when the action that an expression describes cannot be carried out, even if the expression is syntactically correct.

Example: 45%0 is a **semantic error**

Arithmetic calculations – Quiz 4

#Let **x = 8** and **y = 2**. Write the values of the following expressions:

a. x + y * 3

b. (x + y) * 3

c. x ** y

d. x % y

e. x / 12.0

f. x // 6

g. 3 + 4 ** 2//5

Arithmetic calculations – Quiz 4:

#Let x = 8 and y = 2. Write the values of the following expressions:

a. x + y * 3 #14

b. (x + y) * 3 #30

c. x ** y #64

d. x % y #0

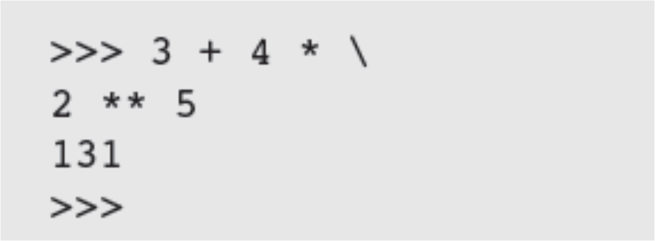
e. x / 12.0 #0.6666666666666666663

f. x // 6 #1

g. 3 + 4 ** 2//5 # 6

Arithmetic Expressions (continued)

- When **both operands of an expression are of the same numeric type**, the resulting value is also of that type
- When each operand is of a different type, the resulting value is of the more general type
  - Example: **3 / 4 is normal division and gives 0.75**, whereas **3 // 4 gives the quotient** which is 0 in this example

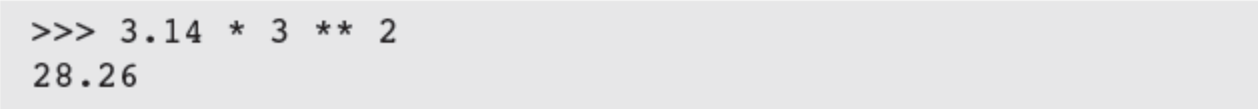


```
>>> 3 + 4 * \
2 ** 5
131
>>>
```

Note: For multi-line expressions, use a \

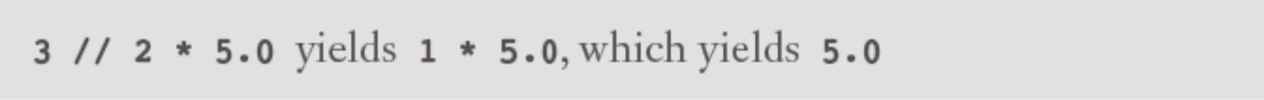
Mixed-Mode Arithmetic and Type Conversions

- **Mixed-mode arithmetic** involves integers and floating-point numbers:

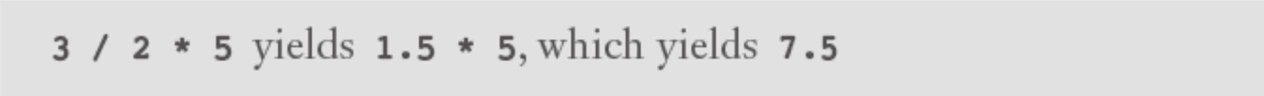


```
>>> 3.14 * 3 ** 2
28.26
```

- **Remember**—Python has different operators for quotient and exact division:



`3 // 2 * 5.0` yields `1 * 5.0`, which yields `5.0`



`3 / 2 * 5` yields `1.5 * 5`, which yields `7.5`

Tip:

- Use exact division
- Use a **type conversion function** with variables

Example: type conversion

Check example 1, 2, 3

Using print; input and type conversion

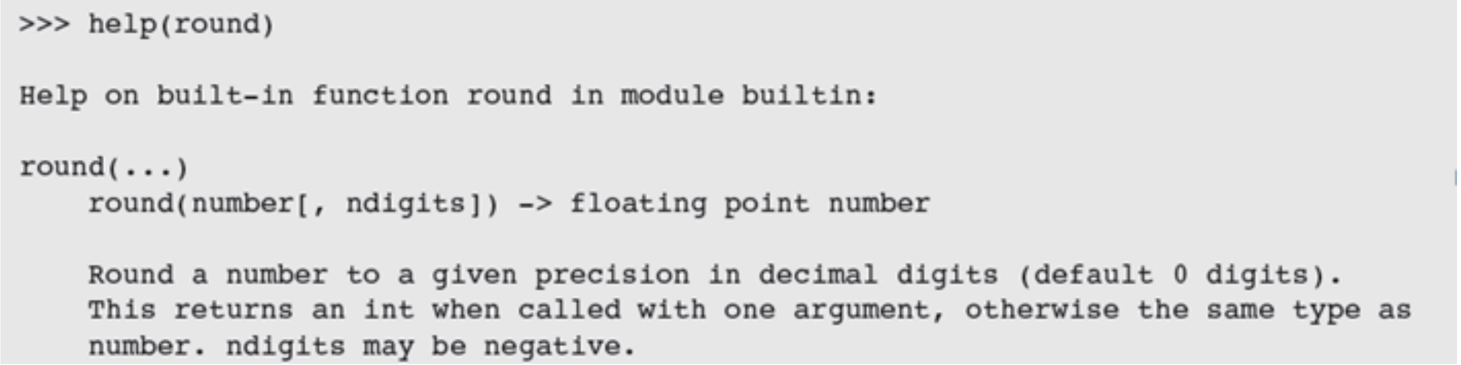
Using Functions and Modules - intro

- Python includes many useful functions, which are organized in libraries of code called **modules**

Note: Functions will be taught in detail later in the course

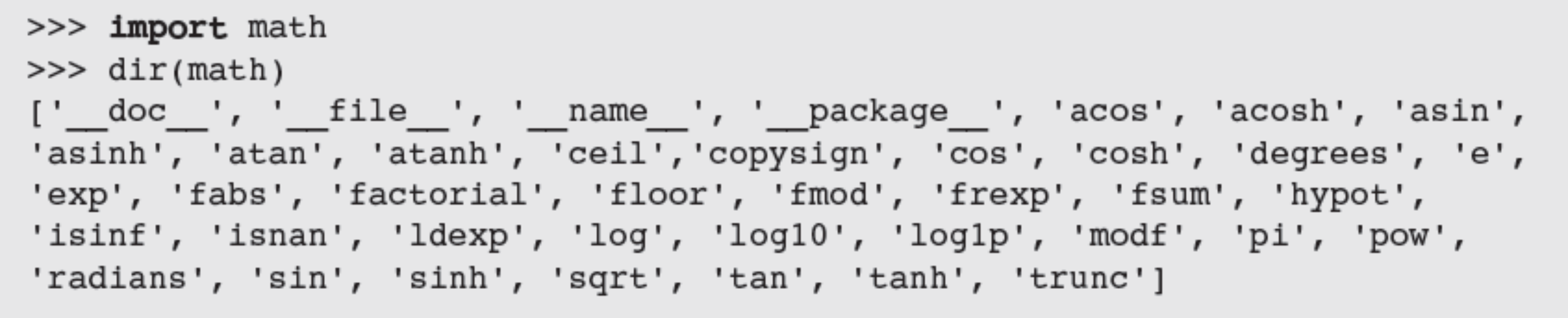
Calling Functions - Intro: Arguments and Return Values

- A **function** is chunk of code that can be called by name to perform a task
- Functions often require **arguments** or **parameters**
  - Arguments may be **optional** or **required**
- When function completes its task, it may **return a value** back to the part of the program that called it



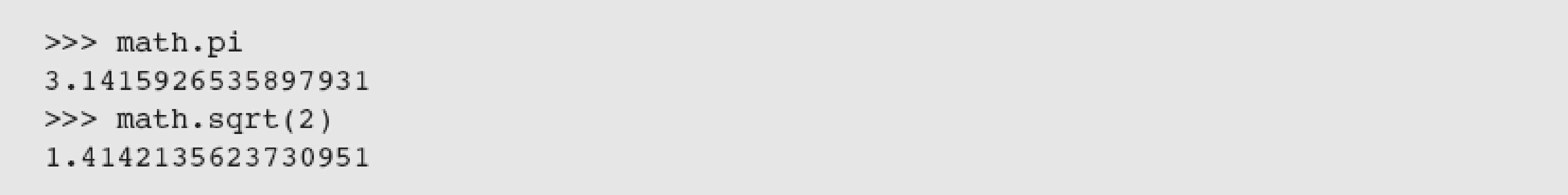
```
>>> help(round)

Help on built-in function round in module builtin:

round(...)
    round(number[, ndigits]) -> floating point number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the same type as
    number. ndigits may be negative.
```

The math Module



```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atanh', 'ceil','copysign', 'cos', 'cosh', 'degrees', 'e',
'exp', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'hypot',
'isinf', 'isnan', 'ldexp', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

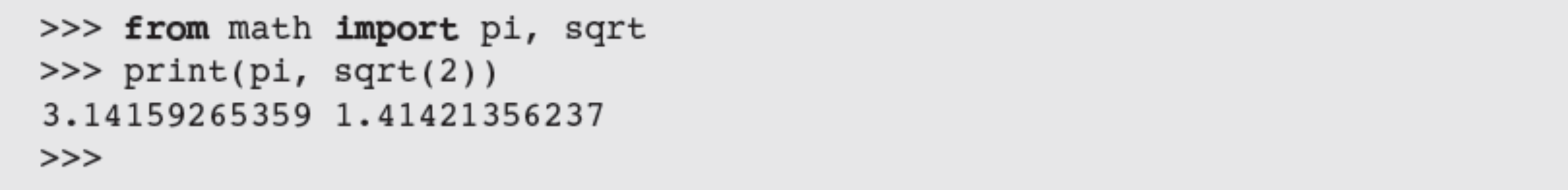
- To use a resource from a module, you write the name of a module as a qualifier, followed by a dot (.) and the name of the resource
  – Example: **math.pi**



```
>>> math.pi
3.1415926535897931
>>> math.sqrt(2)
1.4142135623730951
```

The math Module (continued)

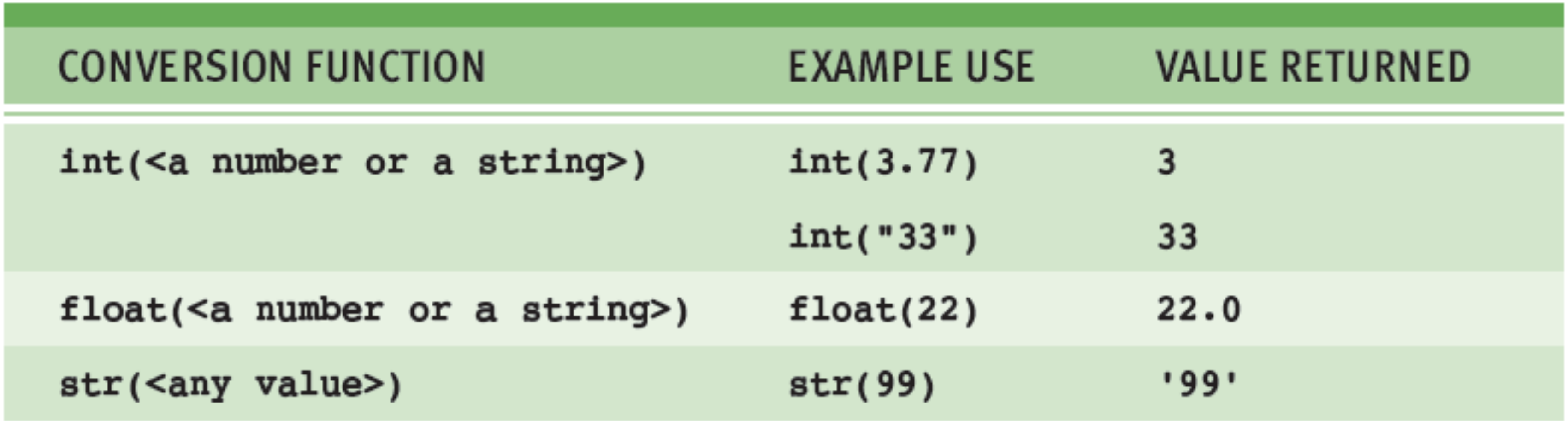
- You can avoid the use of the qualifier with each reference by importing the individual resources



```
>>> from math import pi, sqrt
>>> print(pi, sqrt(2))
3.14159265359 1.41421356237
>>>
```

- You may import all of a module’s resources to use without the qualifier
  - Example: **`from math import *`**

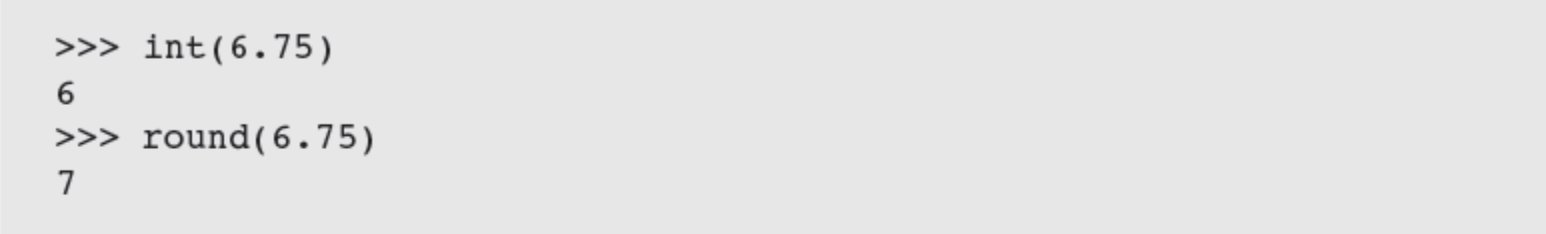
Mixed-Mode Arithmetic and Type Conversions (continued)



CONVERSION FUNCTION	EXAMPLE USE	VALUE RETURNED
`int(<a number or a string>)`	`int(3.77)`	`3`
	`int("33")`	`33`
`float(<a number or a string>)`	`float(22)`	`22.0`
`str(<any value>)`	`str(99)`	`'99'`

Mixed-Mode Arithmetic and Type Conversions (continued)

- Note that the **int** function converts a **float** to an **int** by truncation, not by rounding



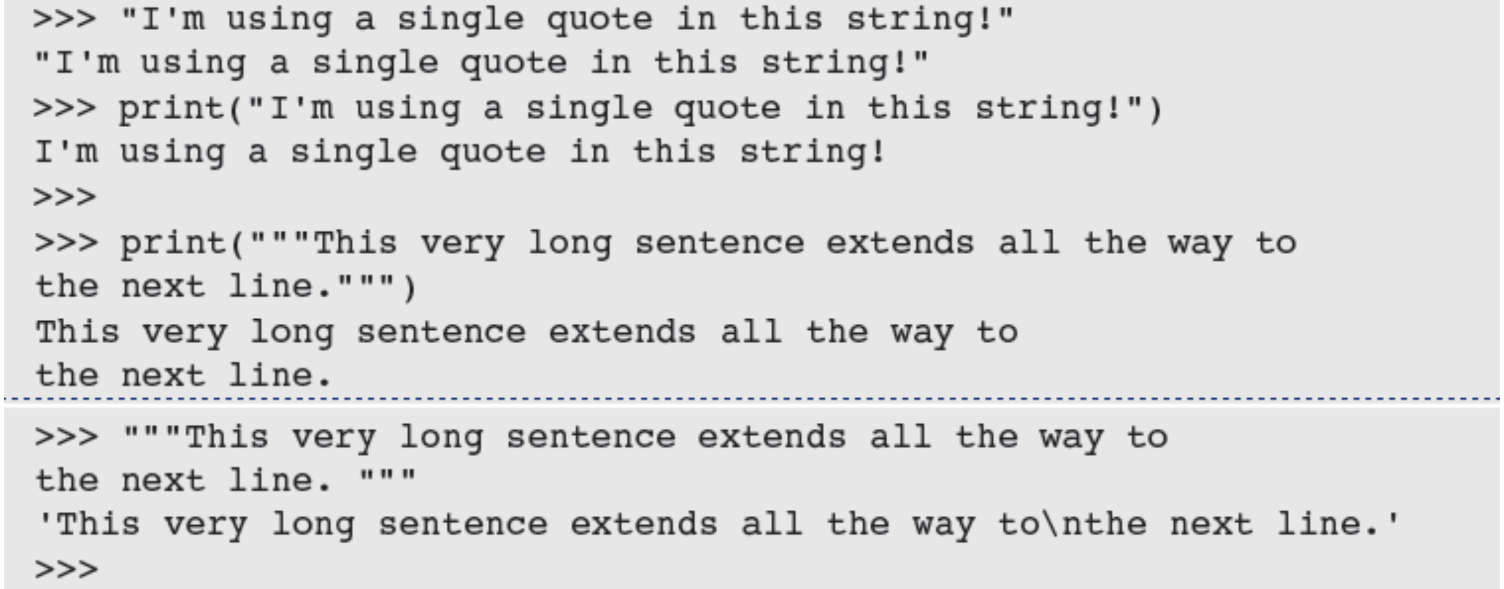
```
>>> int(6.75)
6
>>> round(6.75)
7
```

Examples

Simple example programs with and without math function 6, 7, 8

String Literals

- A string literal is a sequence of characters enclosed in single or double quotation marks
- '' and "" represent the **empty string**
- Use ''' and """ for multi-line paragraphs



```
>>> "I'm using a single quote in this string!"
"I'm using a single quote in this string!"
>>> print("I'm using a single quote in this string!")
I'm using a single quote in this string!
>>>
>>> print("""This very long sentence extends all the way to
the next line.""")
This very long sentence extends all the way to
the next line.
```

```
>>> """This very long sentence extends all the way to
the next line. """
'This very long sentence extends all the way to\nthe next line.'
>>>
```

Escape Characters

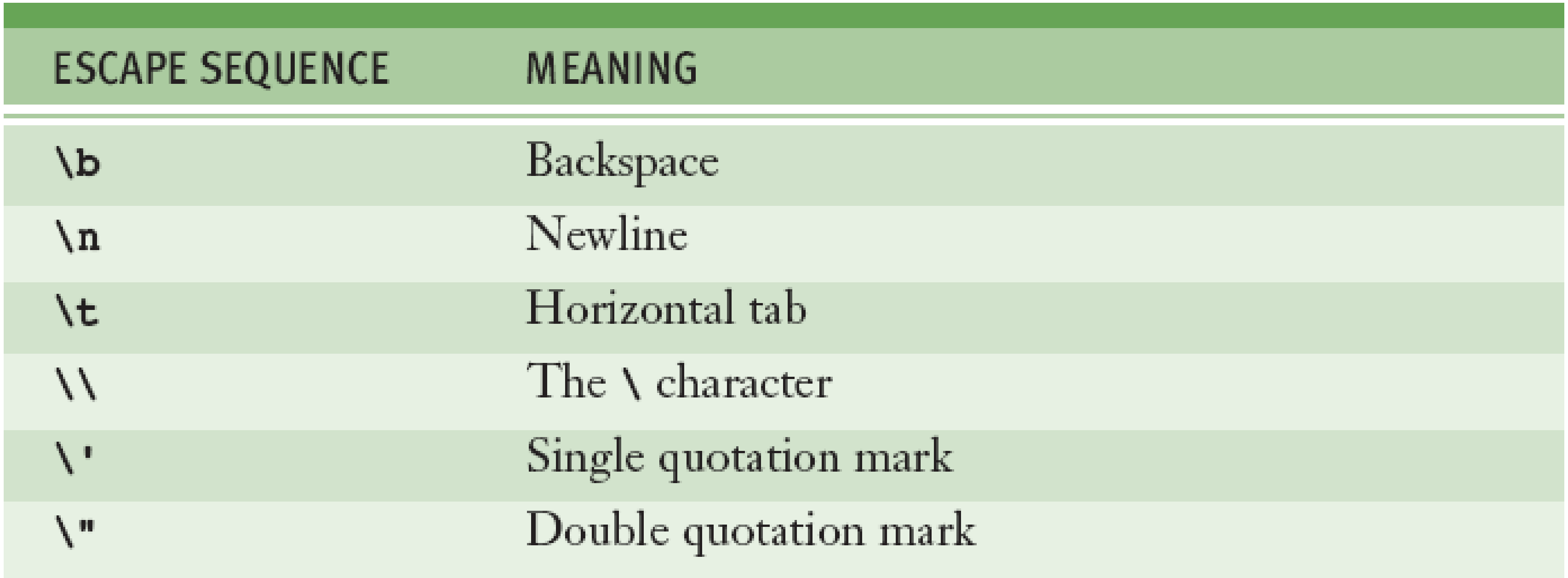
- It is a special character that is preceded with a backslash(\) appearing inside a string literal.
- When a string literal that contains the escape character is printed, the escape characters are treated as special commands that are embedded in the string.

Example: >>>print (‘one\ntwo\nthree’)

```
one
two
three
```

Escape Sequences

- The newline character **\n** is called an **escape sequence**



ESCAPE SEQUENCE	MEANING
`\b`	Backspace
`\n`	Newline
`\t`	Horizontal tab
`\\`	The \ character
`\'`	Single quotation mark
`\"`	Double quotation mark

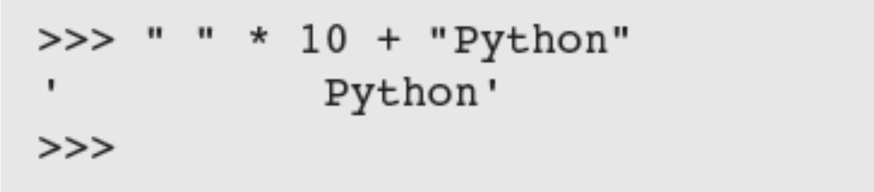
Escape Characters

```
>>> print (‘mon\ttues\twed’)
      mon     tues      wed
```

Lets try some more examples on IDLE!!!!

String Concatenation

- You can join two or more strings to form a new string using the concatenation operator **+**
- The * operator allows you to build a string by repeating another string a given number of times



```
>>> " " * 10 + "Python"
'          Python'
>>>
```

Quiz: 5

# write the output of the following python statements:

a. “hell_no”

b. “hell_no” * 10

c. “hell_no” + “ “ * 10

d. (“hell_no” + “ “) * 10

String concatenation - Quiz: 5

# write the output for the following python statements:

a. “hell_no”

b. “hellno” * 5

c. “hellno” + “ “ * 5

d. (“hellno” + “ “) * 5

Soln:

a. ???

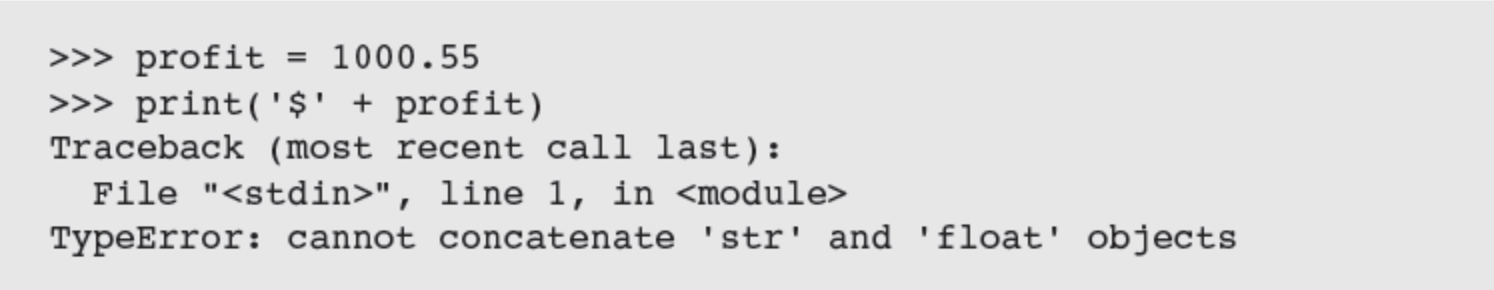
b. 'hellnohellnohellnohellnohellno’

c. ‘hellno   ’

d. 'hellno hellno hellno hellno hellno '

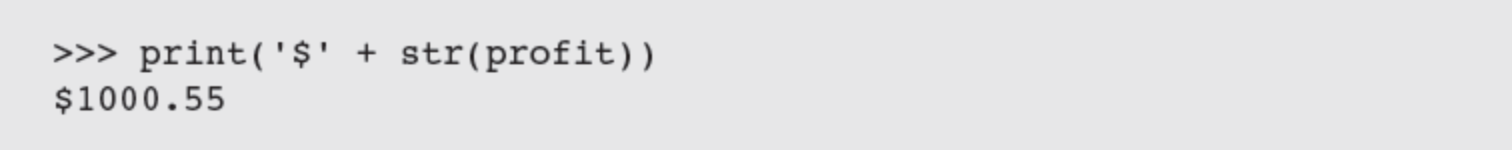
Mixed-Mode Arithmetic and Type Conversions (continued)

- Type conversion also occurs in the construction of strings from numbers and other strings



```
>>> profit = 1000.55
>>> print('$' + profit)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'float' objects
```

- Solution: use `str` function

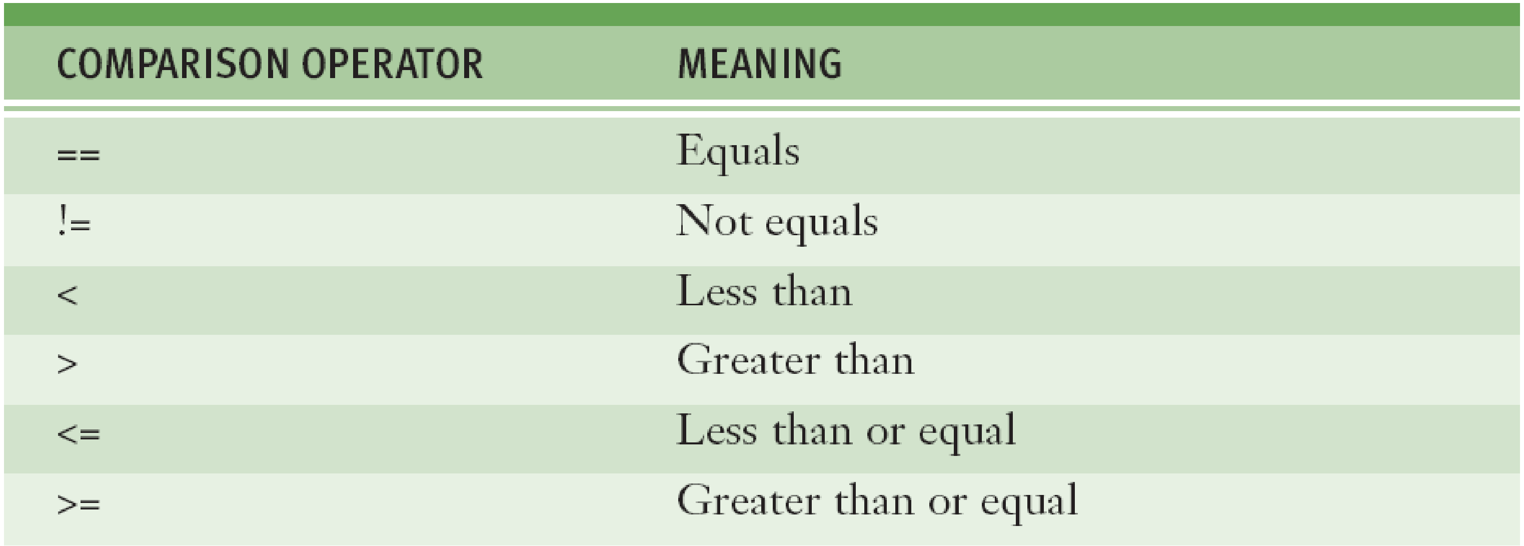


```
>>> print('$' + str(profit))
$1000.55
```

- Python is a strongly **typed** programming language

The Boolean Type, Comparisons, and Boolean Expressions

- **Boolean data type** consists of two values: true and false (typically through standard **True/False**)

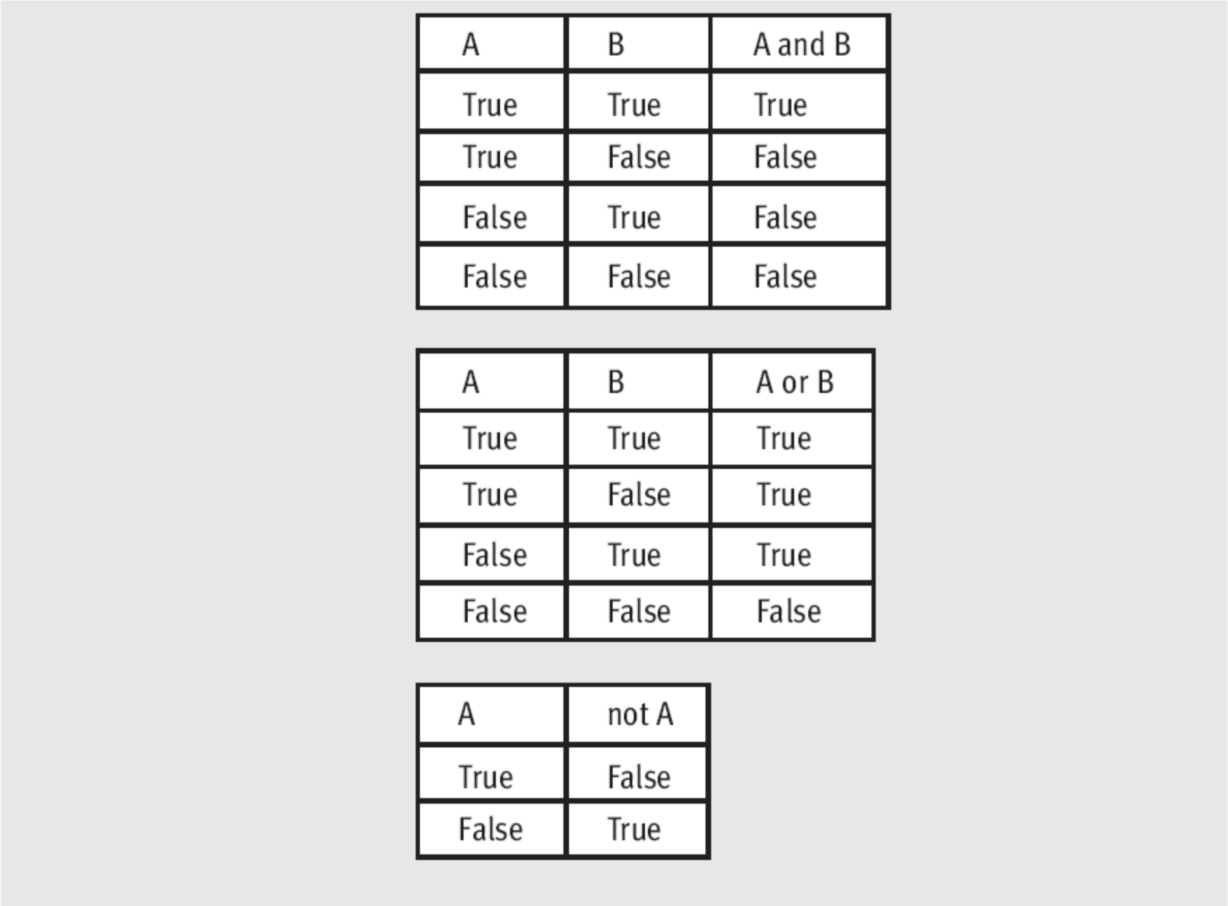


COMPARISON OPERATOR	MEANING
==	Equals
!=	Not equals
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

[TABLE 3.2] The comparison operators

- Example: 4 != 4 evaluates to False

Logical Operators and Compound Boolean Expressions (continued)



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

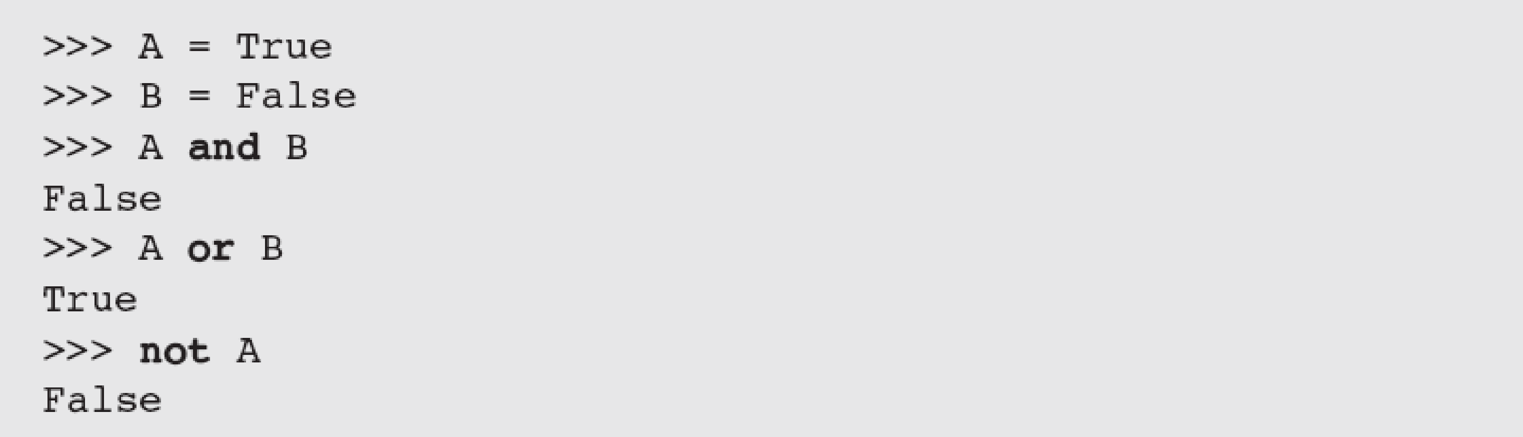
A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

A	not A
True	False
False	True

[FIGURE 3.4] The truth tables for **and**, **or**, and **not**

Logical Operators and Compound Boolean Expressions (continued)

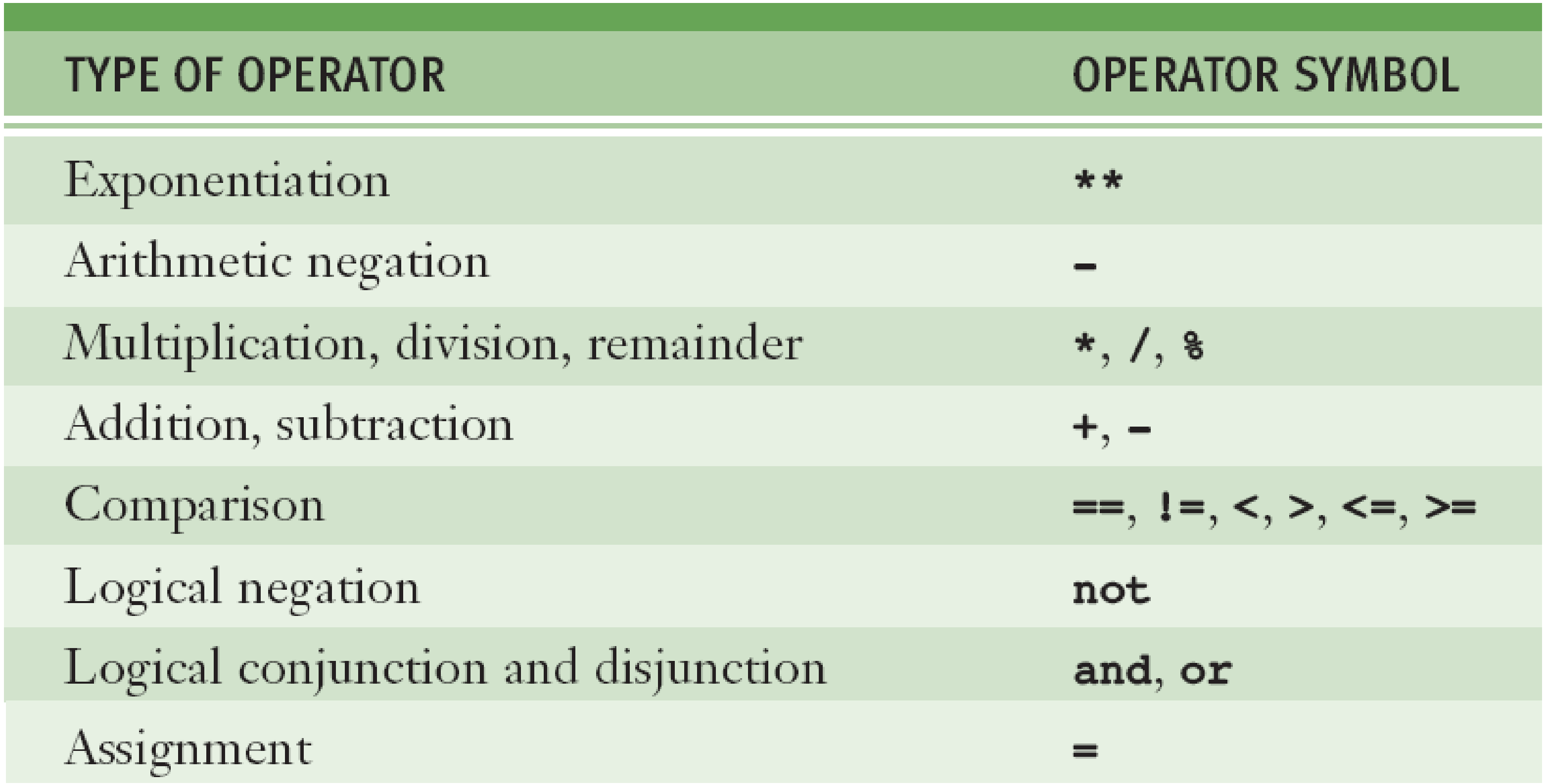
- Next example verifies some of the claims made in the previous truth tables:



```
>>> A = True
>>> B = False
>>> A and B
False
>>> A or B
True
>>> not A
False
```

- The logical operators are evaluated after comparisons but before the assignment operator
  - **`not`** has higher precedence than **`and`** and **`or`**

Logical Operators and Compound Boolean Expressions (continued)



TYPE OF OPERATOR	OPERATOR SYMBOL
Exponentiation	`**`
Arithmetic negation	`-`
Multiplication, division, remainder	`*`, `/`, `%`
Addition, subtraction	`+`, `-`
Comparison	`==`, `!=`, `<`, `>`, `<=`, `>=`
Logical negation	`not`
Logical conjunction and disjunction	`and`, `or`
Assignment	`=`

[TABLE 3-4] Operator precedence, from highest to lowest

Logical operation evaluation: Example

- In **(A and B)**, if **A** is false, then so is the expression, and there is no need to evaluate **B**
- In **(A or B)**, if **A** is true, then so is the expression, and there is no need to evaluate **B**

Quiz: 5

Fill in the blanks:

- A compound boolean expression created with a the ----------- operator is true only if both of its sub expressions are true:

Is it the or, and, not?????

The -------- operator takes a boolean expression as its operand and reverses its logical value.

Is it the or, not or and operators?????