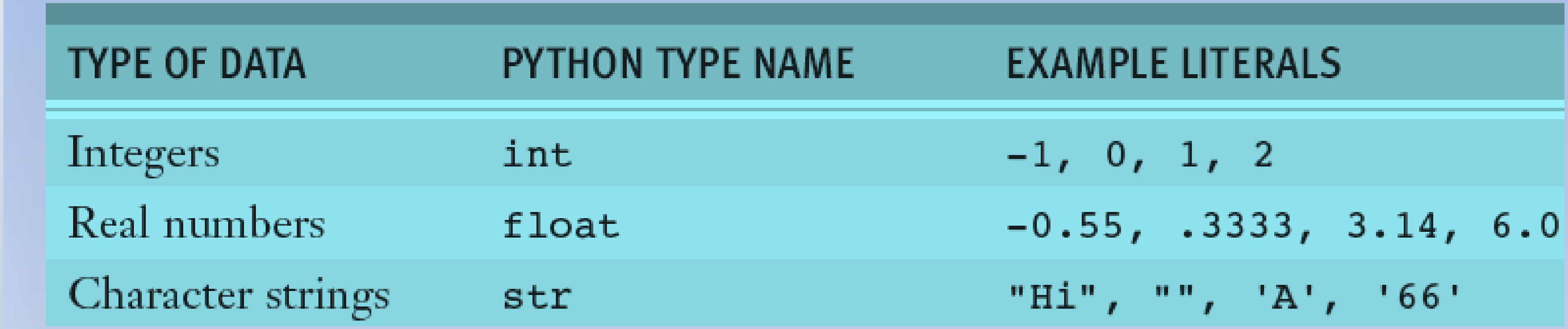
Data Types

- A **data type** consists of a set of values and a set of operations that can be performed on those values
- The basic data types are int, float and str
  - **`int`** and **`float`** are **numeric data types**
  - **str** is **string data type**
- A **literal** is the way a value of a data type looks to a programmer
  - numbers are known as numeric literals
  - strings are known as string literals

Data Types



TYPE OF DATA	PYTHON TYPE NAME	EXAMPLE LITERALS
Integers	`int`	`-1, 0, 1, 2`
Real numbers	`float`	`-0.55, .3333, 3.14, 6.0`
Character strings	`str`	`"Hi", "", 'A', '66'`

Numeric Literals

- Numbers are referred to as *numeric literals*
- Two Types of numbers: *int* and *float*
  - Whole number written without a decimal point called an **int**
  - Number written with a decimal point called a **float**

String Literals

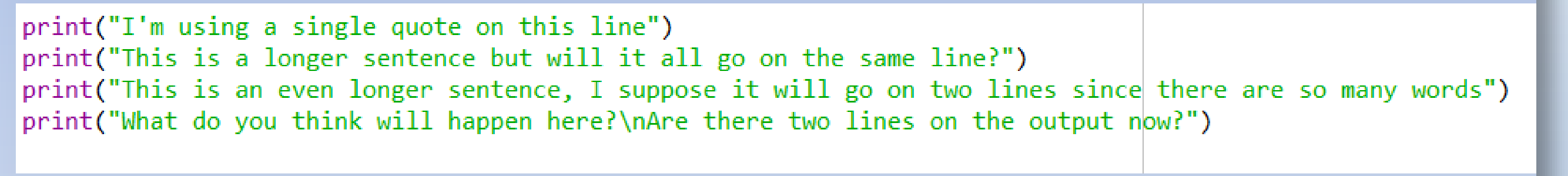
- In Python, a string literal is a sequence of characters enclosed in single or double quotation marks
- '' and "" represent the **empty string**

Processing

- Programs usually accept inputs from a source, process them, and output results to a destination
- For both input and output Python uses “built-in” functions
- Output is done by using ***print()***
  - Prints objects to the output stream
  - In the simplest form you can print a string of characters (enclosed in quotes)or numbers
- Input is done by using ***input()***
  - Accepts input from the user
  - Optionally prompts a user to input data

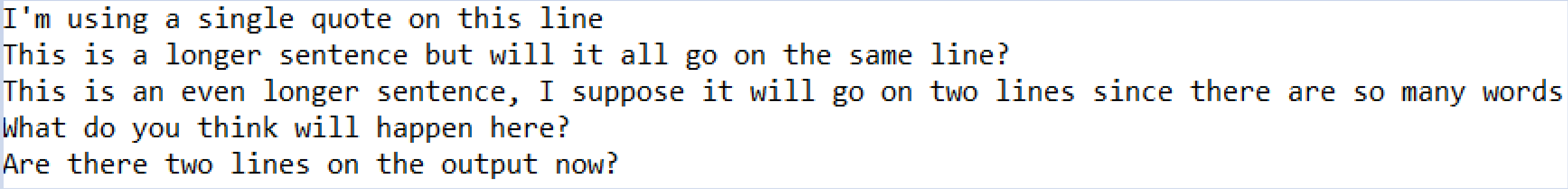
The *print* Function

- Used to display information on the monitor
  - To display a series of characters, enclose them within quotes



```
print("I'm using a single quote on this line")
print("This is a longer sentence but will it all go on the same line?")
print("This is an even longer sentence, I suppose it will go on two lines since there are so many words")
print("What do you think will happen here?\nAre there two lines on the output now?")
```

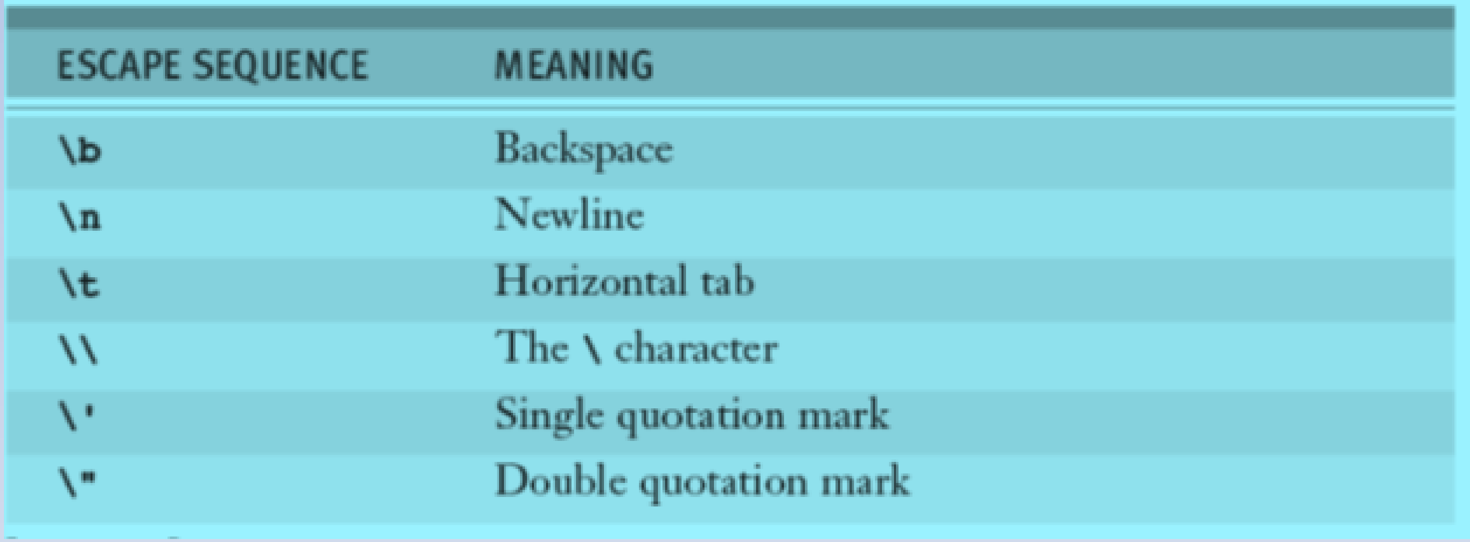
- Output from the above lines of code



```
I'm using a single quote on this line
This is a longer sentence but will it all go on the same line?
This is an even longer sentence, I suppose it will go on two lines since there are so many words
What do you think will happen here?
Are there two lines on the output now?
```

Escape Sequences

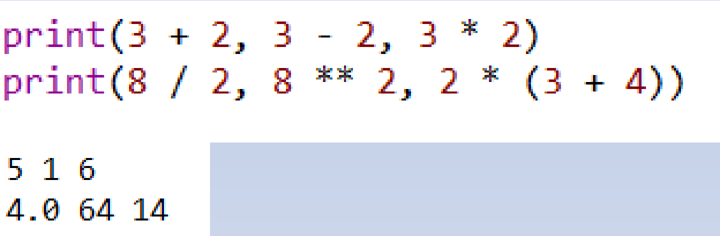
- The newline character **\n** is called an **escape sequence**



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

Printing numbers and expressions

- If *n* is a number, **`print(`*`n`*`)`** displays number n
- The print function can display the result of "evaluated expressions"
- A single print function can also display several values (default separator is a space character)



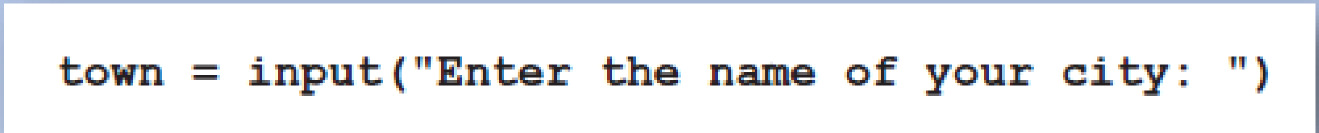
```
print(3 + 2, 3 - 2, 3 * 2)
print(8 / 2, 8 ** 2, 2 * (3 + 4))
```

```
5 1 6
4.0 64 14
```

- *You can also use the "+" symbol to print several string literals concatenated together*

The *input* Function

- Prompts the user to enter data

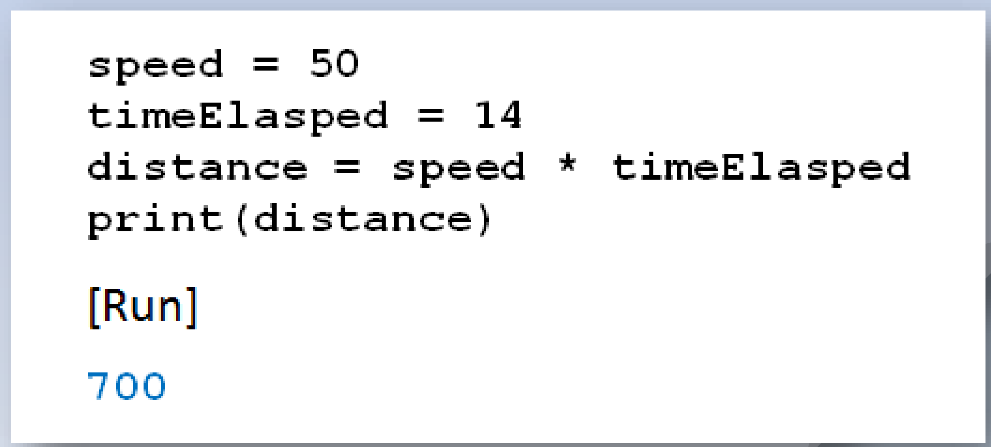


```
town = input("Enter the name of your city: ")
```

  - User types response, presses ENTER key
  - Entry assigned to variable on left

Variables

- A *variable* is something that holds a value that may change
- In simplest terms, a variable is just a box that you can put stuff
- Example: Program uses speed, time … calculates distance



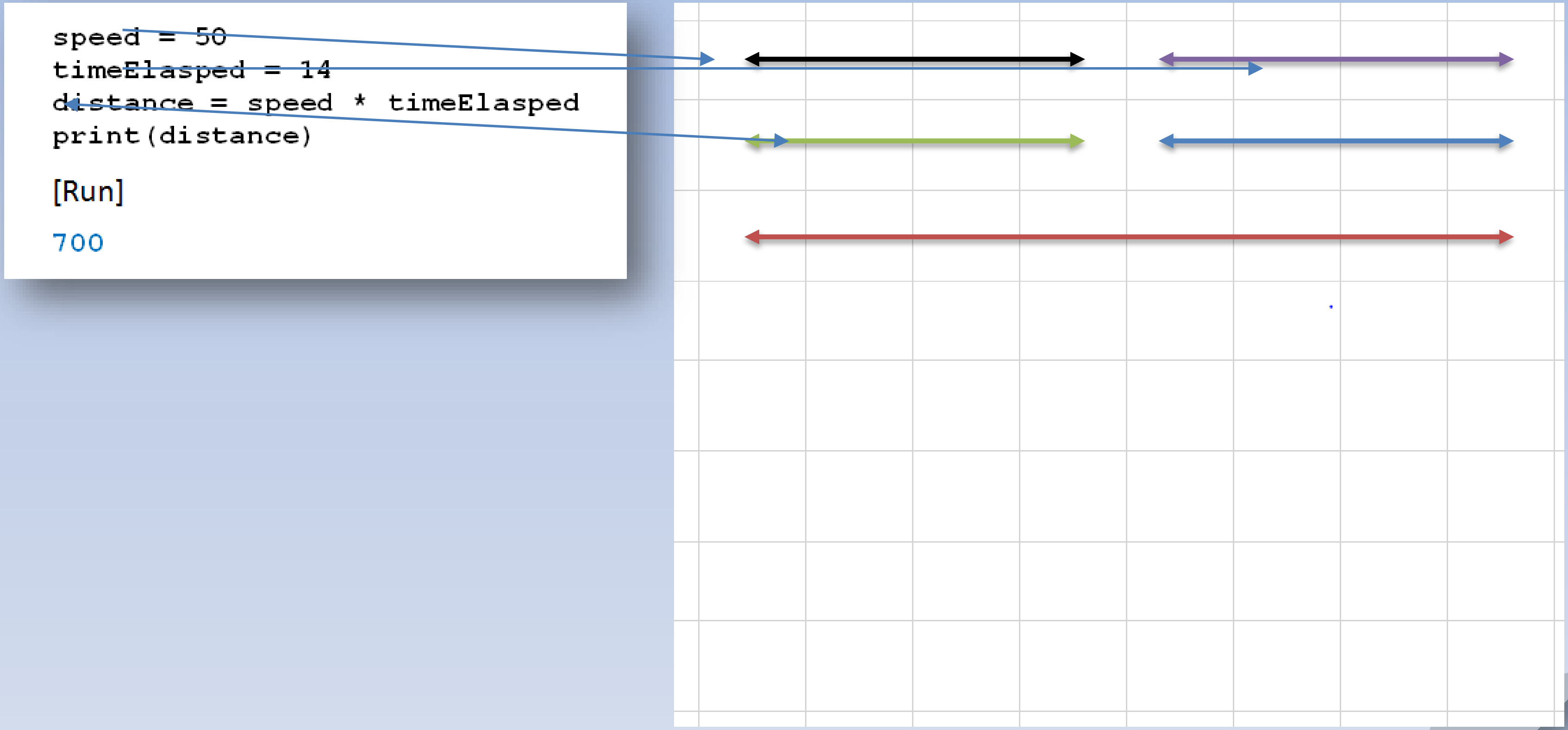
```
speed = 50
timeElasped = 14
distance = speed * timeElasped
print(distance)
```

[Run]

```
700
```

Variables

- If we view memory as a grid of cells …



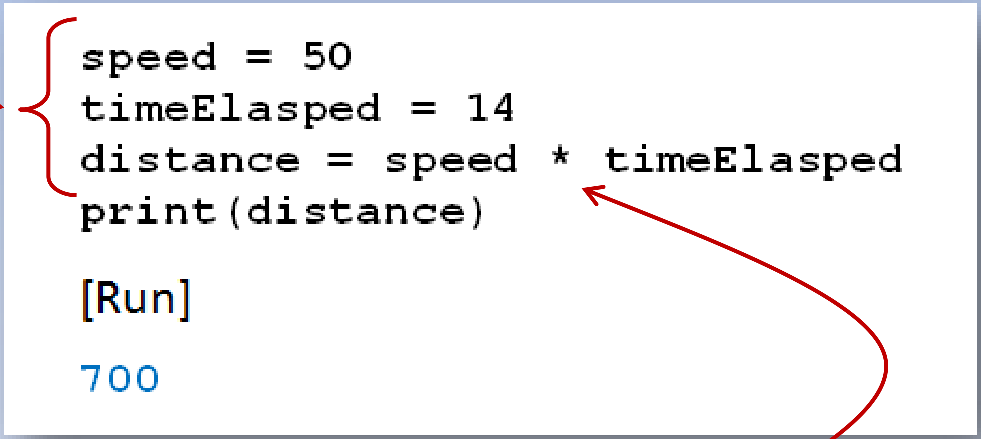
```
speed = 50
timeElasped = 14
distance = speed * timeElasped
print(distance)
```

[Run]

```
700
```

Variables

- Assignment statements



```
speed = 50
timeElasped = 14
distance = speed * timeElasped
print(distance)
```

[Run]

```
700
```

- Expression on right evaluated
  - That value assigned to variable

Variables

- Variable names in Python
  - Begin with letter or underscore _
  - Can only consist of letters, numbers, underscores
- Recommend using descriptive variable names
- Convention will be
  - Begin with lowercase
  - Use cap for additional “word” in the name (“camel casing”)
  - Example: **`rateOfChange`**

Variables

- Names in Python are case-sensitive
- There are several words, called **reserved words (**or **keywords)**
  - Have special meanings in Python
  - Cannot be used as variable names
  - Examples: **if**, **def**, **import**
  - See Appendix in most any text book

Constants & Variables

- Programmers use all uppercase letters for **symbolic constants**
  - Examples: **TAX_RATE** and **STANDARD_DEDUCTION**
- Variables receive initial values and can be reset to new values with an **assignment statement**

  <*variable name*> = <*expression*>

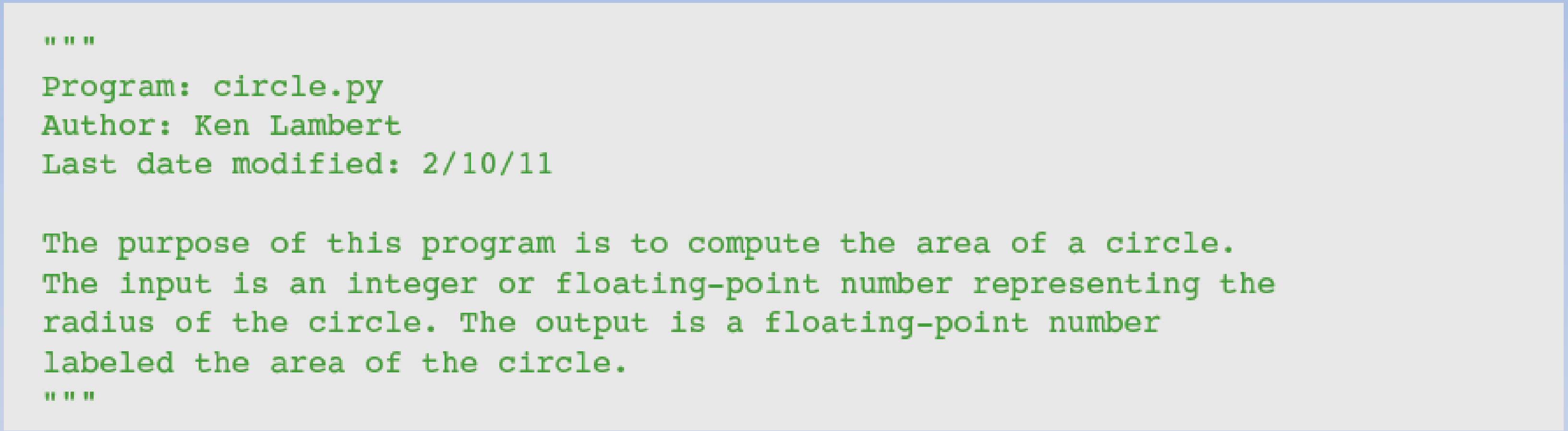
  - Subsequent uses of the variable name in expressions are known as **variable references**

Named Constants

- Program sometimes employs a special constant used several times in program
- Conventional programmers use constants as
  - Global values
  - Name written in uppercase letters with words separated by underscore
- In Python, programmer is responsible for not changing value of the constant (unlike most other languages)

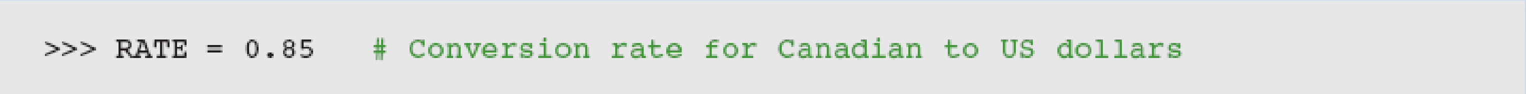
Program Comments and Docstrings

- **Docstring** example:



```
"""
Program: circle.py
Author: Ken Lambert
Last date modified: 2/10/11

The purpose of this program is to compute the area of a circle.
The input is an integer or floating-point number representing the
radius of the circle. The output is a floating-point number
labeled the area of the circle.
"""
```

- **End-of-line comment** example:



```
>>> RATE = 0.85   # Conversion rate for Canadian to US dollars
```

Numeric and Character Data

- The first applications of computers were to crunch numbers
- The use of numbers in many applications is still very important
- Text processing is by far the most common application of computing
  - E-mail, text messaging, Web pages, and word processing all rely on and manipulate data consisting of strings of characters

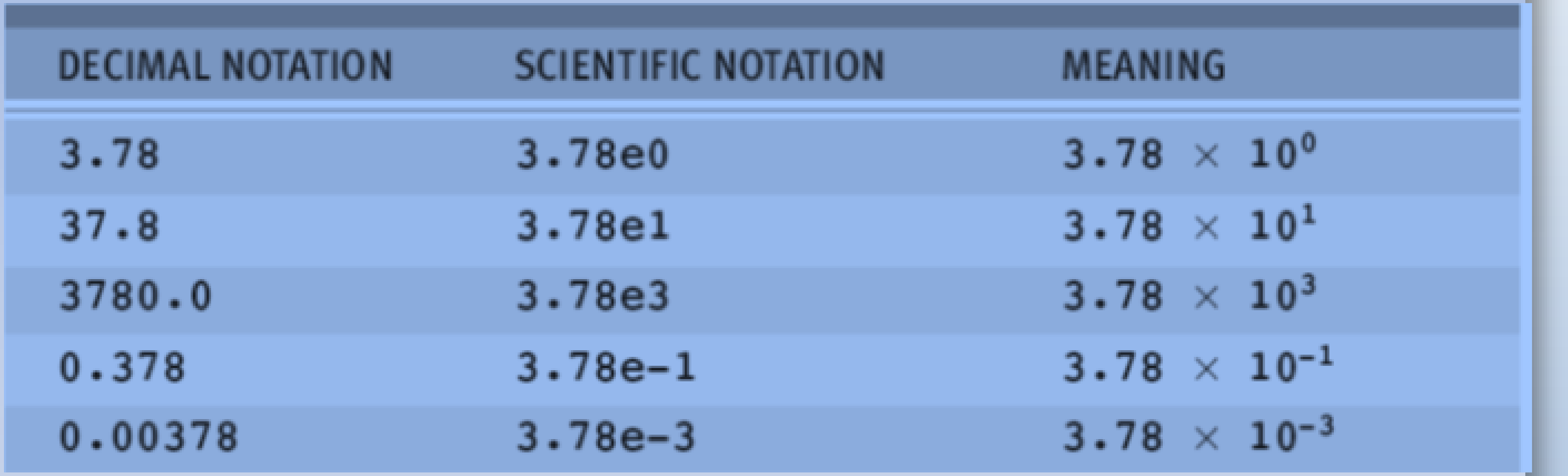
Integers

- In real life, the range of **integers** is infinite
- A computer's memory places a limit on magnitude of the largest positive and negative integers
  - Python's **`int`** typical range: -2^{31} to $2^{31} - 1$
- Integer literals are written without commas

Floating-Point Numbers

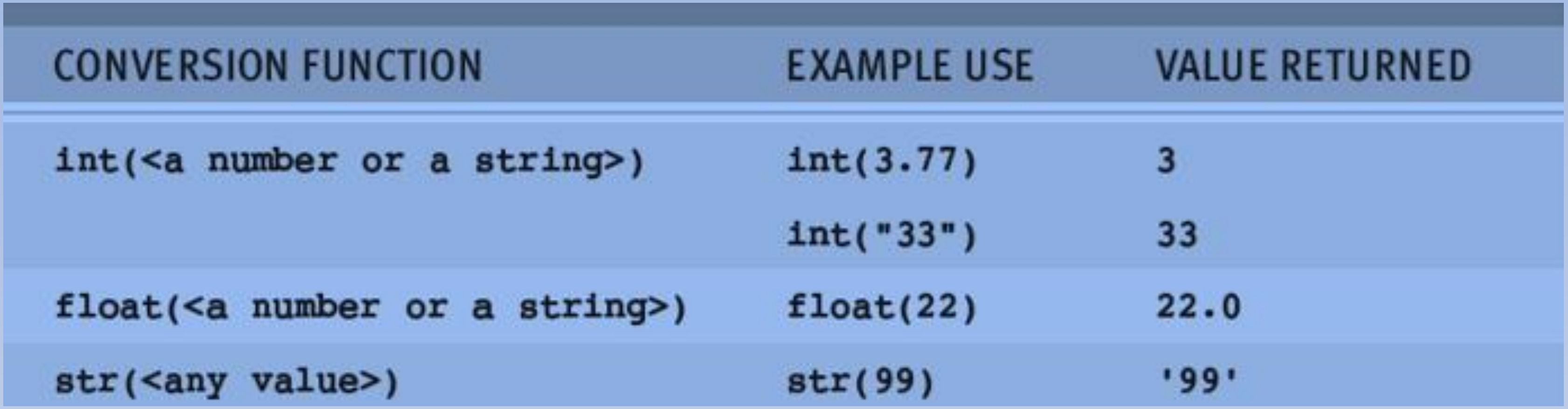
- Python uses **floating-point** numbers to represent real numbers
- Python's **`float`** typical range: -10^{308} to 10^{308} and
- Typical precision: 16 digits

Floating-Point Numbers



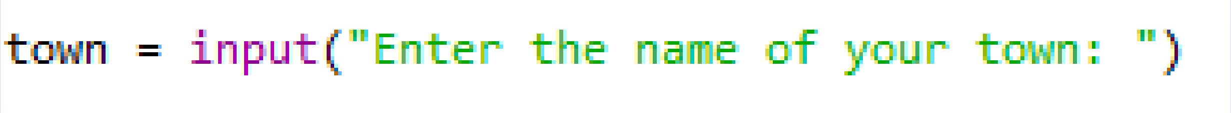
DECIMAL NOTATION	SCIENTIFIC NOTATION	MEANING
3.78	3.78e0	3.78×10^{0}
37.8	3.78e1	3.78×10^{1}
3780.0	3.78e3	3.78×10^{3}
0.378	3.78e-1	3.78×10^{-1}
0.00378	3.78e-3	3.78×10^{-3}

Type Conversions



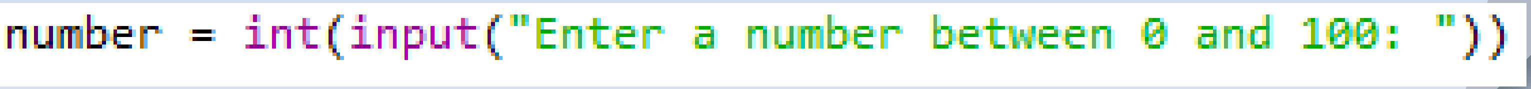
CONVERSION FUNCTION	EXAMPLE USE	VALUE RETURNED
int(<a number or a string>)	int(3.77)	3
	int("33")	33
float(<a number or a string>)	float(22)	22.0
str(<any value>)	str(99)	'99'

- Type conversion must be done while seeking numeric input from the user
- Input obtained is always a string literal



```
town = input("Enter the name of your town: ")
```

Versus



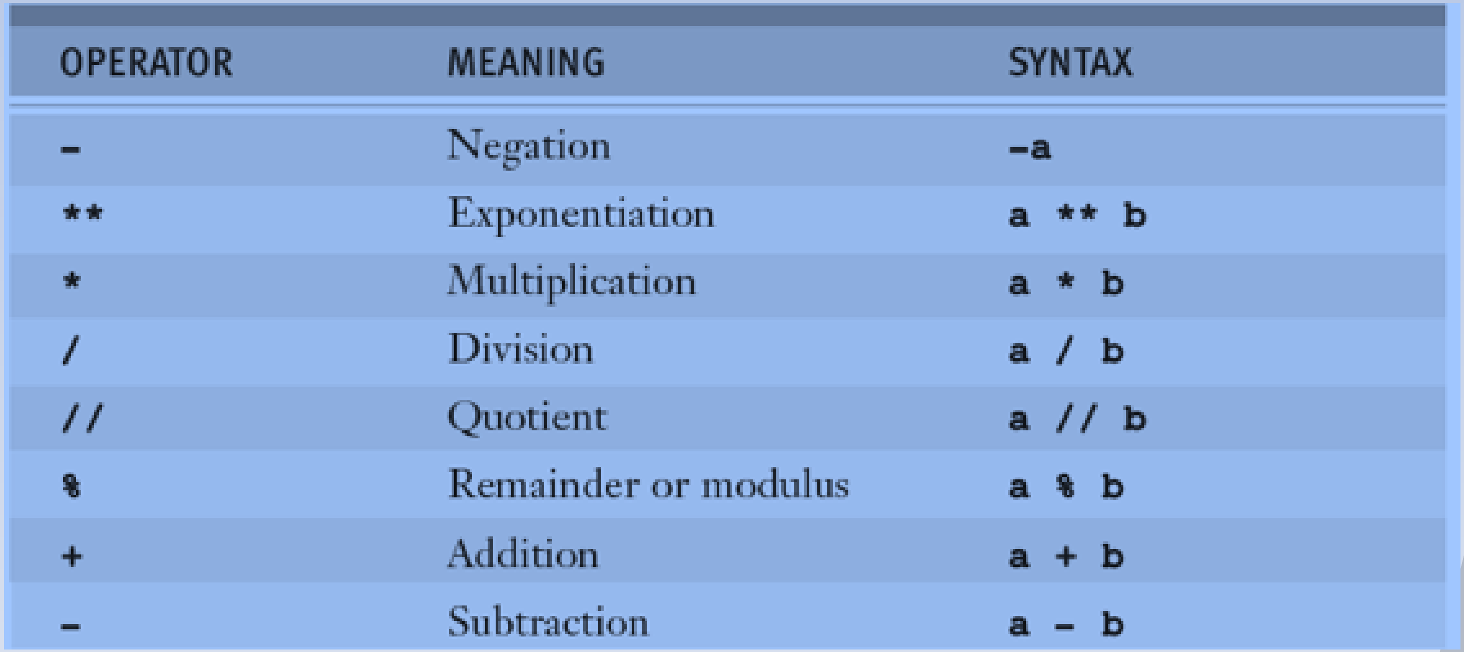
```
number = int(input("Enter a number between 0 and 100: "))
```

Expressions

- An expression is a combination of literals and/or variables (typically)
- A literal evaluates to itself
- A 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

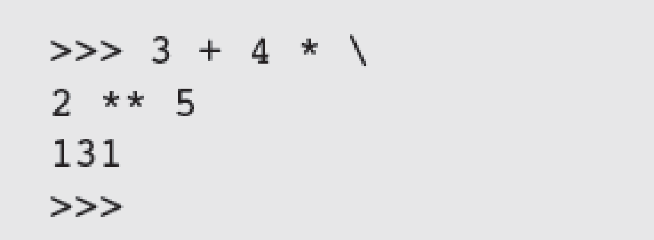
- **Precedence rules**:
  - ** has the highest precedence and is evaluated first
  - Unary negation is evaluated next
  - *, /, and % are evaluated before + and -
  - + and - are evaluated before =
  - With two exceptions, operations of equal precedence are **left associative**, so they are evaluated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

Arithmetic Expressions

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	

Arithmetic Expressions

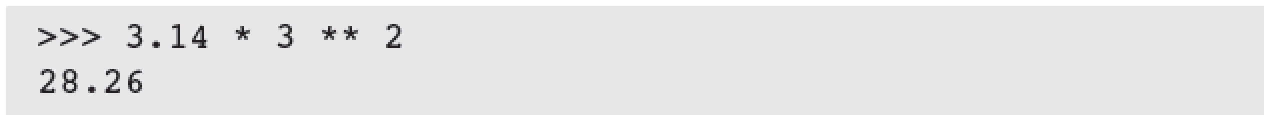
- When both operands of an expression are of the same numeric type, the resulting value is also of that type
- For multi-line expressions, use a \



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

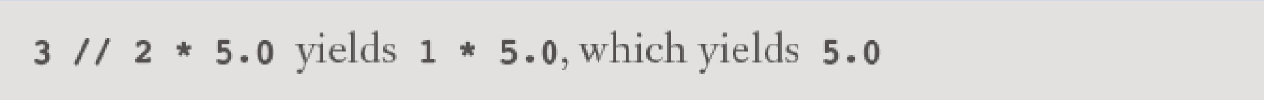
Mixed-Mode Arithmetic

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

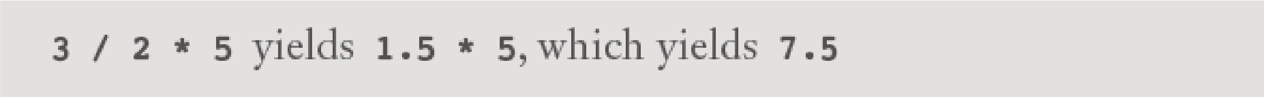


```
>>> 3.14 * 3 ** 2
28.26
```

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



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

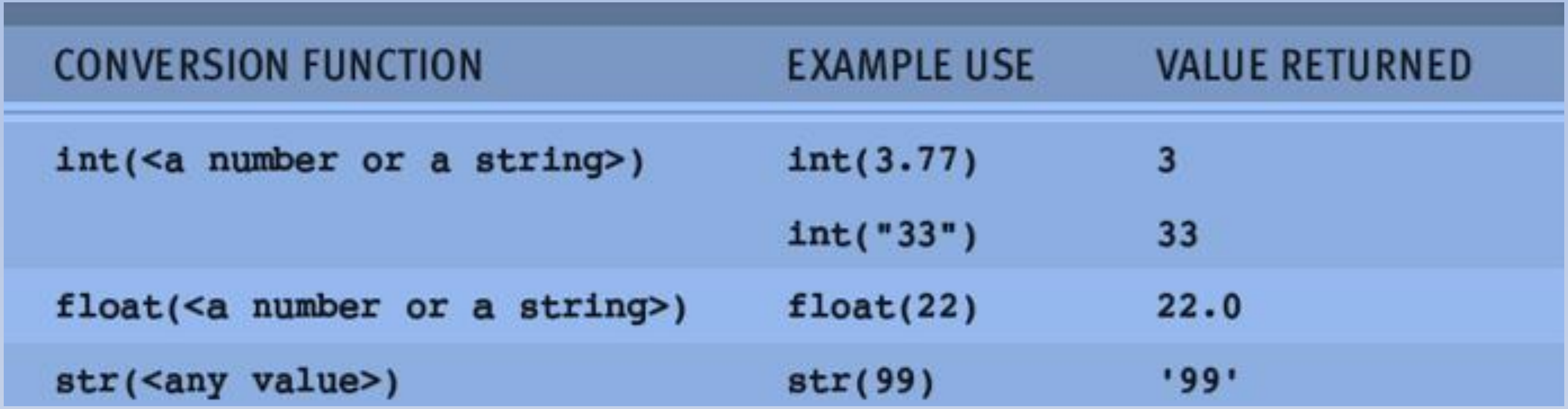


`3 / 2 * 5` yields `1.5 * 5`, which yields `7.5`

Tip:

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

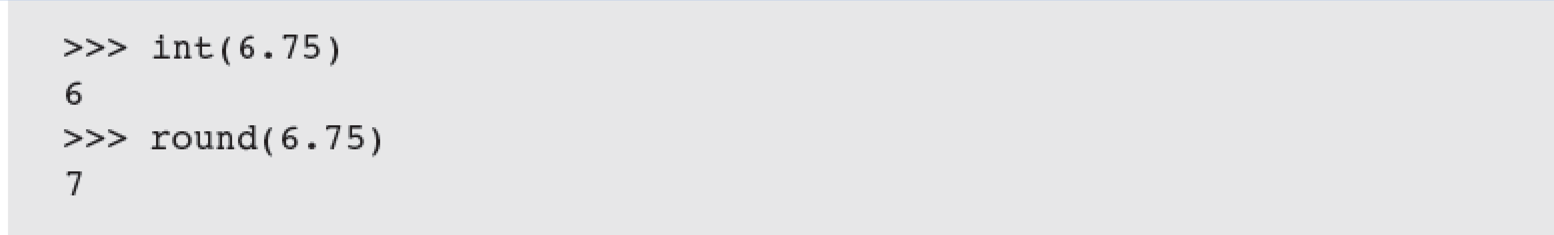
Mixed-Mode Arithmetic and Type Conversions



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

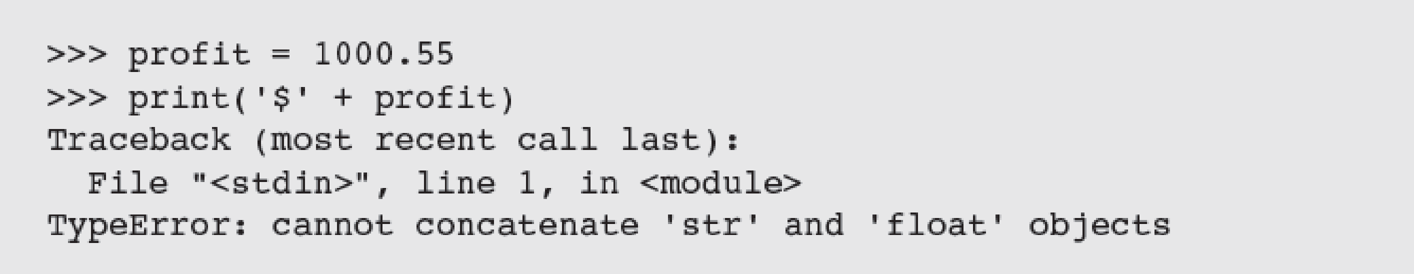
- Note that the **int** function converts a **float** to an **int** by truncation, not by rounding
- Use **round** function for rounding off



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

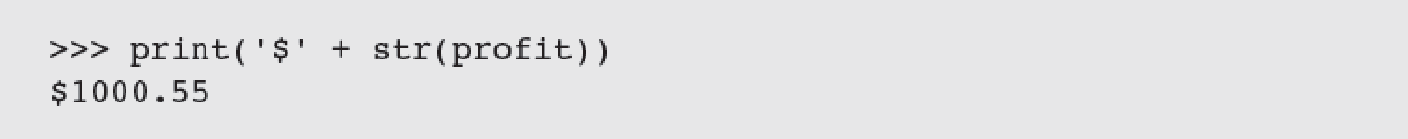
Mixed-Mode Arithmetic and Type Conversions

- 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
```

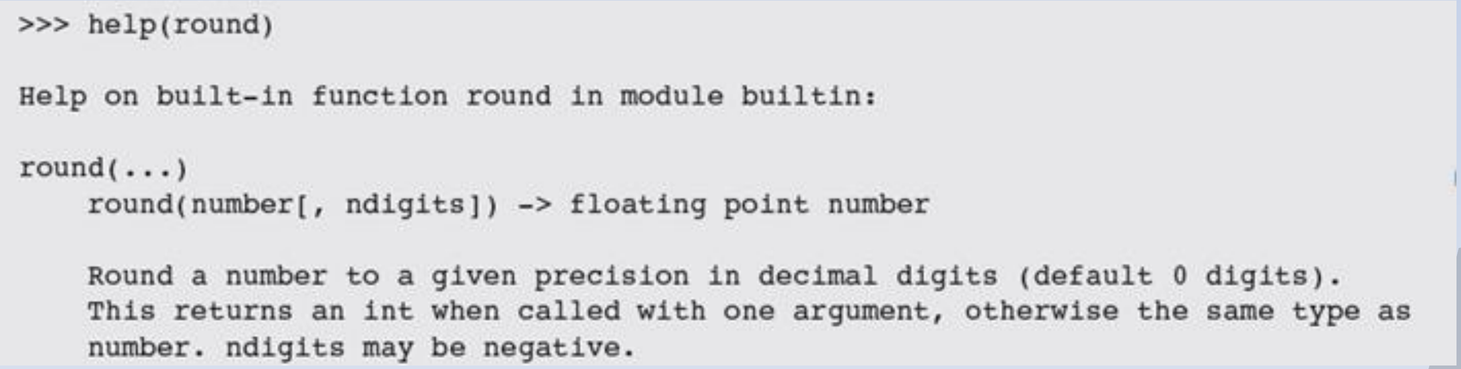
- Python is a strongly **typed** programming language

Using Functions and Modules

- Python includes many useful functions, which are organized in libraries of code called **modules**
- A **function** is chunk of code that can be called by name to perform a task

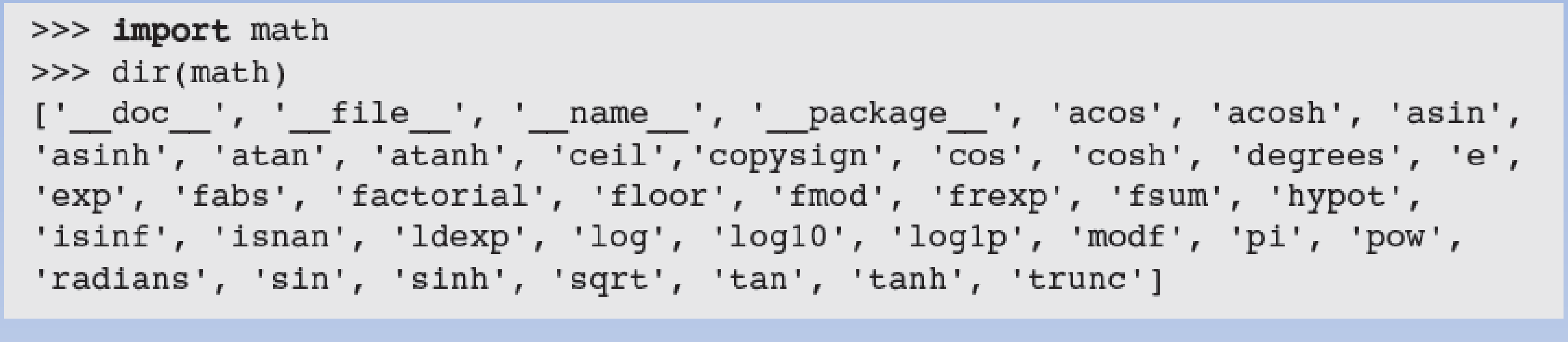
Calling Functions: Arguments and Return Values

- 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



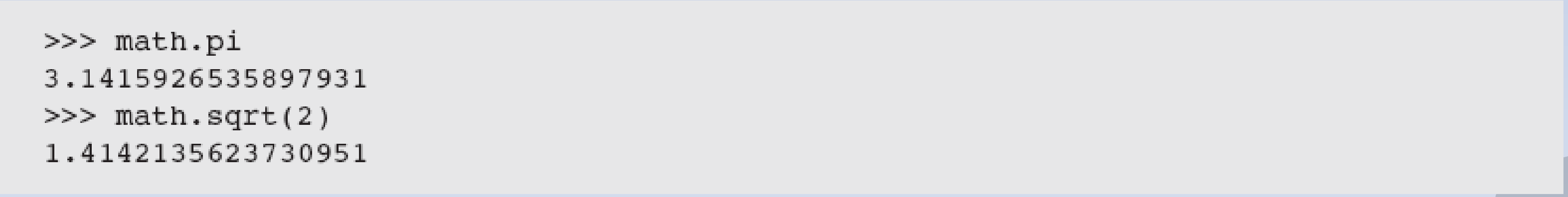
```
>>> help(round)

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']
```

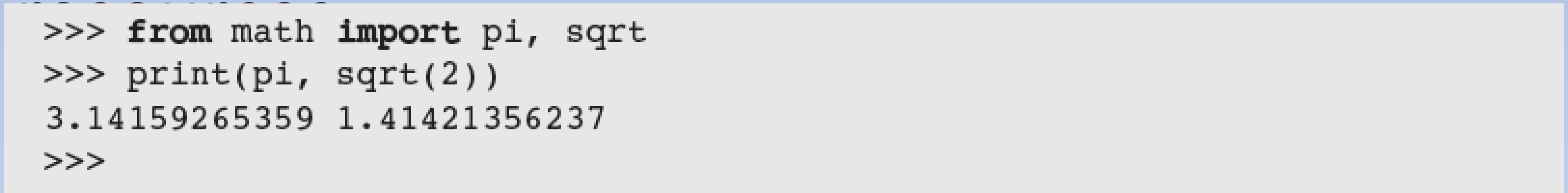
- To use a resource from a module, you write the name of a module as a qualifier, followed by a dot (.) and the name of the resource



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

The math Module

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



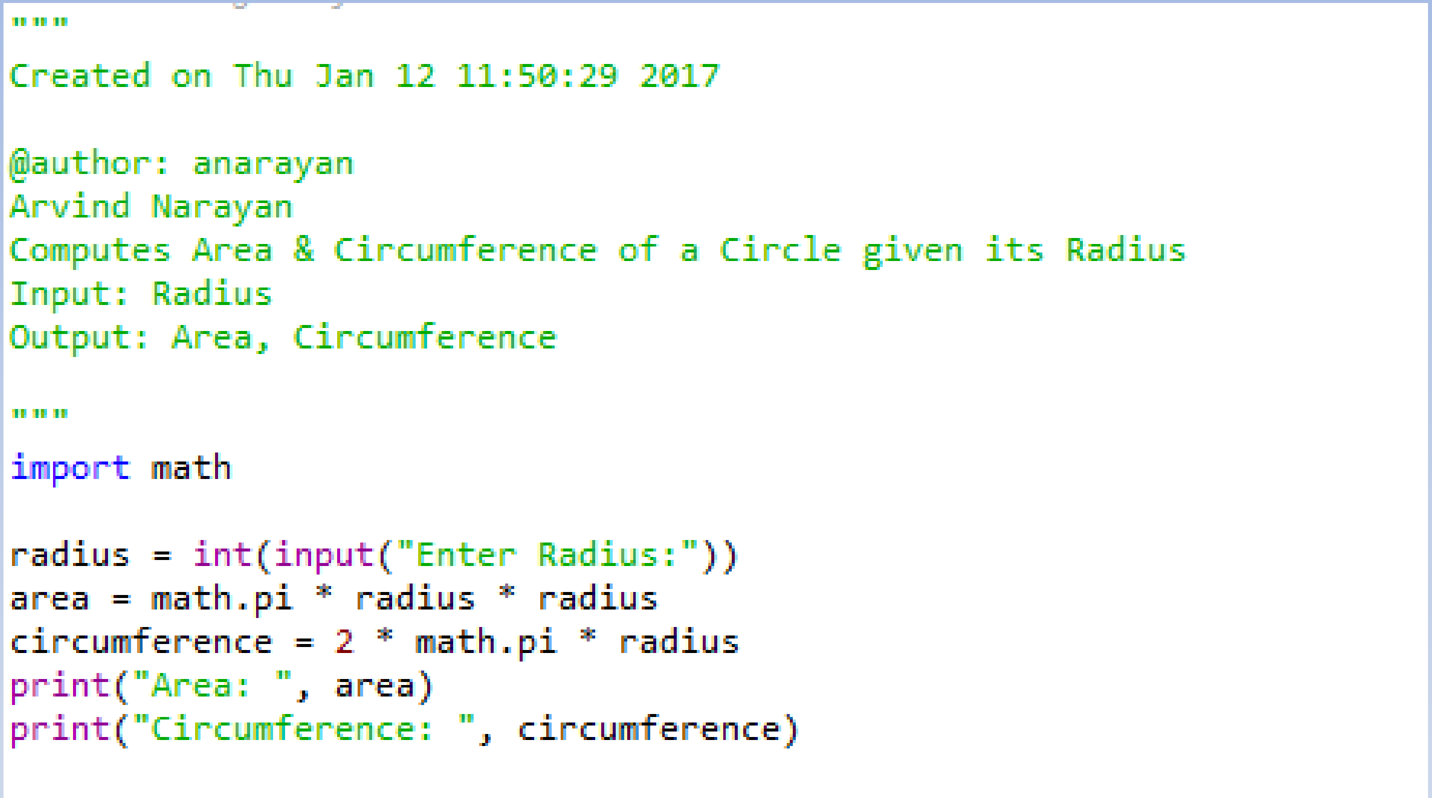
```
>>> from math import pi, sqrt
>>> print(pi, sqrt(2))
3.14159265359 1.41421356237
>>>
```

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

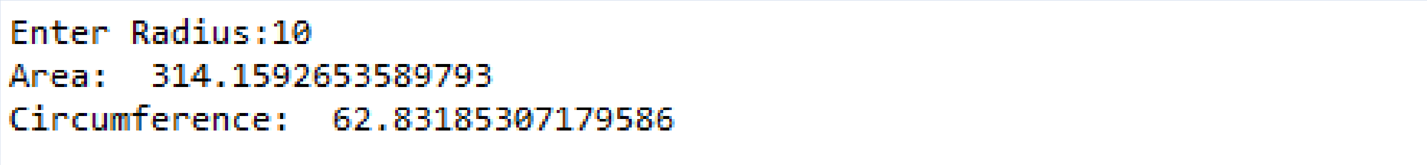
Program Format and Structure

- Start with comment with author’s name, purpose of program, and other relevant information
  - In a docstring
- Then, include statements that:
  - Import any modules needed by program
  - Initialize important variables, suitably commented
  - Prompt the user for input data and save the input data in variables
  - Process the inputs to produce the results
  - Display the results

Example Program

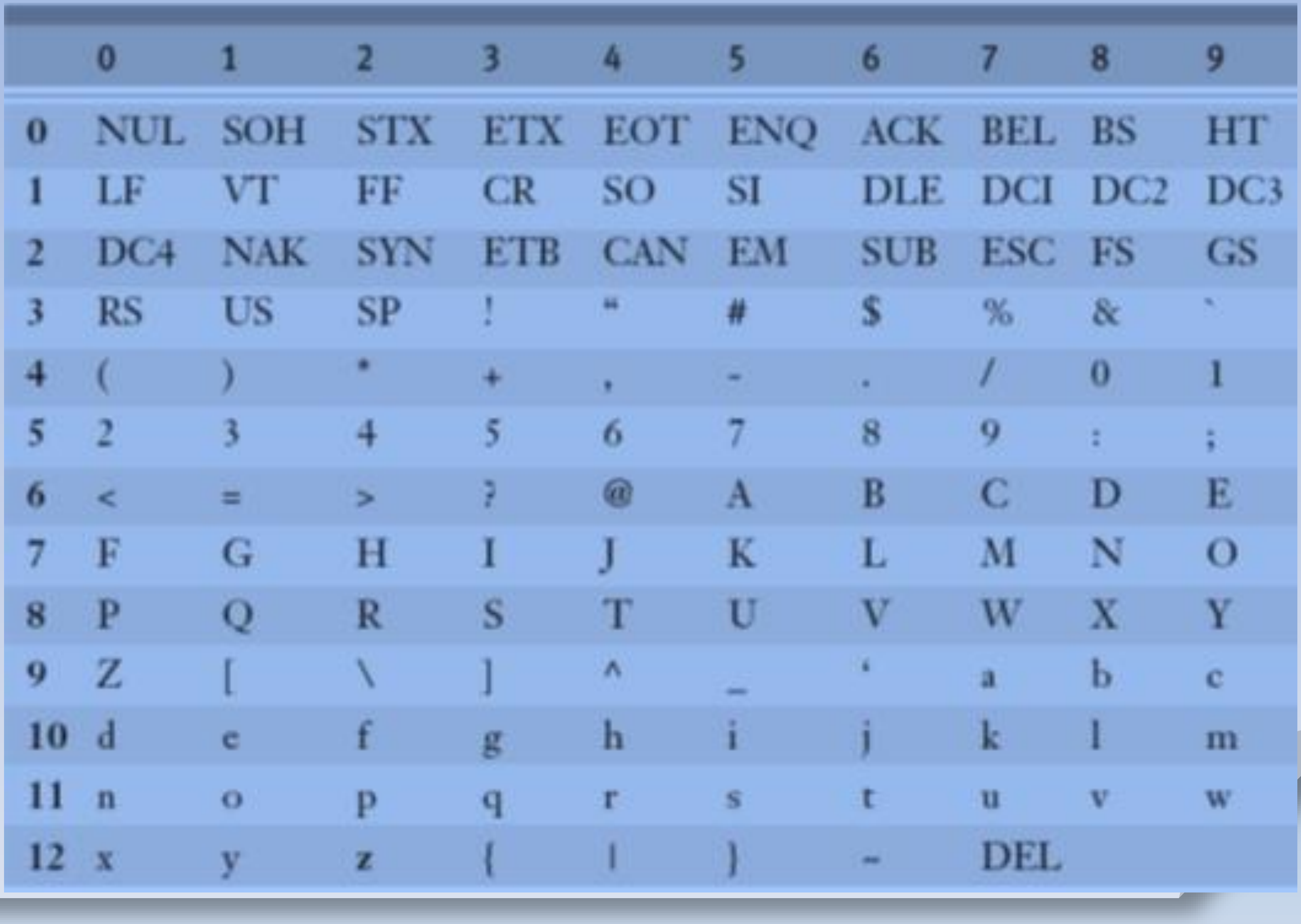


```
"""
Created on Thu Jan 12 11:50:29 2017

@author: anarayan
Arvind Narayan
Computes Area & Circumference of a Circle given its Radius
Input: Radius
Output: Area, Circumference

"""
import math

radius = int(input("Enter Radius:"))
area = math.pi * radius * radius
circumference = 2 * math.pi * radius
print("Area: ", area)
print("Circumference: ", circumference)
```



```
Enter Radius:10
Area:  314.1592653589793
Circumference:  62.83185307179586
```

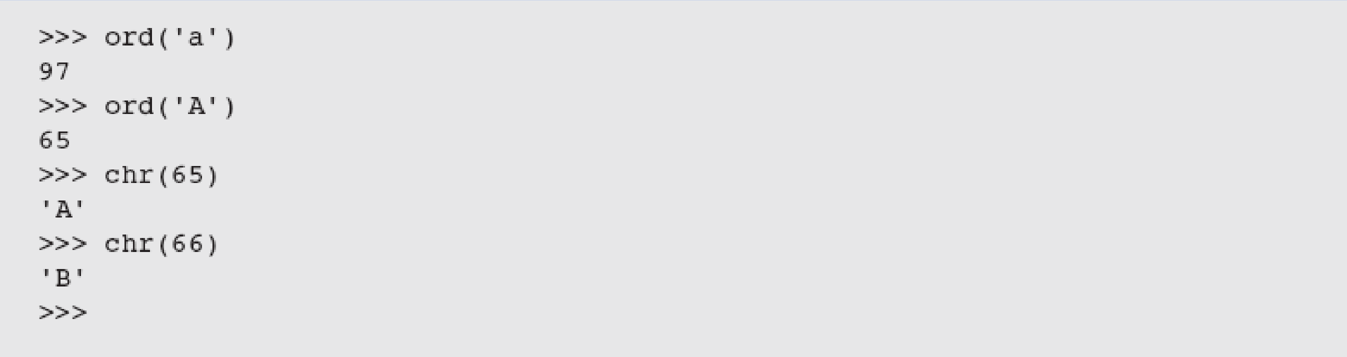
Character Sets



	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	DC1	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		

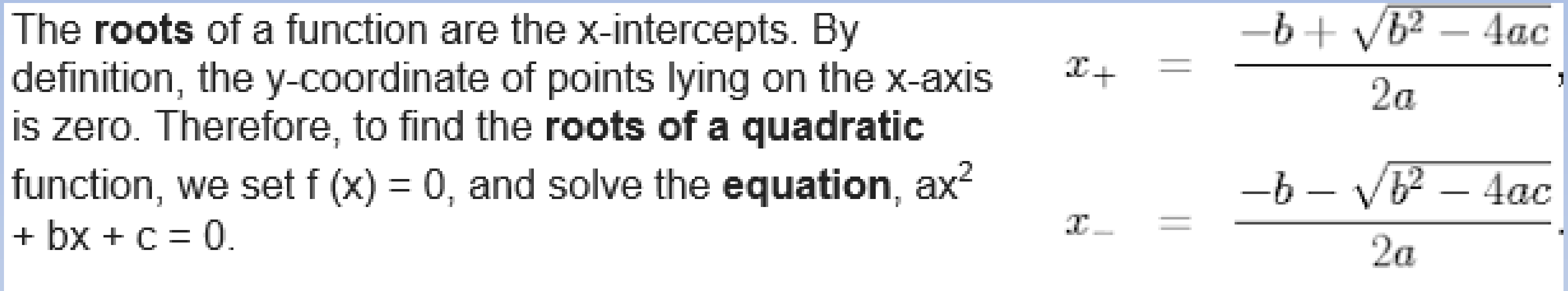
Character Sets

- 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** and the **Unicode 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'
>>>
```

Representing Mathematical Expressions/Equations



The **roots** of a function are the x-intercepts. By definition, the y-coordinate of points lying on the x-axis is zero. Therefore, to find the **roots of a quadratic** function, we set f (x) = 0, and solve the **equation**, ax^2 + bx + c = 0.

$$x_+ = \frac{-b+\sqrt{b^2-4ac}}{2a},$$

$$x_- = \frac{-b-\sqrt{b^2-4ac}}{2a}.$$

$$\frac{(x-2)^2}{9}+\frac{(y+2)^2}{{}^{1}\!/_{4}}$$