Some Topics on Math library, String and formatting

String Conversions

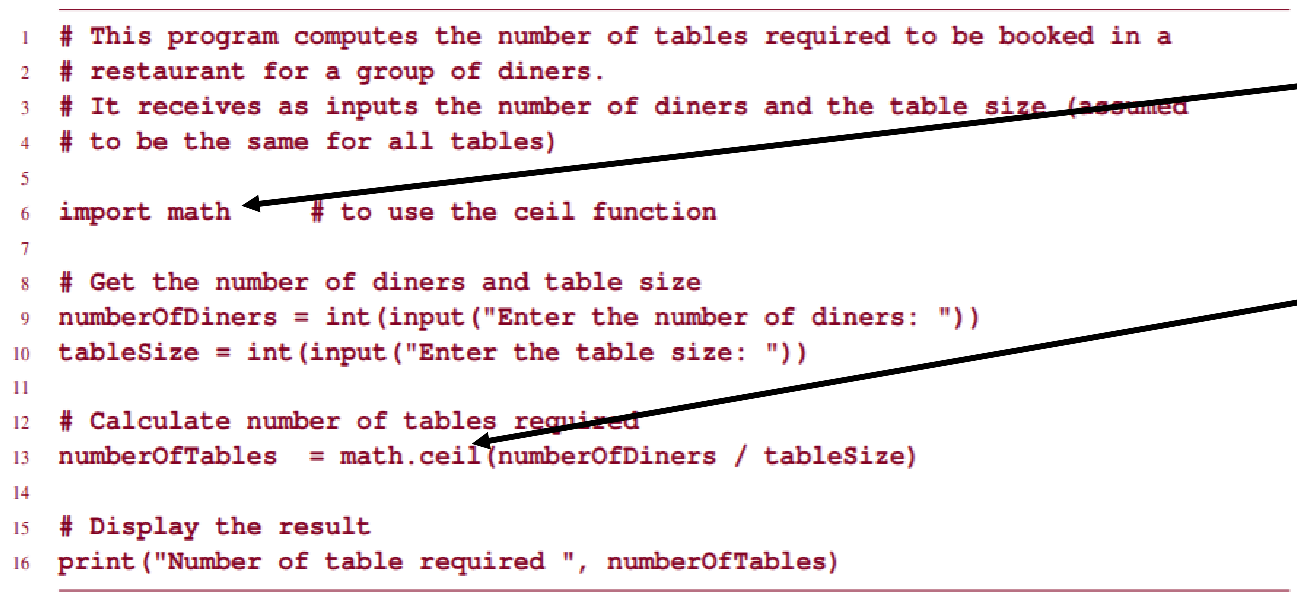
You can also use **int()** and **float()** to convert between strings and integers

You will get an **error** if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object
to str implicitly
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
with base 10: 'x'
```

Math Functions

- Example: Consider this program:

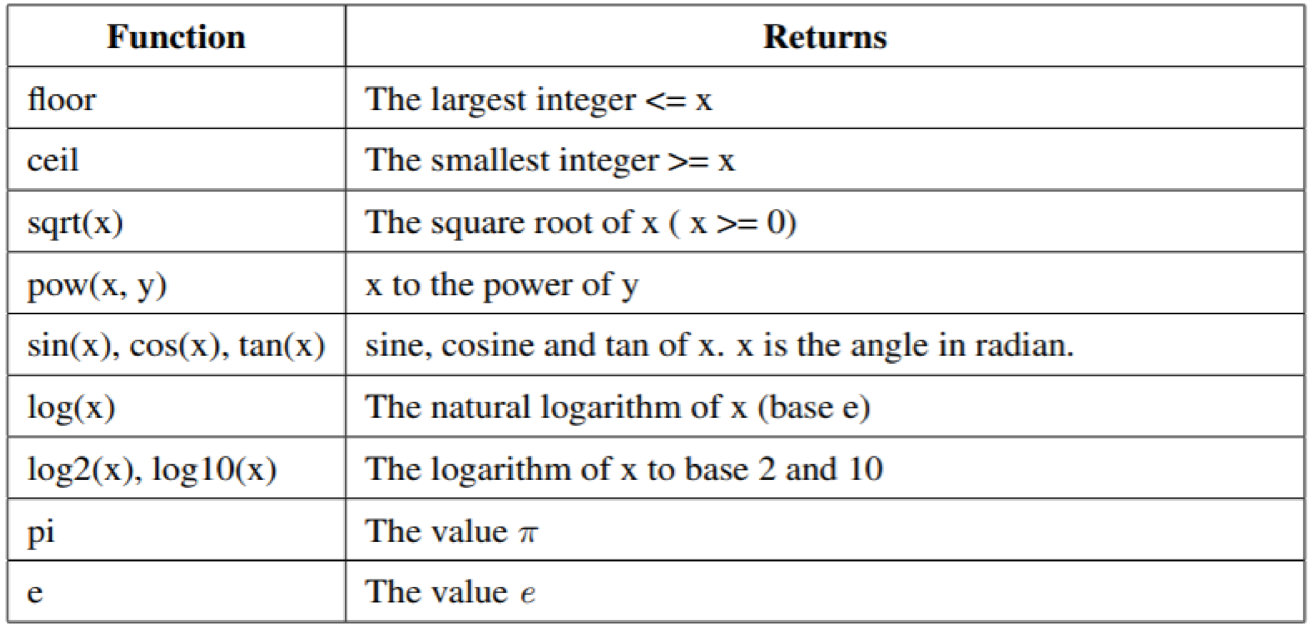


```
1   # This program computes the number of tables required to be booked in a
2   # restaurant for a group of diners.
3   # It receives as inputs the number of diners and the table size (assumed
4   # to be the same for all tables)
5
6   import math       # to use the ceil function
7
8   # Get the number of diners and table size
9   numberOfDiners = int(input("Enter the number of diners: "))
10  tableSize = int(input("Enter the table size: "))
11
12  # Calculate number of tables required
13  numberOfTables  = math.ceil(numberOfDiners / tableSize)
14
15  # Display the result
16  print("Number of table required ", numberOfTables)
```

On line 6, we import the **math** module so that we can subsequently access and use function **ceil**. On line 13, we perform a division, and then use the ceil function to get the smallest integer that is >= to the division result.

Some Commonly Used Math Functions

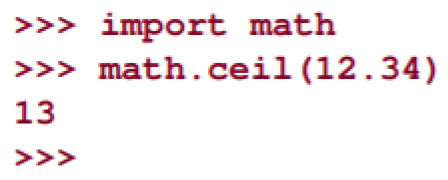
- The math module provides many useful mathematical functions and constants. Here is a small sample (we also list two constants in the first column):



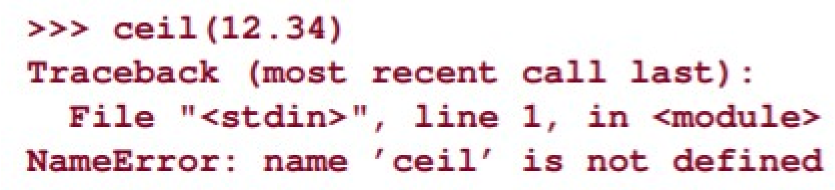
Function	Returns
floor	The largest integer <= x
ceil	The smallest integer >= x
sqrt(x)	The square root of x (x >= 0)
pow(x, y)	x to the power of y
sin(x), cos(x), tan(x)	sine, cosine and tan of x. x is the angle in radian.
log(x)	The natural logarithm of x (base e)
log2(x), log10(x)	The logarithm of x to base 2 and 10
pi	The value π
e	The value e

Importing Modules

- We can import the math module (or any other module) in several ways. Here are three commonly used ones:
- We can import the whole module (as shown below). In this case, we cannot refer to a function by its name alone. We must precede it with the name of the module



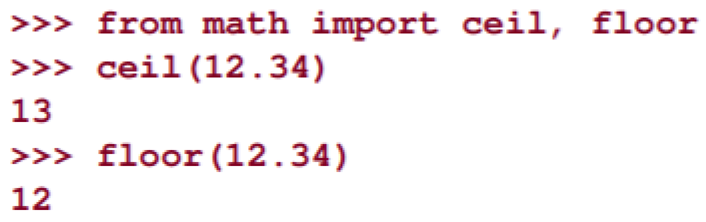
```
>>> import math
>>> math.ceil(12.34)
13
>>>
```



```
>>> ceil(12.34)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ceil' is not defined
```

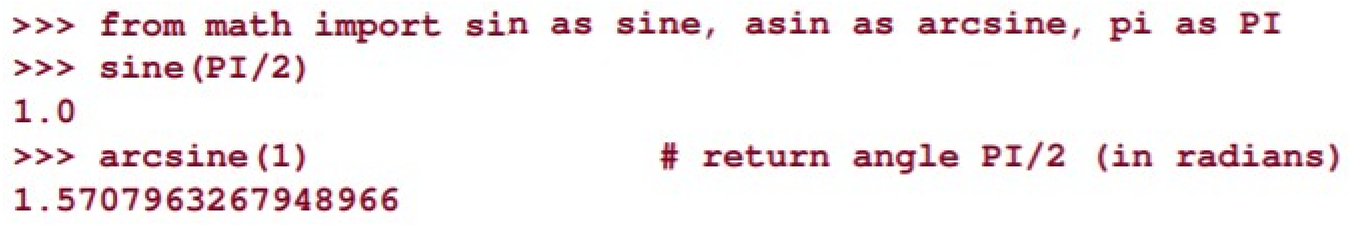
Importing Modules

- We can import specific elements of the module. In this case, we can refer to a function by its name alone.



```
>>> from math import ceil, floor
>>> ceil(12.34)
13
>>> floor(12.34)
12
```

- We can import specific elements and rename them:



```
>>> from math import sin as sine, asin as arcsine, pi as PI
>>> sine(PI/2)
1.0
>>> arcsine(1)                    # return angle PI/2 (in radians)
1.5707963267948966
```

Getting help on math functions (or elements of any module)

- First, we can import the **math** module, and list all it elements using the **dir** function:

```
>>> import math
>>> dir(math)
    # we get the  list of functions and constants defined in module math
```

- Then we can use the **help** function to get more information about a particular function, e.g.

```
>>> help(math.sin)
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
```

Dir(Math)

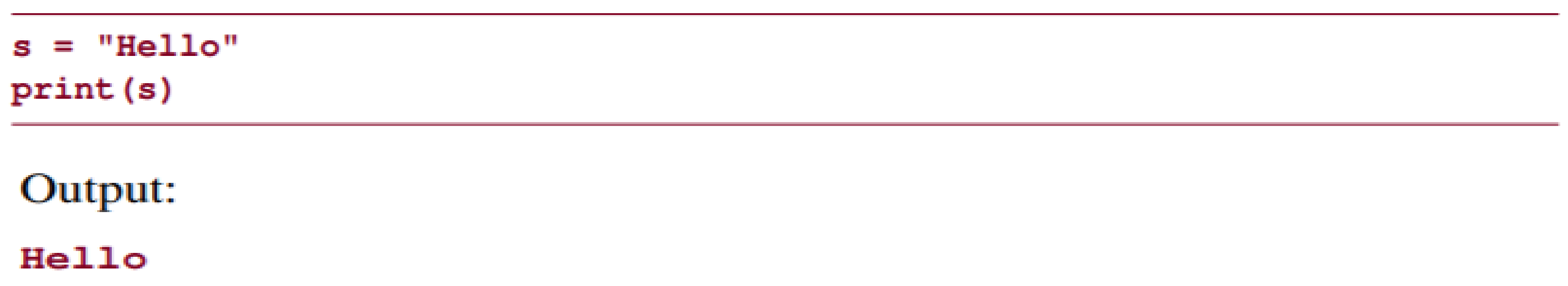
```
>>> dir(math)

[‘ doc', 'file', 'loader', 'name', 'package', 'spec', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign',
'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter',
'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

Help(math.sqrt)

```
>>> help(math.sqrt)

Help on built-in function sqrt in module math:

sqrt(x, /)

    Return the square root of x.
```

String Literals

- In Python, we can define string literals in **four ways**:
  - By enclosing a sequence of characters between ***a pair of double quotes***



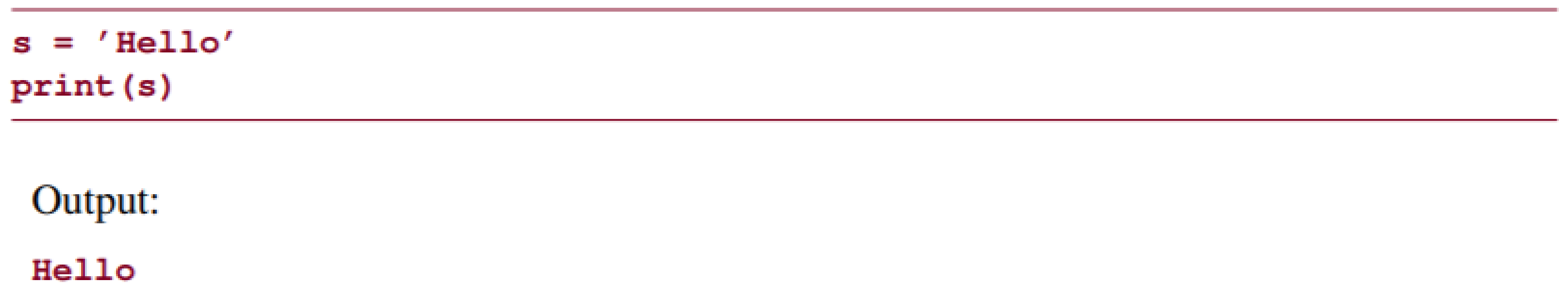
```
s = "Hello"
print(s)
```

Output:

```
Hello
```

String Literals

– By enclosing a sequence of characters between ***a pair of single quotes***.



```
s = 'Hello'
print(s)
```

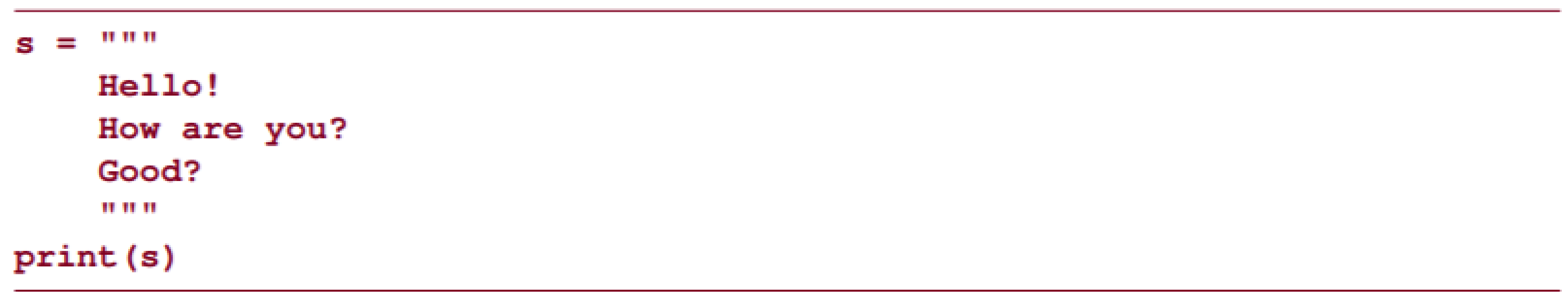
Output:

```
Hello
```

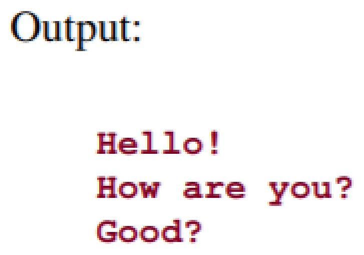
To the interpreter, it makes no difference whether we use a pair of double quote characters or a pair of single quote character. In other words, the strings defined in the previous two examples are exactly the same.

String Literals

- By enclosing several lines of text between ***a pair of triple double quotes***. This kind of string is known as multi-line string or triple-quoted string.



```
s = """
    Hello!
    How are you?
    Good?
    """
print(s)
```



Output:

```
    Hello!
    How are you?
    Good?
```

String Literals

- Similarly, we can enclose the line of text between ***a pair of triple single quotes***

```
s = '''
    Hello!
    How are you?
    Good?
    '''
print(s)
```

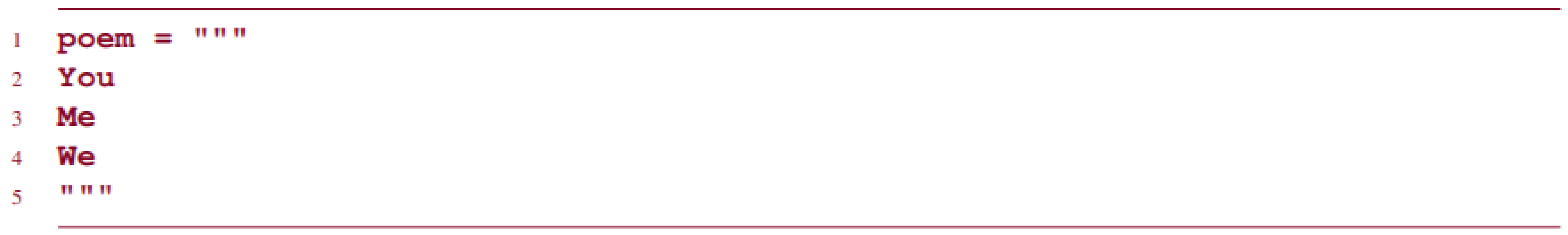
Output:

```
    Hello!
    How are you?
    Good?
```

- To the compiler, it makes no difference whether we use a pair of triple double quotes or a pair of triple single quotes. The strings defined in the last two example are exactly the same.

Example

- To gain a clearer understanding of multi-line strings, let us consider this one-statement program:



```
poem = """
You
Me
We
"""
```

Example

```
poem = """
You
Me
We
"""
```

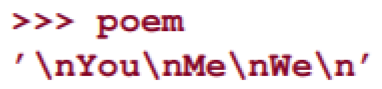
Now, let us run this program in **IDLE**, and in the subsequent **interactive shell**, let us enter first a statement to display the

```
>>> print(poem)

You
Me
We
```

The string poem has been displayed in what is known as the reading-friendly format.

Example

- Next, let us type **poem** in **interactive shell** to display the string again, and we will get



```
>>> poem
'\nYou\nMe\nWe\n'
```

- The string has been displayed in what is known as the unambiguous format.
- As we can see from our little experiment,
  - The multi-line string poem is a sequence of characters.
  - Some of the characters are occurrences of what is known as the **newline** character **\n**!. This character, when displayed by the print function, causes the “output cursor” to be advanced to the next line

Example

```
s = '''
   Hello!
   How are you?
   Fine. Thank you!
   '''

print(s)
```

Example

Output:

```
Hello!
    How are you?
    Fine. Thank you!
```

Example

Run the previous program in IDLE

Then we can find out more about the string:

```
>>> print(s)

    Hello!
    How are you?
    Fine. Thank you!
```

$\longleftarrow$ Reading-friendly format

```
>>> s
'\n   Hello!\n   How are you?\n   Fine.
Thank you!\n    ‘
```

$\longleftarrow$ unambiguous format

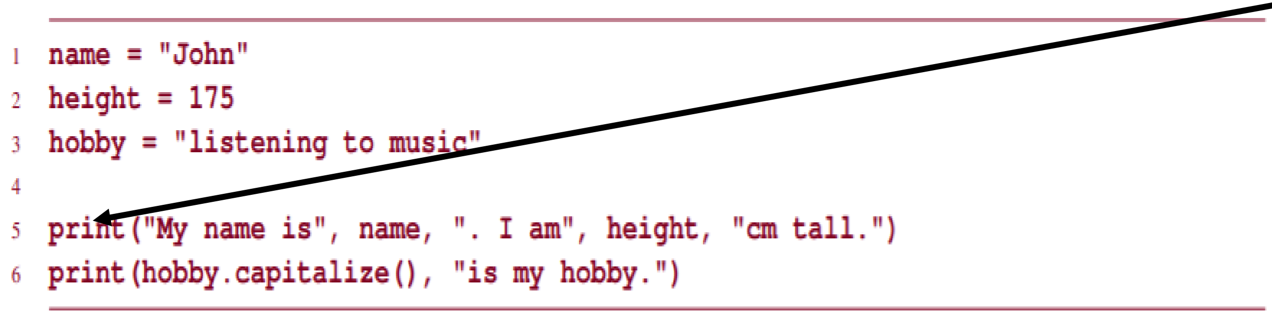
Keyboard Input

- For example, suppose we want to write a program that
  - asks the user to enter several numbers (0, 1 or more)
  - reads them, and
  - calculates and displays the average.

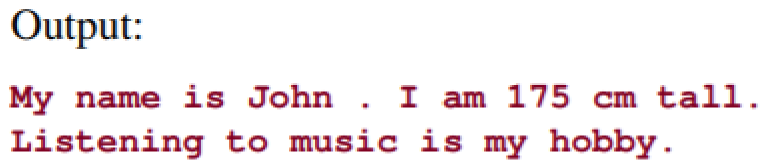
- Then, we will need to know about the **split** function, some basic ideas about lists, and how to use the for loop to process elements of a list.

Screen Output (Basic print Statement)

- Consider the program:



```
1  name = "John"
2  height = 175
3  hobby = "listening to music"
4
5  print("My name is", name, ". I am", height, "cm tall.")
6  print(hobby.capitalize(), "is my hobby.")
```



Output:

```
My name is John . I am 175 cm tall.
Listening to music is my hobby.
```

On line 5,

The **print** function takes 5 items.

It displays the items on the screen. All the items are displayed on one line, separated by a space character

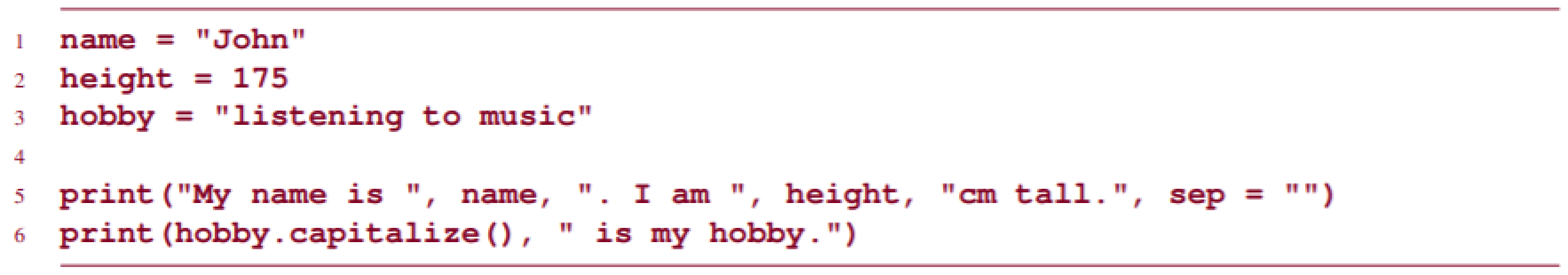
It then moves on to the next line. Consequently, the ***next print*** statement will display its output on this new line.

Specifying How To Separate the displayed items

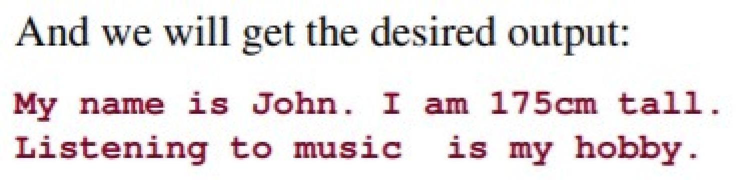
- We can use the so-called keyword parameter ***sep*** to specify how the items are to be separated. (We will cover keyword parameters later in the subject.)
- Consider the previous program again. By looking carefully at the output, we can see that there is a space between **John** and the period ., and a space between **175** and **cm**.

Specifying How To Separate the displayed items

- Now, suppose we want to get rid of those spaces. Then one way to do this is to specify that the separation string is an empty string, as shown on line 5 of the program below:



```
1  name = "John"
2  height = 175
3  hobby = "listening to music"
4
5  print("My name is ", name, ". I am ", height, "cm tall.", sep = "")
6  print(hobby.capitalize(), " is my hobby.")
```



And we will get the desired output:

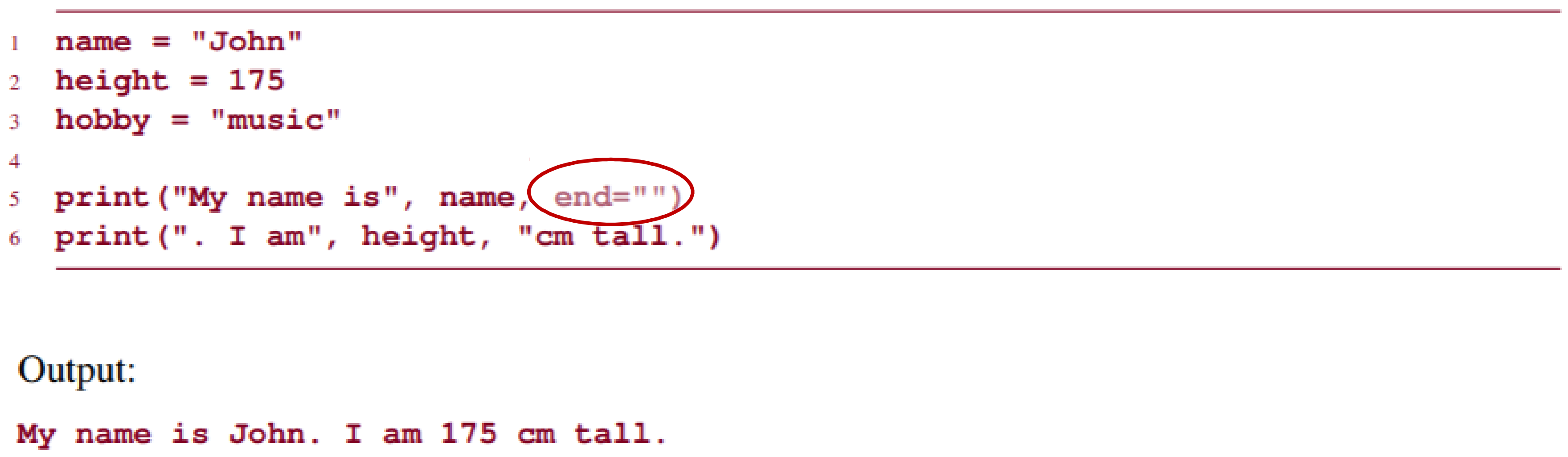
```
My name is John. I am 175cm tall.
Listening to music  is my hobby.
```

capitalize() function in Python

- In Python, the **capitalize()** method converts the first character of a string to capital **(uppercase)** letter. If the string has its first character as capital, then it returns the original string.
- Parameter: The capitalize() function does not takes any parameter.
- Return value: The capitalize() function returns a string with the first character in capital.

How to Print and Stay on the Same Line

- To print and then stay on the same line, we can use the keyword parameter **end** as shown line 5 below:



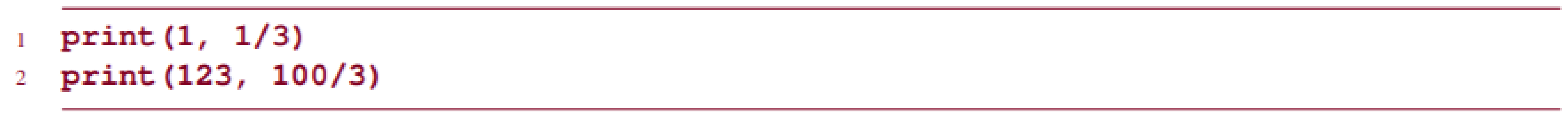
```
1  name = "John"
2  height = 175
3  hobby = "music"
4
5  print("My name is", name, end="")
6  print(". I am", height, "cm tall.")
```

Output:

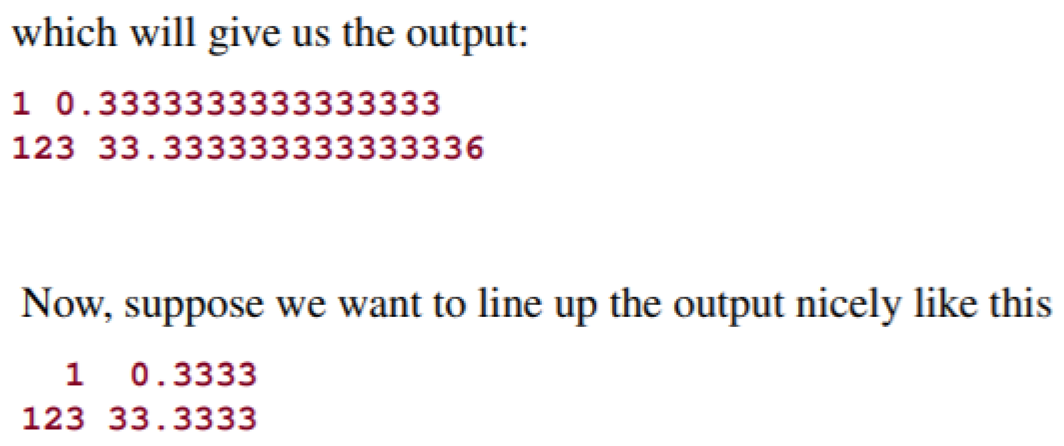
```
My name is John. I am 175 cm tall.
```

Formatting – Using format Methods to Control the Display

- Consider this program:



```
print(1, 1/3)
print(123, 100/3)
```



which will give us the output:

```
1 0.3333333333333333
123 33.333333333333336
```

Now, suppose we want to line up the output nicely like this

```
  1  0.3333
123 33.3333
```

Formatting – Using format Methods to Control the Display

- That is, we want to use the space of 3 characters to display the integers, and the space of 7 characters to display the floats, with 4 places reserved for the decimal digits.
- To do precisely that, we have two options.

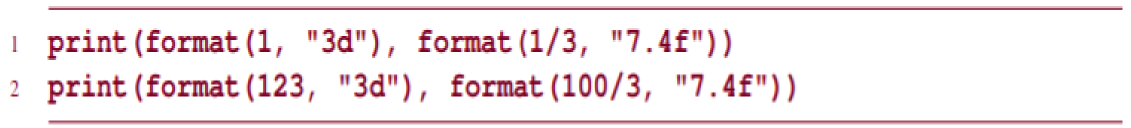
The built-in format function and the string’s format function

- To format an object (integer, float or string) , we can use the *built-in* function **format**, which has the general syntax (here, underlined are the parts that are to be substituted in a particular instance of use.)

format(object, format-specifier)

Formatting – Using format Methods to Control the Display

- **Option 1:**
  - We use the *built-in format function* as shown in the program below:



```
print(format(1, "3d"), format(1/3, "7.4f"))
print(format(123, "3d"), format(100/3, "7.4f"))
```

  - The **format** function takes two parameters: the first is the value to be displayed, and the second is a string, which specified how the value is to be displayed. This string is known as the *format specifier*.

In this program,

The format specifier "**3d**" specifies that the integer is to be displayed with the display width of 3 characters

The format specifier "**7.4f**" specifies that the float is to be displayed with the total display width of 7 characters, of which 4 places are reserved for decimal digits.

PrintOut

```
  1   0.3333
123  33.3333
```

Formatting – Using format Methods to Control the Display

- **Option 2:**
  - Alternatively, we can use the **string format function** (i.e. a function defined in the **str** class) as shown in the program below:

```
print("{:3d} {:7.4f}".format(1, 1/3))
print("{:3d} {:7.4f}".format(123, 100/3))
```

  - In this program, we put the specifier in a string, which is known as a *format string*.

PrintOut

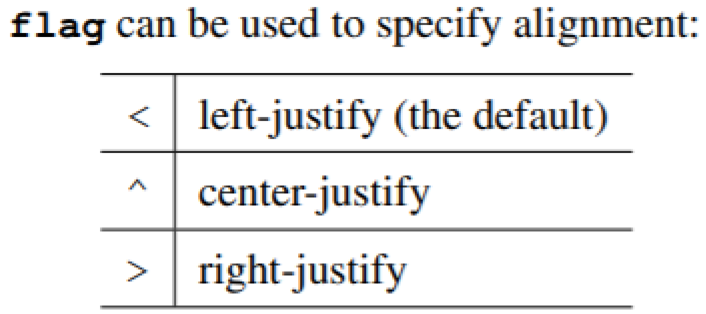
```
    1  0.3333
  123 33.3333
```

Format-specifier for strings

- Format specifier for strings has the general syntax (what inside pair of square brackets are **optional**):

[flag][minimum-width][.maximum-width] s

where

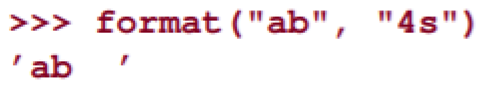


`flag` can be used to specify alignment:

<	left-justify (the default)
^	center-justify
>	right-justify

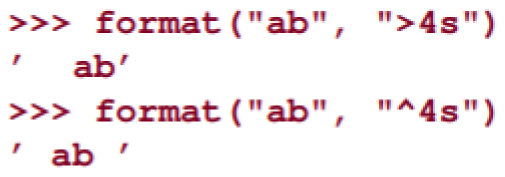
Format-specifier for strings

- Examples: Display the provided string with the minimum display width 4 characters (left-justified by default)



```
>>> format("ab", "4s")
'ab  '
```

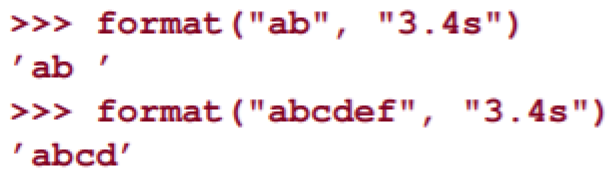
- Right-justify and center-justify the display



```
>>> format("ab", ">4s")
'  ab'
>>> format("ab", "^4s")
' ab '
```

Format-specifier for strings

- Display the provided string with the minimum width of 3 characters and the maximum width of 4 characters



```
>>> format("ab", "3.4s")
'ab '
>>> format("abcdef", "3.4s")
'abcd'
```

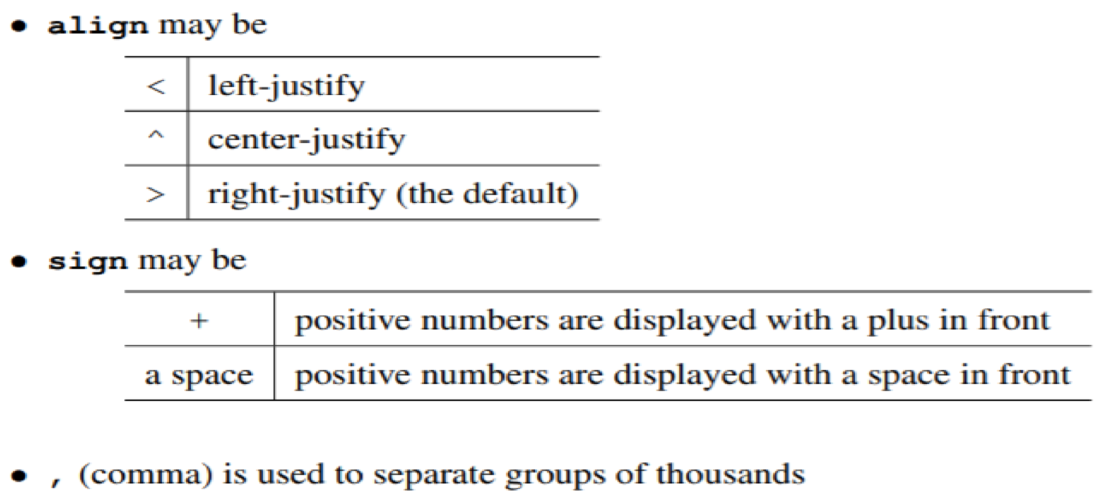
- We can see that if the string's length exceeds the specified maximum width, the resulting display will be truncated, starting from the right-most characters

Format-specifier for integers

- Format specifier for integers has the general syntax:

[align][sign][minimum-width][,] d

where



- `align` may be

<	left-justify
^	center-justify
>	right-justify (the default)

- `sign` may be

+	positive numbers are displayed with a plus in front
a space	positive numbers are displayed with a space in front

- `,` (comma) is used to separate groups of thousands

Format-specifier for integers

- Examples:

```
>>> format(12, "d")
'12'
>>> format(-12, "d")
'-12'
```

```
>>> format(12, "+d")
'+12'
>>> format(-12, "+d")
'-12'
```

```
>>> format(12, " d")     # a space before d
' 12'
>>> format(-12, " d")
'-12'
```

We can specify minimum display width:

```
>>> format(12, "3d")
' 12'
```

But there is no maximum limit. The number is never truncated:

```
>>> format(123456, "3d")
'123456'
```

We can separate groups of thousands:

```
>>> format(1234567890, ",d")
'1,234,567,890'
```

Format-specifier for floats

- Format specifier for floats has the general syntax:

  [<u>align</u>][<u>sign</u>][<u>minimum-width</u>][,][.<u>precision</u>] f

  where

  precision specifies the number of digits after the decimal points.

Format-specifier for floats

- Examples: we can specify the number of decimal points:

```
>>> format(12.3456, ".3f")
'12.346'

>>> format(12.3, ".3f")
'12.300'
```

- We can specify the minimum display width (which includes the decimal point). Similar to the case of integers, there is no maximum width (numbers are never truncated):

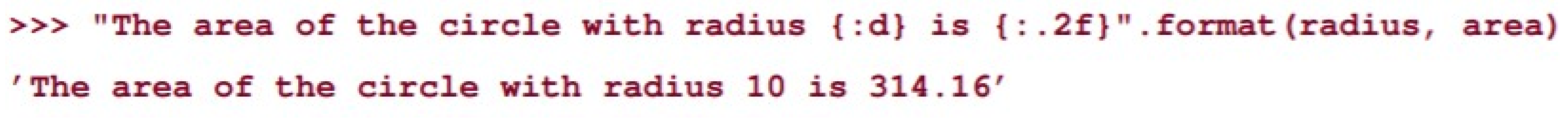
```
>>> format(12.3456, "8.2f")
'   12.35'

>>> format(12345678.1234, "8.2f")
'12345678.12'
```

As for integers, we can separate groups of thousands:

```
>>> format(12345678.1234, "8,.2f")
'12,345,678.12'
```

Formatting Multiple Objects

- To format multiple objects, we use the format method of the string class



```
>>> "The area of the circle with radius {:d} is {:.2f}".format(radius, area)
'The area of the circle with radius 10 is 314.16'
```

- The general syntax is <u>format-string</u>.format(<u>objects</u>)
- The format string has a number of replacement items (also known as “**replacement fields**”). Each replacement item is enclosed in a pair of curly brackets and has the general (slightly simplified) syntax

{:format-specifier}

where **format-specifier** is the same as what we have above.

Acknowledgement

- Acknowledgement to https://www.py4e.com/ and to Charles R. Severance (www.dr-chuck.com) of the University of Michigan School of Information.

- Also acknowledgement to Rick Skarbez (CSE1PES/CSE5CES).