



CSE4IP

Lecture 3 -Numbers, Expressions, and Statements

Ayad Turkey

Last time

- Python is a pretty good programming language
- We will be writing programs in Python
- We went over hello.py and some other examples in gory detail
- We went over variables, numbers and assignments

Constants

- **Fixed values** such as numbers, letters, and strings, are called “constants” because their value does not change
- Numeric **constants** are as you expect
- String **constants** use single quotes (') or double quotes (")

```
>>> print(123)
123
>>> print(98.6)
98.6
>>> print('Hello world')
Hello world
```

Sentences or Lines

x = **2**



Assignment statement

x = **x** + **2**



Assignment with expression

print(**x**)



Print statement

Variable

Operator

Constant

Function

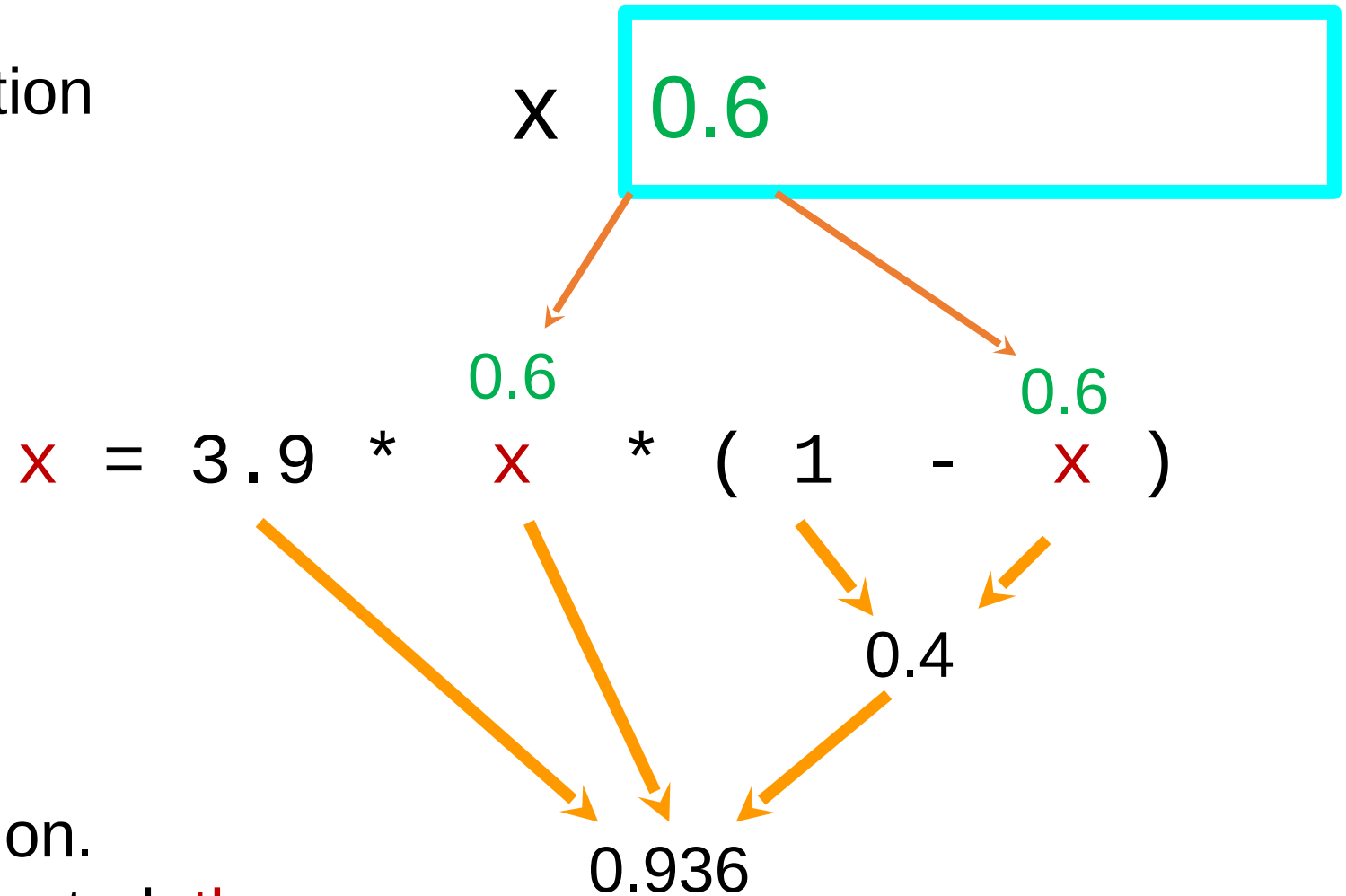
Assignment Statements

We assign a value to a variable using the assignment statement (=)

An assignment statement consists of an **expression on the right-hand side** and a **variable** to store the result

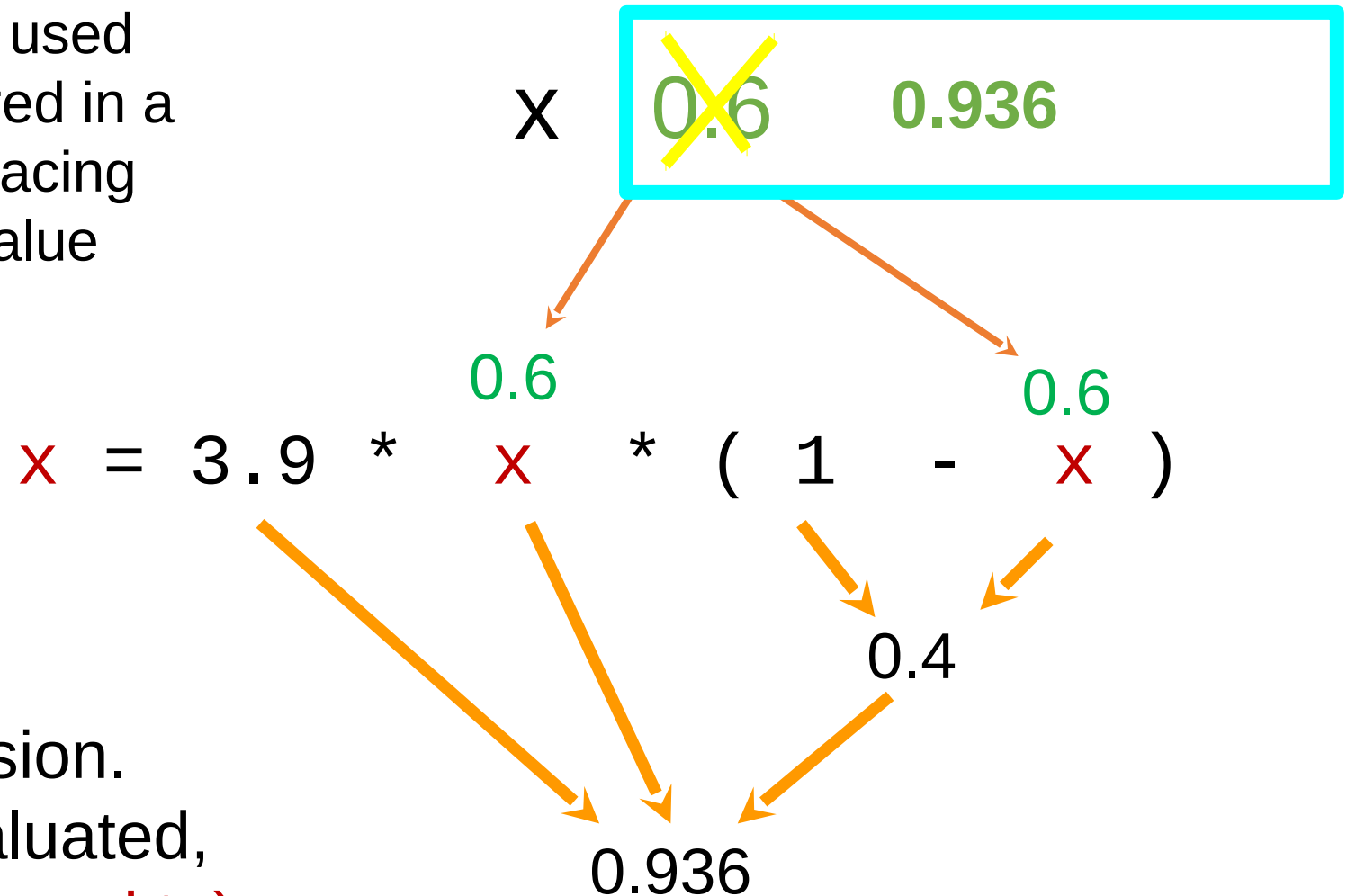
$$x = 3.9 * x * (1 - x)$$

A **variable** is a memory location used to store a value (0.6)



The right side is an expression.
Once the expression is evaluated, **the result is placed in (assigned to) x.**

A **variable** is a memory location used to store a value. The value stored in a variable can be updated by replacing the old value (0.6) with a new value (0.936).



The right side is an expression. Once the expression is evaluated, the result is placed in (assigned to) the variable on the left side (i.e., x).

Numeric Expressions

Because of the lack of mathematical symbols on computer keyboards - we use “computer-speak” to express the classic math operations

Asterisk is multiplication

Exponentiation (raise to a power) looks different than in math

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Numeric Expressions

```
>>> xx = 2
>>> xx = xx + 2
>>> print(xx)
4
```

```
>>> yy = 440 * 12
>>> print(yy)
5280
```

```
>>> zz = yy / 1000
>>> print(zz)
5.28
```

```
>>> jj = 23
>>> kk = jj % 5
>>> print(kk)
3
```

```
>>> print(4 ** 3)
64
```

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Order of Evaluation

When we string operators together - Python must know which one to do first

This is called “**operator precedence**”

Which operator “takes precedence” over the others?

$x = 1 + 2 * 3 - 4 / 5 ** 6$

Operator Precedence Rules

Highest precedence rule to lowest precedence rule:

Parentheses are always respected


Exponentiation (raise to a power)

Multiplication, Division, and Remainder

Addition and Subtraction

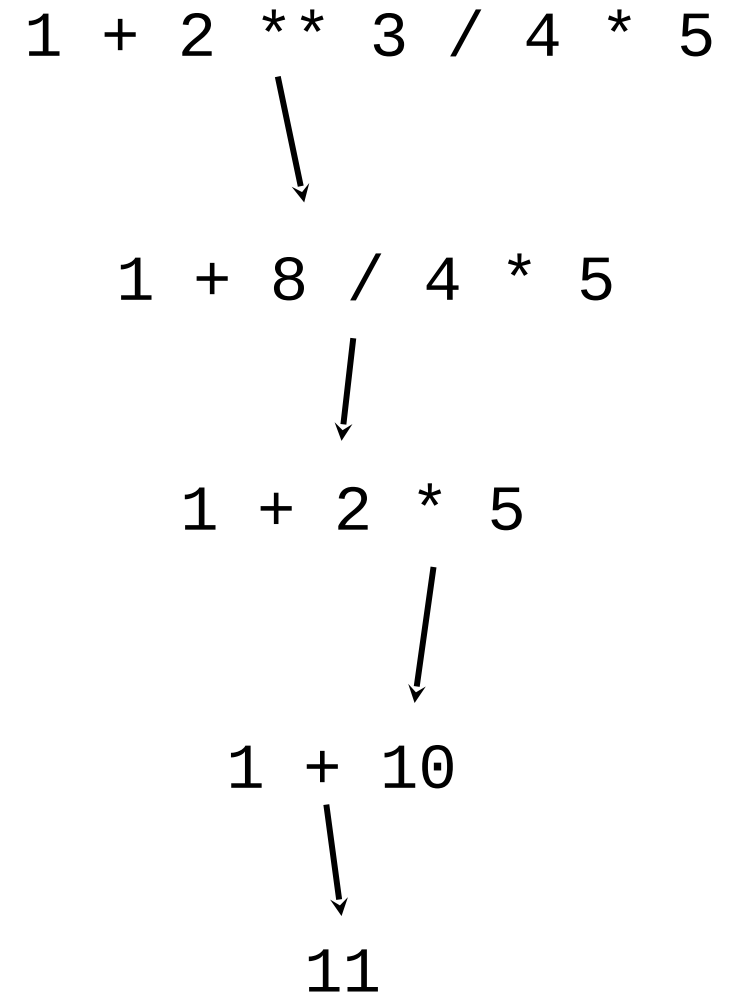

Left to right

Parenthesis
Power
Multiplication
Addition
Left to Right



```
>>> x = 1 + 2 ** 3 / 4 * 5
>>> print(x)
11.0
>>>
```

Parenthesis
Power
Multiplication
Addition
Left to Right



Operator Precedence


Remember the rules top to bottom

When writing code - use parentheses

When writing code - keep mathematical expressions simple enough that they are easy to understand

Break long series of mathematical operations up to make them more clear

Parenthesis
Power
Multiplication
Addition
Left to Right



What Does “Type” Mean?

In Python variables, literals, and constants have a “**type**”

Python knows the difference between an integer number and a string

For example “+” means “**addition**” if something is a number and “concatenate” if something is a string

```
>>> ddd = 1 + 4
>>> print(ddd)
5
```

```
>>> eee = 'hello ' + 'there'
>>> print(eee)
hello there
```

concatenate = put together

Type Matters

Python knows what “**type**”
everything is

Some operations are prohibited

You cannot “add 1” to a string

We can ask Python what type
something is by using the **type()**
function

```
>>> eee = 'hello ' + 'there'
>>> eee = eee + 1
Traceback (most recent call last):
File "<stdin>", line 1, in
<module>TypeError: Can't convert
'int' object to str implicitly
```

```
>>> type(eee)
<class 'str'>
>>> type('hello')
<class 'str'>
>>> type(1)
<class 'int'>
>>>
```

Several Types of Numbers

Numbers have two main types

- **Integers** are whole numbers:
-14, -2, 0, 1, 100, 401233
- **Floating Point Numbers** have decimal parts: -2.5 , 0.0, 98.6, 14.0

There are other number types - they are variations on float and integer

```
>>> xx = 1
>>> type (xx)
<class 'int'>
>>> temp = 98.6
>>> type(temp)
<class 'float'>
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>>
```


Type Conversions

When you put an integer and floating point in an expression, the integer is **implicitly** converted to a float

You can control this with the built-in functions `int()` and `float()`

```
>>> print(float(99) + 100)
199.0
>>> i = 42
>>> type(i)
<class 'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class 'float'>
>>>
```

Integer Division

Integer division produces a floating point result

```
>>> print(10 / 2)
```

```
5.0
```

```
>>> print(9 / 2)
```

```
4.5
```

```
>>> print(99 / 100)
```

```
0.99
```

```
>>> print(10.0 / 2.0)
```

```
5.0
```

```
>>> print(99.0 / 100.0)
```

```
0.99
```

This was different in Python 2.x

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
>>> nsval = 'hello bob'
>>> niv = int(nsval)
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 \geq 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 $\leq x$
ceil	The smallest integer $\geq x$
sqrt(x)	The square root of x ($x \geq 0$)
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 its 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).
```


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 examples are exactly the same.

Example

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

```
1 poem = """  
2 You  
3 Me  
4 We  
5 """
```

(The poem is often cited as being the shortest one in English, and its author is supposed to be Muhammad Ali.)

Example

```
1 poem = """  
2 You  
3 Me  
4 We  
5 """
```

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!
```

← Reading-friendly format

```
>>> s
```

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

← unambiguous format

Keyboard Input

- You have seen how to read a string, an integer or a float, one at a time, from the keyboard.
- Looking ahead: Later in this subject, you will learn how to read several items from one line. It will involve the use of the **split** function of the string (**str**) class, the **list** data type and the for loop.

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, as said before, 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.")
```

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.

Output:

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

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:

```
1 print(1, 1/3)
2 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.

Formatting - Using format Methods to Control the Display

● Option 1:

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

```
1 print(format(1, "3d"), format(1/3, "7.4f"))  
2 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.

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:

```
1 print("{:3d} {:7.4f}".format(1, 1/3))  
2 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*. More details about formatting is presented below.

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

- It is convenient to consider two cases:
 - Format an object (integer, float or string), and
 - Format multiple objects
- To format an object, 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)`

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:

[align][sign][minimum-width][,][.precision] 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 format-string.format(objects)
- 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.

Recap

- Python has a set of built-in arithmetic operators
- Fixed values such as numbers, letters, and strings, are called “constants” because their value does not change
- Assignment and initialization in Python are accomplished with the = operator
- Math module in Python provides access to the mathematical functions.
- Python user input from the keyboard can be read using the input() built-in function.

Acknowledgements

- Acknowledgement to <https://www.py4e.com/>

NEXT LECTURE: String and More



Thank
you.

Be
well.

latrobe.edu.au