CS303E: Elements of Computers and Programming

More Simple Python

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: February 5, 2025 at 14:43

Common Python Functions

A **function** is a group of statements that performs a specific task. You'll be writing your own functions soon, but Python also provides many functions for your use.

Function	Description
`abs(x)`	Return the absolute value
`max(x1, x2, ...)`	Return the largest arg
`min(x1, x2, ...)`	Return the smallest arg
`pow(a, b)`	Return a^b, same as `a ** b`
`round(x)`	Return integer nearest x; rounds to even
`round(x, b)`	Returns the float rounded to b places after the decimal

Common Functions

```
>>> abs( 2 )
2
>>> abs( -2 )            # absolute value
2
>>> max( 1, 4, -3, 6 )
6
>>> min( 1, 4, -3, 6 )
-3
>>> pow( 2, 10 )         # same as 2**10
1024
>>> pow( 2, -2 )         # same as 2**(-2)
0.25
>>> round( 5.4 )
5
>>> round( 5.5 )         # round to even
6
>>> round( 6.5 )         # round to even
6
>>> round( 5.466, 2 )    # set precision
5.47
```


Python Libraries

There are also many available functions that are not in the Python "core" language. These are available in **libraries**.

- os — interact with the operating system (e.g., change directory)
- math — access special math functions such as `log()`, `sin()`, `sqrt()`, `pi`
- random — random number generation
- datetime — clock and calendar functions

To use the functions/constants from a library you have to `import` it.

Import Examples

```
>>> import os                # import module os
>>> os.name                 # what's my OS?
'posix'
>>> os.getcwd()             # get current working directory
'/u/byoung/cs303e/slides'
>>> import random           # import module random
>>> random.random()         # generate random value
0.36552104405513963         # between 0 and 1
>>> random.random()         # do it again
0.7465680663361102
>>> import math             # import module math
>>> math.sqrt( 1000 )       # square root of 1000
31.622776601683793
>>> math.pi                 # approximation of pi
3.141592653589793
>>> import datetime         # import module datetime
>>> print( datetime.datetime.now() )
2024-08-26 10:17:20.895247
```


Some Functions in the `math` Library

`floor(x)`	returns the largest integer no bigger than x
`ceil(x)`	returns the smallest integer no less than x
`exp(x)`	exponential function e^x
`log(x)`	natural logarithm (log to the base e of x)
`log(x, b)`	log to the base b of x
`sqrt(x)`	square root of x

Trigonometric functions, including:

`sin(x)`	sine of x
`asin(x)`	arcsine (inverse sine) of x
`degrees(x)`	convert angle x from radians to degrees
`radians(x)`	convert angle x from degrees to radians

Functions from the `math` Library

```
>>> import math
>>> math.floor( 3.2 )
3
>>> math.ceil( 3.2 )
4
>>> math.exp( 2 )               # e ** 2
7.38905609893065
>>> math.log( 7.389 )           # log base e
1.9999924078065106
>>> math.log( 1024, 2 )         # log base 2
10.0
>>> math.sqrt( 1024 )
32.0
>>> math.sin( math.pi )
1.2246467991473532e-16
>>> math.sin( 90 )
0.8939966636005579
>>> math.degrees( math.pi )     # pi radians is 180 deg.
180.0
>>> math.radians( 180 )         # 180 deg. is pi radians
3.141592653589793
```


Example Using Math Functions

In file `ComputeAngles.py`:

```
"""Given the three vertices of a triangle, compute and
display the three sides and three angles."""

import math
# Our three vertices are (1, 1), (6.5, 1) and (6.5, 2.5)
x1 = 1
y1 = 1
x2 = 6.5
y2 = 1
x3 = 6.5
y3 = 2.5

# This computes the lengths of the three sides:
a = math.sqrt((x2 - x3) ** 2 + (y2 - y3) ** 2)
b = math.sqrt((x1 - x3) ** 2 + (y1 - y3) ** 2)
c = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

# This prints the three sides:
print("The three sides have lengths: ", round(a, 2), \
      round(b, 2), round(c, 2) )

# Continues on the next slide.
```

Example Using Math Functions

In file `ComputeAngles.py`:

```
# Continues from previous slide.

# This computes the three angles:
A = math.degrees(math.acos((a**2 - b**2 - c**2) / (-2*b*c)))
B = math.degrees(math.acos((b**2 - a**2 - c**2) / (-2*a*c)))
C = math.degrees(math.acos((c**2 - b**2 - a**2) / (-2*a*b)))

# This prints the three angles:
print("The three angles are ", round(A * 100) / 100.0, \
    round(B * 100) / 100.0, round(C * 100) / 100.0)
```

```
> python ComputeAngles.py
The three sides have lengths:  1.5 5.7 5.5
The three angles are  15.26 90.0 74.74
```

This example is from Listing 3.2 in the book, but without `eval` or `input`.

Random Numbers

Several useful functions are defined in the `random` module.

`randint(a,b)` : return a random integer between a and b, inclusively.

`randrange(a, b)` : return a random integer between a and b-1, inclusively.

`random()` : return a random float in the range $[0 \ldots 1)$.

Random Numbers: Examples

Note: You need to specify the module, even after you import it. *There's a way around that; we'll give you that later.*

```
>>> import random
>>> random.randint(0, 9)       # same as randrange(0, 10)
8
>>> random.randint(0, 9)
3
>>> random.randrange(0, 10)    # same as randint(0, 9)
2
>>> random.randrange(0, 10)
0
>>> random.randrange(0, 10)
3
>>> random.random()
0.689013943338249
>>> random.random()
0.546606113402984 3
```

It's often useful to generate random values to test your programs or to perform scientific experiments.

Random Floats: Scaling

Suppose you needed a random float between 0 and 100:

```
>>> random.random() * 100
63.90818900268016
>>> random.random() * 100
19.090419531785873
>>> random.random() * 100
4.8139113372750675
```

Or between 600 and 1000:

```
>>> random.random() * 400 + 600
921.05464024715
>>> random.random() * 400 + 600
824.1866143790331
>>> random.random() * 400 + 600
676.7450442494322
```

Aside on Import

There are several different ways to use `import`.

```
>>> import random              # imports module, not names
>>> random()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'module' object is not callable
>>> random.random()
0.46714522525882873
>>> from random import random  # import name random
>>> random()
0.9893720304830842
>>> randint(0, 9)              # but no others from module
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'randint' is not defined
>>> from random import *       # import all names in module
>>> randint(0, 9)
5
```


Let's Take a Break

Strings and Characters

A **string** is a sequence of characters. Python treats strings and characters in the same way.

```
letter = 'A'      # same as letter = "A"
numChar = '4'
msg = "Good morning"
```

Notice that you can use single quotes or double quotes, but they must match.

(Many) characters are represented in memory by binary strings, called the ASCII (American Standard Code for Information Interchange) encoding.

Triple Quotes

Quoted strings with three single or double quotes allow embedded linebreaks without the need to escape them:

```
"""This is a long quote that goes across
several lines. This is really handy for
documentation strings."""
```

You can use also such a string *as a comment* at the start of a module or function. Unlike regular comments, Python stores these as the function *docstring*. They can be retrieved with the `Functionname.__doc__` command, on both user-defined and system-defined functions.

```
>>> from math import sqrt
>>> print(sqrt.__doc__)
Return the square root of x.
>>>
```

Triple Quotes

You *can't* use triple-quoted strings anywhere you might want a comment. It's just like a pass statement; but you still have to watch indentation.

```
if True:
    print("abcd")
# Some comment                    # indentation
  doesn't matter here
else:
    print("efgh")

if True:
    print("abcd")
    """ Some comment """          # weird but
      harmless
else:
    print("efgh")
```


ASCII

The following is part of the ASCII (American Standard Code for Information Interchange) representation for characters.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	—	}		

The standard ASCII table defines 128 character codes (from 0 to 127), of which, the first 32 are control codes (non-printable), and the remaining 96 character codes are representable characters.

Strings and Characters

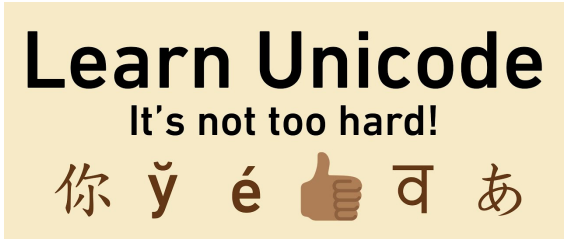
A string is represented in memory by a sequence of ASCII character codes. So manipulating characters really means manipulating these numbers in memory.

...	...	
...	...	
2000	01001010	Encoding for character 'J' (74)
2001	01100001	Encoding for character 'a' (97)
2002	01110110	Encoding for character 'v' (118)
2003	01100001	Encoding for character 'a' (97)
...	...	
...	...	

Note that a string is *probably* stored internally as a pointer to the first character and a length. *Most of the time, we don't care!*

Unicode

ASCII codes are only 7 bits (some are extended to 8 bits). 7 bits only allows 128 characters. *There are many more characters than that in the world.*



Unicode is an extension to ASCII that uses multiple bytes for character encodings. With Unicode you can have Chinese characters, Hebrew characters, emojis, etc.

Unicode was defined such that ASCII is a subset. So Unicode readers recognize ASCII.

Operating on Characters

Notice that:

- The lowercase letters have consecutive ASCII values (97...122); so do the uppercase letters (65...90).
- The uppercase letters have lower ASCII values than the uppercase letters, so "less" alphabetically.
- There is a difference of 32 between any lowercase letter and the corresponding uppercase letter.

To convert a letter from upper to lower, add 32 to the ASCII value.

To convert a letter from lower to upper, subtract 32 from the ASCII value.

To sort characters/strings, sort their ASCII representations.

ord and chr

Two useful functions for characters:

`ord(c)` : give the ASCII code for character `c`; returns a number.

`chr(n)` : give the character with ASCII code `n`; returns a character.

```
>>> ord('a')                              # ascii code for 'a'
97
>>> ord('A')                              # ascii code for 'A'
65
>>> diff = (ord('a') - ord('A'))
>>> diff
32
>>> upper = 'R'
>>> lower = chr( ord(upper) + diff ) # upper to lower
>>> lower
'r'
>>> lower = 'm'
>>> upper = chr( ord(lower) - diff ) # lower to upper
>>> upper
'M'
```


Escape Characters

Some special characters wouldn't be easy to include in strings, e.g., single or double quotes.

```
>>> print("He said: "Hello"")
  File "<stdin>", line 1
    print("He said: "Hello"")
                          ^
SyntaxError: invalid syntax
```

What went wrong?

To include these in a string, we need an *escape sequence*.

Escape Sequence	Name	Escape Sequence	Name
\b	backspace	\r	carriage return
\t	tab	\\	backslash
\n	newline	\'	single quote
\f	formfeed	\"	double quote

```
>>> print("He said: \"Hello\"")
He said: "Hello"
```


Printing without a Newline

Our previous examples using the `print` function always added a newline at the end.

```
>>> print("abc")
abc
>>>
```

You can print without the implicit newline using the `end` parameter (good for staying on the same line).

```
>>> print("abc", end = "")
abc>>> print("abc", end = 'xyz\n')
abcxyz
>>>
```

Note that `print(x)` is equivalent to `print(x, end='\n')`.

Printing with Different Separation

By default, `print` separates arguments with a space.

```
>>> print(1, 2, 3, 4)
1 2 3 4
>>> print("abc", "def")
abc def
```

You can override that behavior using the `sep` parameter.

```
>>> print(1, 2, 3, 4, sep="")
1234
>>> print("abc", "def", sep="xyz")
abcxyzdef
>>> print(8, 5, 2020, sep="/")
8/5/2020
```

`print( x, y )` is equivalent to `print( x, y, sep=" ")`.

Keyboard Input

The `input()` function is used to read data from the user during program execution.

General form:

`input` (<prompt string >)

When it's called:

- It prints the prompt string to the terminal. This is the message to tell the user to enter some input.
- It waits until the user types something and hits "Enter" or "Return."
- It reads in what the user typed *as a string*.

If you don't show the prompt string, it may not be obvious to your user that user input is expected.

Input Example

```
>>> input("Enter a number: ")
Enter a number: 32
'32'
>>> numEntered = input("Enter a number: ")
Enter a number: 32
>>> numEntered + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>> int(numEntered) + 1
33
```

Notice that the error happened because we tried to add a `str` to an integer.

Remember that keyboard input is always read as a `str`. You can interpret that string as an integer by using the `int` function.

Exercise

Try to rewrite our earlier program that computes the sides and angles of a triangle. Use `input` statements to accept the six vertex coordinates from the user.

Let's Take a Break

String Concatenation

The "+" operator can be used to concatenate two strings:

```
>>> msg1 = "My name is Bill "
>>> msg2 = "Young"
>>> msg1 + msg2
'My name is Bill Young'
>>> msg3 = "Good night"
>>> msg3 += " and good luck!"
>>> msg3
'Good night and good luck!'
```

Here "+" is actually shorthand for a method (function) associated with the `str` object *class*. More on classes later.

Some Uses of "+"

Note that "+" is *overloaded*, meaning that this same syntax is used for multiple purposes. The Python interpreter must figure out what you meant.

```
>>> 123 + 456                       # add two ints
579
>>> 12.3 + 45.6                     # add two floats
57.900000000000006                  # notice approximation
>>> "123" + "456"                   # "add" two strings
'123456'
>>> 123 + "456"                     # add int and str
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:'int' and 'str'
```

Python doesn't define "+" to have meaning in the last case, *but you could if you wanted to!*

Formatting

Note that `print()` automatically coerces each argument to a `str` value.

```
>>> print( 19.2, -20, "abc" )
19.2 -20 abc
```

Often you'd like to print numbers and strings nicely (fixed width, certain precision, right or left justified). Use the `format` function.

General Form:

```
format ( value, format-string )
```

This generates a `str` representation of `value` formatted as indicated by the `format-string`.

Formatting: Floats

To specify a format for a **float**, give a *format string* including:

1. field width: overall width of the resulting string
2. precision: number of digits after the decimal point

For example, the format string `"10.2f"`, means to create a string representing a float (the "f") right justified in a field of width 10, with 2 digits after the decimal point.

```
>>> format(123.456789, "10.2f")
'    123.46'
>>> format(123456.7891234, "8.3f")
'123456.789'
>>> format(123, "8.3f")
' 123.000'
```

Notice that if the field is not wide enough for the value, it is expanded. What would `format(123.456), "0.2f")` return?

Formatting: Floats

If you want to ensure that you print the right number of digits, `format` is your function.

This usually comes up if you need to display trailing zeros:

```
>>> x = 12.4005
>>> print( round(x, 2) )
12.4
>>> print( format(x, ".2f" ) )
12.40
>>>
```

`round` is about generating a rounded number; `format` is about generating a string representation to display:

Formatting: Floats

You can also format floats in scientific notation or as percentages.

```
>>> format(123.45678, "10.2e")
'  1.23e+02'
>>> format(123.45678, "10.2%")
' 12345.68%'
>>> format(0.000123, "10.2e")
'  1.23e-04'
>>> format(0.000123, "10.2%")
'     0.01%'
```

And you can *left justify* in the field:

```
>>> format(0.000123, "<10.2%")
'0.01%     '
>>> format(123.45678, "<10.2e")
'1.23e+02  '
```


Formatting: Ints

You can format an integer in decimal, hexadecimal, octal, or binary. You can also specify a *field width*.

```
>>> format(10000, "10d")          # base 10
'     10000'
>>> format(10000, "10x")          # base 16
'      2710'
>>> format(10000, "10o")          # base 8
'     23420'
>>> format(10000, "10b")          # base 2
'10011100010000'
```

Again, you can left justify:

```
>>> format(10000, "<10d")
'10000     '
```

Formatting: Strings

For strings, you can specify a width. Strings are left justified by default. You can right justify with ">".

```
>>> format("Hello, world!", "20s")
'Hello, world!        '
>>> format("Hello, world!", "5s")
'Hello, world!'
>>> format("Hello, world!", ">20s")
'        Hello, world!'
```

If the width is too short, the field is expanded; it's assumed that the data being displayed is more important than the format.

Putting It All Together

Let's design a program to input names, midterm and final exam grades for three students, compute the average for each and print the results out in a nice table.

In file `ExamExample.py`:

```
def main():

    """Input the names of three students with two exam
    grades for each, compute their test average, and
    print in table form."""

    # Enter info for Student1
    name1 = input("Enter the first student's name: ")
    midterm1 = int( input("Enter " + name1 + \
        "\'s midterm grade: ") )
    final1 = int( input("Enter " + name1 + \
        "\'s final exam grade: ") )

    < Do the same for the other two students. >
    # Continues on next slide.
```


Putting It All Together

```
    # Continues from previous slide.
    # Now, we print the table:

    # Print header:
    print( "\nName          MT    FN     Avg")
    print( "----------------------------")

    # Student1
    avg1 = (midterm1 + final1) / 2
    print( format(name1, "10s"), format(midterm1, "4d"), \
           format(final1, "4d"), format(avg1, "7.2f") )

    < Do the same for the other two students. >

main()
```


Running Our Program

```
> python ExamExample.py
Enter the first student's name: Charlie
Enter Charlie's midterm grade: 90
Enter Charlie's final exam grade: 75

Enter the second student's name: Susie
Enter Susie's midterm grade: 60
Enter Susie's final exam grade: 80

Enter the third student's name: Frank
Enter Frank's midterm grade: 8
Enter Frank's final exam grade: 77

Name          MT    FN     Avg
----------------------------
Charlie       90    75    82.50
Susie         60    80    70.00
Frank          8    77    42.50
```

Next stop: Selections.