CSC108: Introduction to Computer Programming

Lecture 4

Wael Aboulsaadat

Acknowledgment: these slides are based on material by: Velian Pandeliev, Diane Horton, Michael Samozi, Jennifer Campbell, and Paul Gries from CS UoT

Announcements

- Quiz average is 88%
- Solutions will be posted.
- Re-mark requests are due a week from today.
- Quiz 2 next Thursday

What have we learnt up till now?

- Variables
- Logical & Mathematical Operators
- Assignment Statement
- Types & Type conversion
- if/else Statement
- print
- input & raw_input
- Functions
- Docstrings
- while loops

Functions (revisited)

print vs return

- print and return do very different things:
- print is used to display information to the user by outputting it to the screen
- print can be used anywhere, as many times as is needed
- print gives information only to the user, it doesn't make it available to the programmer for future use

print vs return

- print and return do very different things:
- return is used to extract a value from a function for further use inside a program (in fact, it's the **only way to extract a value from a function**)
- return only appears at the end of a function body
- return passes information to other parts of the program, and does not make it available to the user

print vs return

- Write a function that computes the sum of all integers between 1 and a given number (inclusive).

Algorithm:

1) make a variable to keep track of the sum
2) starting at 1, add the integer to the sum and increment it by 1
3) repeat Step 2 until you have added the given number

print vs return

```
def sum_range(num):
    sum = 0
    curr_number = 1
    while curr_number <= num:
        sum = sum + curr_number
        curr_number = curr_number  + 1
```

- Now what? Do we print? Do we return? The instructions didn't specify.
- When in doubt, use the more general statement, the statement that lets the programmer decide what to do next: use return.

print vs return

```
def sum_range(num):
        sum = 0
        curr_number = 1
        while curr_number <= num:
                sum = sum + curr_number
                curr_number = curr_number  + 1
         return sum
```

- This makes sense also because the function itself shouldn't know or care where num came from, or what the intended use of the sum is.

Variables (revisited)

Variable Scope

- What does this program do?

```
def f():
        t = 5
        print t
x = 9
print x
f()
print t
```

Variable Scope

- What about this one?

```
def f():
        t = 5
        print x
x = 9
print x
f()
print t
```

Namespaces

- In Python, the structure that keeps track of the names Python knows is called a **namespace.**
- Namespaces contain names associated with variables, functions, imported modules, etc.
- Python programs have multiple namespaces, meaning that they store names in several different places.
- This matters because the method Python uses to look for names can affect the **scope of your** variables: which parts of your code 'know' about them

Namespaces

- At the lowest level of every Python program there's a **built-in namespace, which automatically contains** the names of all available built-in functions.

- When the program starts, a **global namespace is** created to keep track of global variables.

- Finally, a new **local namespace is created every time** a function body is executed. It contains only variables local to that function (such as parameters).

Namespaces

- Local namespaces are destroyed when the function body exits.

- Since function bodies can contain other function definitions, namespaces can contain other nested local namespaces.

Namespaces

```
def f():
    t = 5
    def g():
        s = 3
        t = 4
    g()
    m = 10
x = 9
f()
y = x + 2
```

Namespaces

```
→ def f():
      t = 5
      def g():
          s = 3
          t = 4
      g()
      m = 10
  x = 9
  f()
  y = x + 2
```

Global namespace:
f (function)

Built-in namespace:
abs (function)
....

Namespaces

```
def f():
    t = 5
    def g():
        s = 3
        t = 4
    g()
    m = 10
→ x = 9
f()
y = x + 2
```

Global namespace:
f (function)
x = 9

Built-in namespace:
abs (function)
....

Namespaces

```
def f():
    t = 5
    def g():
        s = 3
        t = 4
    g()
    m = 10
x = 9
f()
y = x + 2
```

f() namespace:
t = 5

Global namespace:
f (function)
x = 9

Built-in namespace:
abs (function)
....

Namespaces

```
def f():
    t = 5
    def g():
        s = 3
        t = 4
    g()
    m = 10
x = 9
f()
y = x + 2
```

f() namespace:
t = 5
g (function)

Global namespace:
f (function)
x = 9

Built-in namespace:
abs (function)
....

Namespaces

```
def f():
    t = 5
    def g():
        s = 3
        t = 4
    g()
    m = 10
x = 9
f()
y = x + 2
```

g() namespace:
s = 3

f() namespace:
t = 5
g (function)

Global namespace:
x = 9
f (function)

Built-in namespace:
abs (function)
....

Namespaces

```
def f():
    t = 5
    def g():
        s = 3
        t = 4
    g()
    m = 10
x = 9
f()
y = x + 2
```

g() namespace:
s = 3
t = 4

f() namespace:
t = 5
g (function)

Global namespace:
x = 9
f (function)

Built-in namespace:
abs (function)
….

Namespaces

```
def f():
    t = 5
    def g():
        s = 3
        t = 4
    g()
→   m = 10
x = 9
f()
y = x + 2
```

Namespace erased

f() namespace:
t = 5
g (function) m = 10

Global namespace:
x = 9
f (function)

Built-in namespace:
abs (function)
....

Namespaces

```
def f():
    t = 5
    def g():
        s = 3
        t = 4
    g()
    m = 10
x = 9
f()
→ y = x + 2
```

Namespace erased

Namespace erased

Global namespace:
x = 9
f (function) y = 11

Built-in namespace:
abs (function)
....

Consquences

1) If you want to hold on to a local variable's value, you have to make the function return it

2) A local namespace cannot change the values of more global variables

3) A variable in a local namespace will 'hide' variables of the same name in more global namespaces

4) Namespaces can get a little tricky:

Tricky namespaces

```
def tricky():
    print x
    x = 5
x = 4
tricky()
```

- **Use different variable names.**

Strings (revisited)

Strings

- A sequence of characters in Python is called a **string.**
- A string is how Python represents text:

'Hello World'

"Dear auntie"

"123 is 321"

String formatting

- We can write better than

```
print "The sum of", w, ","x,",",y,"and",z,"is",sum,". "
```

  Python has a way to specify where in a string you'd like a value to appear, and in what format.

- If x is an integer, instead of using:

```
print "You have", x, "dollars"
```

  We can use:

```
print "You have %d dollars " % x
```

- %d means "Insert the value of the variable I give you here, and format it as an integer"

Format Placeholder

- %d displays the value as an integer
- %f displays the value as a floating-point decimal
- %f.2 displays the value as a floating-point decimal accurate (and padded) to 2 decimal places
- %s displays the value as a string of characters

Multiple Variables

- To specify multiple variables with placeholders, you have to separate them with commas and enclose them in parentheses after the %:

```
dollars = 4
cents   = 35
print "You have %d dollars and %d cents" %
                        (dollars, cents)
```

New Line

- In Python, you can insert a new line in the middle of a string by using \n:

```
print "Sincerely,\nB. Pitt "
```

- You can break up a line that's too long (over 80 characters) into multiple lines with \:

```
print "When I was a little girl,\
Barbara Stanwick and I used to dance "
```

- For expressions:

```
return (number_of_generals *
    number_of_soldiers_per_general)
```

String Comparison

- Comparison operators apply to strings.

- In the case of strings, a 'greater' string is one which is further down the list in **alphabetical order** **than** a 'lesser' string.

```
>>> 'Alice' < 'Zimbabwe'
True
>>> 'Timmy' > 'Tommy'
False
>>> 'Timmy' < 'timmy'
True
```

String Comparison

- ASCII Table & codes

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Special Strings

- There are several special characters that can be represented inside strings:

Character	Representation
new line	`\n`
tab	`\t`
backslash	`\\`
quotation	`\"`

Strings Can Be Sliced

- What if we wanted a string representing all the characters of 'Hello!' except the first?

- We can **slice (or ask for substrings of) a string** using the following notation:

  string[start:end]

  - start and end are both **indices within the string** (which could be negative).

- The character at position start is included, but the character at position end is not! start and end are both optional.

for loop (revisited)

For loop

- Unlike the while loop which checks the status of a condition before it runs, the for loop will execute once for each element in the collection.

```
for elmt in list_of_items:
    statement1
    statement2
    …
```

- At the beginning of every cycle, the next element in list_of_items is assigned as the value of variable elmt. Then, statements are executed in order.

For loop

- Let's try the for loop on a collection of characters (a string):

```
for x in 'Hello World!':
    print x
```

- This means:

  "Take every element in collection 'Hello World!' in turn, assign it to variable x, and print it to the screen.

range() and for loops

- Python has a built-in function called range() which generates lists of integers.

- If you call range(a,b) with two arguments a and b, it will generate a **list of integers from a up to** b-1 (b is excluded!). a should be less than b.

- If you call range(a) with one argument a, it generates a **list of integers from 0 up to a-1 (a** is excluded!).

- Note: if a is less than or equal to 0, range(a) will return an empty list [].

range()

```
>>> range(5)
    [0,1,2,3,4]
>>> range(1,5)
   [1,2,3,4]
>>> range(6,3)
 []
>>> range(-5,-9)
 []
>>> range(-9,-5)
[-9,-8,-7,-6]
```

docstrings (revisited)

Docstrings vs Comments

- Docstrings are for external use. They are meant to synthesize **what** **a function does so other** programmers using it don't have to read through its code.

- Comments are for internal use. They explain **how** **a** function accomplishes a task. Their purpose is to make code easier to read by future programmers.

Docstring for sum_range

```
def sum_range(num):
    sum = 0
    curr_number = 1
    while curr_number <= num:
        sum += curr_number
        curr_number += 1
    return sum
```

- Our docstring should specify that:

  - we expect a positive integer num
  - function returns the sum of all the integers between 1/num
  - 1 and num are included in the calculation.
  - num should be greater than or equal to 1

Docstring for sum_range

```
def sum_range(num):
    '''Return the sum of all integers between 1 and num
    (inclusive). Num is an integer >= 1.'''
    sum = 0
    curr_number = 1
    while curr_number <= num:
        sum += curr_number
        curr_number += 1
    return sum
```

Comments for sum_range

```
def sum_range(num):
    '''Return the sum of all integers between 1 and num
    (inclusive). Num is an integer >= 1.'''
    sum = 0
    curr_number = 1
    while curr_number <= num:
        sum += curr_number
        curr_number += 1
    return sum
```

Comments should describe what each line does and how the task is accomplished.

Comments for sum_range

```
def sum_range(num):
    sum = 0              # keeps running total
    curr_number = 1 # init. count
    # loop through numbers in
    # range until you reach num
    while curr_number <= num:
        # add the number to sum
        sum += curr_number
        # increment the number
        curr_number += 1
    # when loop finishes, sum will
    # equal desired quantity
    return sum
```

Testing

Testing in __main__

- This will be useful to know when completing your assignment.
- You've written a function. You think it does what it's supposed to, but how can you be sure?
- You should **test your function: try to call it with** different values, and see if the result is what you expect it to be.
- The place for testing code is the __main__ block of your program.

Summing the numbers in a range

```
def sum_range(num):
    sum = 0
    for curr in range(1 , num + 1):
        sum = sum + curr
    return sum

if __name__ == "__main__":
    print sum_range(4)  # should be 10
    print sum_range(5)  # should be 15
    print sum_range(1)  # border case: num == 1
```

Summing the numbers in a range

```
def sum_range(num):
    sum = 0
    for curr in range(1 , num + 1):
        sum = sum + curr
    return sum

if __name__ == "__main__":
    if sum_range(4) == 10:     # range(4) should be 10
        print "range(4) OK"
    else:
        print "range(4) FAILED"
```

Lists

Lists

- We've seen lists before—that's what **range()** returns.

- Lists are very powerful structures.
  - Lists can contain strings, numbers, even other lists.
  - They work very much like strings
    - You get pieces out with []
    - You can add lists together
    - You can use **for** loops on them
  - We can use them to process a variety of kinds of data.

Demonstrating lists

```
>>> mylist = ["This","is","a", 12]
>>> print mylist
['This', 'is', 'a', 12]
>>> print mylist[0]
This
>>> for i in mylist:
...         print i
...
This
is
a
12
>>> print mylist + ["Really!"]
['This', 'is', 'a', 12, 'Really!']
```

Examples

Factorial

```
def factorial(n):
    f = 1
    while (n > 0):
        f = f * n
        n = n - 1
    return f
```

What have we learnt today?

- Variable scope & Namespaces
- String Formatting
- Testing
- for-loops & range
- Lists

This Week's To Do List

- Go through lecture slides – make sure you try the code snippets
- Try the lecture's programs posted on course website