CSC108: Introduction to Computer Programming

Lecture 2

Wael Aboulsaadat

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

Recap of Lecture 1

- Variables & Types
- Assignment Statement
- Logical & Mathematical Operators
- if statement
- print
- input

- Example code

Variables revisited

- Every programming language has a list of reserved keywords that are used to parse the program correctly.

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

- No variable name can be the same as one of these words

Expressions revisited: Truth Tables

a	b	a and b
True	True	**True**
True	False	False
False	True	False
False	False	False

a	b	a or b
True	True	**True**
True	False	**True**
False	True	**True**
False	False	False

a	not a
True	False
False	**True**

Expressions revisited: brackets

- Use brackets to force precedence of evaluation
- Examples:

2 + 3 * 4

vs.

(2 + 3) * 4

10.0 – 4.0 / 2.0 / 22.0 + 19.0 * 2.0

vs.

(10.0 – (4.0 / 2.0 / 22.0) + 19.0) * 2.0

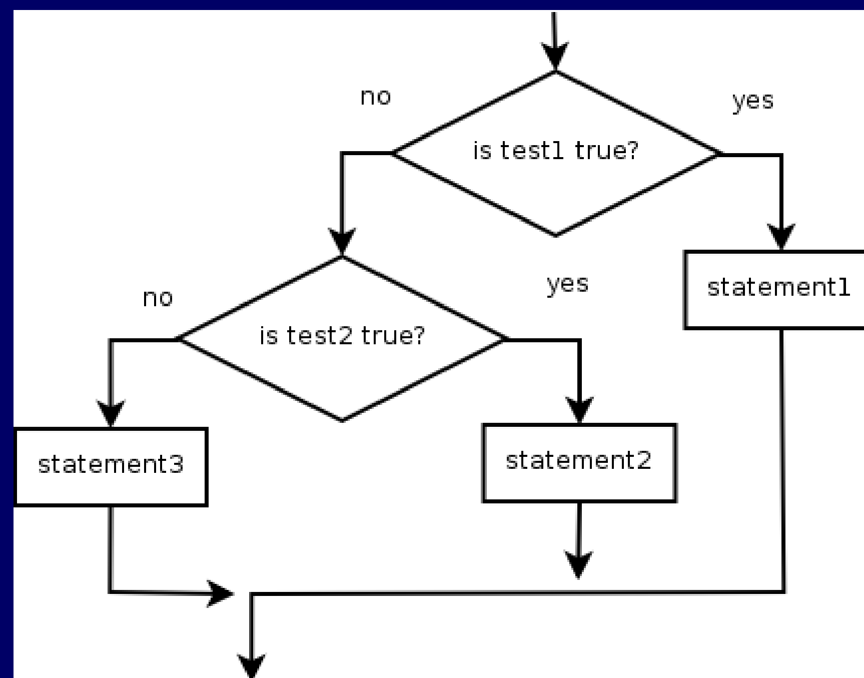
vs.

10.0 – 4.0 / ((2.0 / 22.0) + 19.0) * 2.0

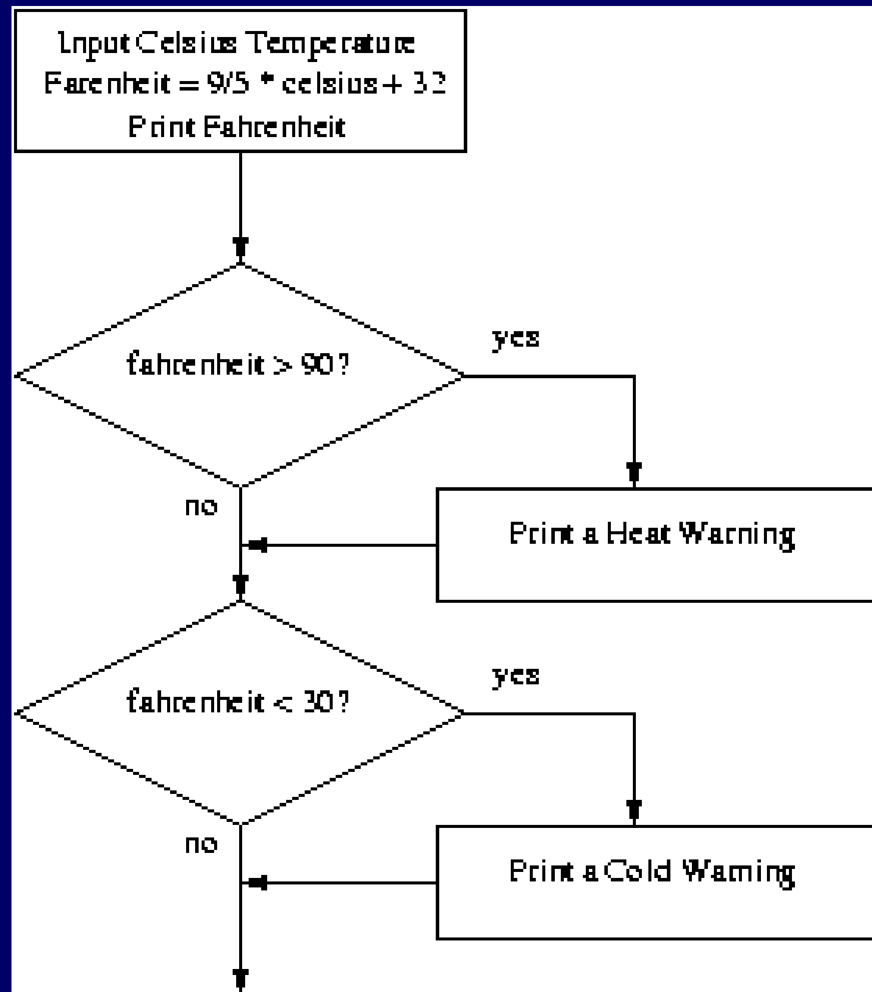
If-else revisited: chains

- Multiple conditions can be chained with elif ("else if"):

```
if condition:
    statements
elif condition:
    statements
else:
    statements
```



Example: temperature warning



Example: temperature warning

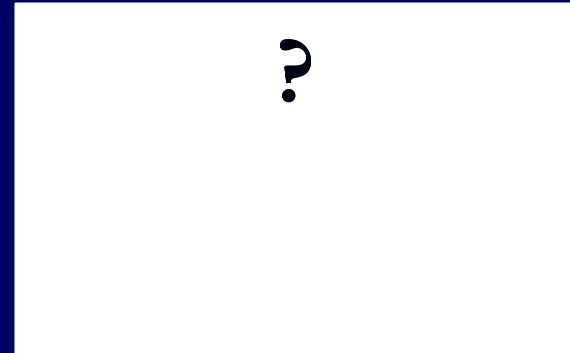
```
celsius = input("What is the Celsius temperature? ")
fahrenheit = 9 / 5 * celsius + 32
print("The temperature is", fahrenheit, "degrees fahrenheit.")
if fahrenheit >= 90:
    print("It's really hot out there, be careful!")
if fahrenheit <= 30:
    print("Brrrrr. Be sure to dress warmly")
```

Functions

- Basic building block of any program
- A typical python program consists of a set of functions working together to achieve some goal

Functions vs machines

- Similarities between a function and a machine !
  - Machine has an engine — function has an engine (body)
  - Machine is initialized with input — function takes input (input parameters)
  - Machine produces an output — function produces an output (return val)
  - Machine has a predetermined way to activate — function has an activation mechanism (invoke/call)



Functions vs machines

- Similarities between a function and a machine !
  - Machine has an engine | function has an engine (body)
  - Machine is initialized with input | function takes input (input parameters)
  - Machine produces an output | function produces an output (return val)
  - Machine has a predetermined way to activate | function has an activation mechanism (invoke/call)

- Differences between a function and a machine
  - Machine does not stop unless you deactivate while function stops after the last line
  - Machines are often independent of each other while functions are chained…

Function Syntax

Defining a function → def

name → function-name

inputs → Parameter1, Parameter2, etc…

```
def      function-name(Parameter1,
                       Parameter2 , etc…     ):
         statement
         statement
         statement
         statement

         return value
```

Function header

Tab not space!

Function body (the engine)

output → value

Function Example

name

input

```
def cube(  X   ):
        Product = X * X * X
    return Product
…
…
…
```

output

How do you get a function to work?

name

input

```
def cube(  X   ):
        Product = X * X * X
    return Product
…
…
…
Result = cube( 20 )
```

output

Call it and it will execute!

Function Parameter mapping

- When a function is called, the parameters are mapped 1-to-1 to arguments
- Example

```
def average(x,y,z)
        avg = (x + y + z) / 3.0
        return avg
num1 = input(" Enter number 1")
num2 = input(" Enter number 2")
num3 = input(" Enter number 3")
res= average( num1, num2, num3 )
print "the average of 3 numbers is", res
```

Function Parameter mapping

- When a function is called, the parameters are mapped 1-to-1 to arguments

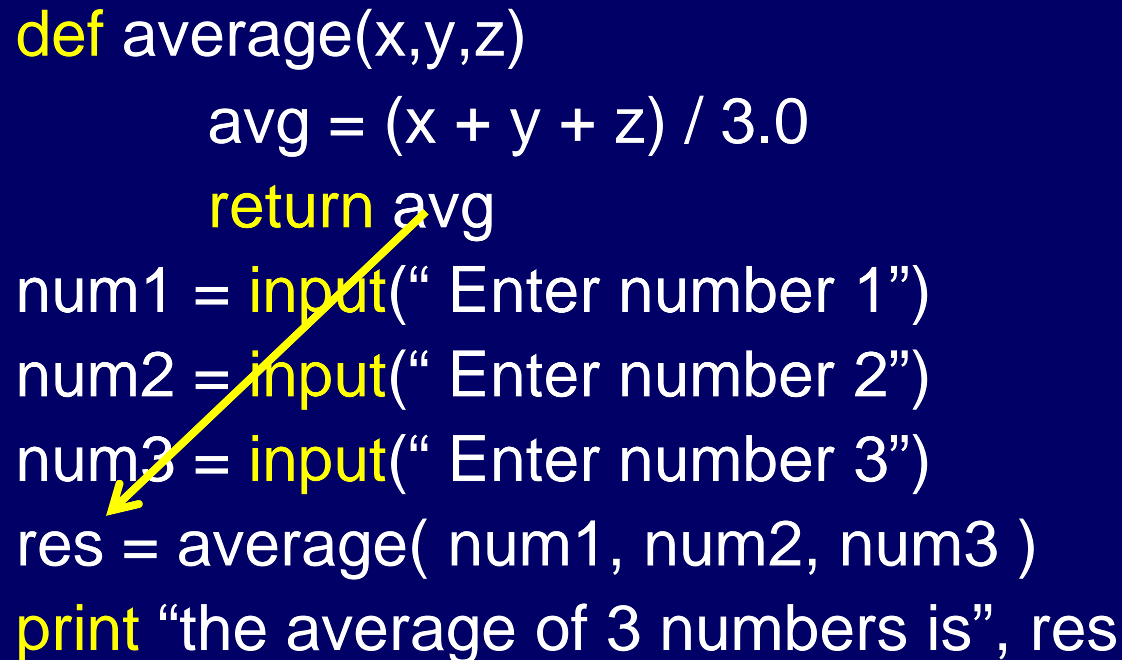
- Example

```
def average(x,y,z)
        avg = (x + y + z) / 3.0
        return avg
num1 = input(" Enter number 1")
num2 = input(" Enter number 2")
num3 = input(" Enter number 3")
res= average( num1, num2, num3 )
print "the average of 3 numbers is", res
```

Function Return

- When a function is called, the parameters are mapped 1-to-1 to arguments

- Example



```
def average(x,y,z)
        avg = (x + y + z) / 3.0
        return avg
num1 = input(" Enter number 1")
num2 = input(" Enter number 2")
num3 = input(" Enter number 3")
res = average( num1, num2, num3 )
print "the average of 3 numbers is", res
```

Functions and Math

- In math, we have a concept of a function as a relationship between two quantities.

- It is possible to model a mathematical function using a Python function:

Math:

$$f(x) = 2x + 5$$

Python:

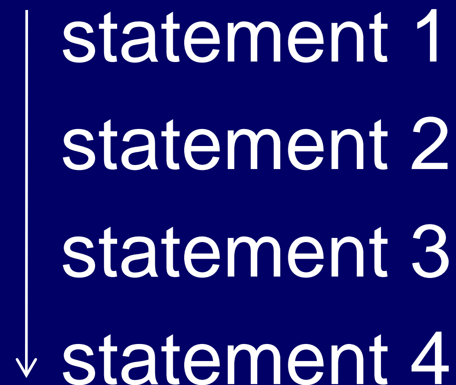
```
def f(x):
    return 2 * x + 5
```

Summary of Python Rules for Functions

- Define a new function using def
- Argument names follow in parentheses
  - No types for either return or parameters
- Finish at any time with return
  - Functions without return statements return None

Functions and Execution Flow

- A python program is interpreted one line at a time
- So far, we've been writing programs that follow a rigidly defined sequence of execution:



Functions and Execution Flow

- A function call causes the execution to transfer to the first line of code inside the function.
- After a functions finishes the next line after function call is executed

Functions and Execution Flow

- Example

```
def average(x,y,z):
        avg = (x + y + z) / 3.0
        return avg

1  ——→  num1 = input(" Enter number 1")
        num2 = input(" Enter number 2")
        num3 = input(" Enter number 3")
        res= average( num1, num2, num3 )
        print "the average of 3 numbers is", res
```

Functions and Execution Flow

- Example

```
def average(x,y,z)
        avg = (x + y + z) / 3.0
        return avg

1       num1 = input(" Enter number 1")
2  ---> num2 = input(" Enter number 2")
        num3 = input(" Enter number 3")
        res= average( num1, num2, num3 )
        print "the average of 3 numbers is", res
```

Functions and Execution Flow

- Example

```
def average(x,y,z)
        avg = (x + y + z) / 3.0
        return avg

1       num1 = input(" Enter number 1")
2       num2 = input(" Enter number 2")
3  ---> num3 = input(" Enter number 3")
        res= average( num1, num2, num3 )
        print "the average of 3 numbers is", res
```

Functions and Execution Flow

- Example

```
def average(x,y,z)
        avg = (x + y + z) / 3.0
        return avg

1       num1 = input(" Enter number 1")
2       num2 = input(" Enter number 2")
3       num3 = input(" Enter number 3")
4  ---> res= average( num1, num2, num3 )
        print "the average of 3 numbers is", res
```

Functions and Execution Flow

- Example

```
          def average(x,y,z)
5  ---->          avg = (x + y + z) / 3.0
                  return avg

1         num1 = input(" Enter number 1")
2         num2 = input(" Enter number 2")
3         num3 = input(" Enter number 3")
4         res= average( num1, num2, num3 )
          print "the average of 3 numbers is", res
```

Functions and Execution Flow

- Example

```
        def average(x,y,z)
5               avg = (x + y + z) / 3.0
6  ---->        return avg

1       num1 = input(" Enter number 1")
2       num2 = input(" Enter number 2")
3       num3 = input(" Enter number 3")
4       res= average( num1, num2, num3 )
        print "the average of 3 numbers is", res
```

Functions and Execution Flow

- Example

```
        def average(x,y,z)
5           avg = (x + y + z) / 3.0
6           return avg

1       num1 = input(" Enter number 1")
2       num2 = input(" Enter number 2")
3       num3 = input(" Enter number 3")
4       res= average( num1, num2, num3 )
7 ----> print "the average of 3 numbers is", res
```

Global Variables and Functions

- Variables defined outside functions are global
- Example

```
x = 100
def New_Value(param):
    product = param * x
    return product

y = New_Value (9)
print "new value", y, " is obtained by multiplying by ", x
```

Local Variables in Functions

- Variables created in functions are local to the function
- Example

```
def New_Value(param):
        x = 100
        product = param * x
        return product


y = New_Value (9)
print “new value”, y, “ is obtained by multiplying by “, x
```

Nested Function Calls

- You can have a function call inside a function call. This is called **nesting**

- **Example**

```
def f(x):
        return 2 * x + 5

def g(x):
        return x * 2
n=f(g(6))        ➔ m = g(6)
                   n = f(m)
```

Where is the start of the program?

- Mixing code not placed in functions and functions makes the program hard to read!
  - Example:

- Python has a way to mark the start of a program

```
if __name__ == '__main__':
```

Modules: python program files .py

- Python programs are saved in .py files, which are **plain text files (you can open them in any text** editor)

Modules

- Modules are additional pieces of code that further extend Python's functionality
- A module typically has a specific function
  - additional math functions, databases, network…
- Python comes with many useful modules

Modules: importing

- Modules are accessed using import

```
import somefile
from somefile import *
from somefile import subset
```

- Modules can have subsets of functions
  - os.path is a subset within os
- Modules are then addressed by modulename.function()
  - filename = os.path.splitext("points.txt")

Built-in Functions in Python

- Many popular functions already coded for you
- http://docs.python.org/library/functions.html

Type Conversion

- Some built-in functions allow you to convert between data types:

function	converts to
`str()`	a str value
`int()`	an int value
`float()`	a float value
`long()`	a long value

Type conversion & raw_input

- There's a built-in function that allows a Python program to ask for data from the user before continuing:

  raw_input(*prompt_string)*

- The result of a raw_input() call is a **string** which can be assigned to a variable.

Getting help

- You can find out what a function does by using another built-in, help(*functionname)*

Comments: making your programs understandable

- A **comment is a plain English note inserted into a** piece of code to make it more readable.

- In order not to try to execute comments, Python will ignore any line that starts with a #.

- Comments should be used to explain what your code does, for any piece of code whose purpose isn't immediately obvious.

What have we learnt today?

- Functions
- Type conversion & input
- Comments

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