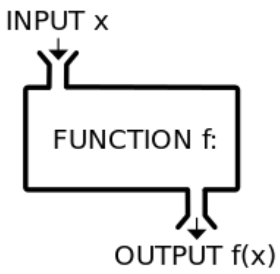


Functions and Abstraction

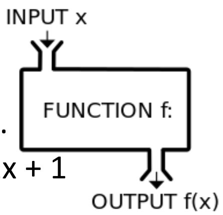
BBM 101 - Introduction to Programming I

Hacettepe University
Fall 2016

Fuat Akal, Aykut Erdem, Erkut Erdem

Slides based on material prepared by Ruth Anderson, Michael Ernst and Bill Howe in the course CSE 140
University of Washington

Functions



- In math, you use functions: sine, cosine, …
- In math, you define functions: $f(x) = x^2 + 2x + 1$

- A function packages up and names a computation
- Enables re-use of the computation (generalization)
- **D**on't **R**epeat **Y**ourself (DRY principle)
- Shorter, easier to understand, less error-prone

- Python lets you use and define functions
- We have already seen some Python functions:
  - **`len`**, **`float`**, **`int`**, **`str`**, **`range`**

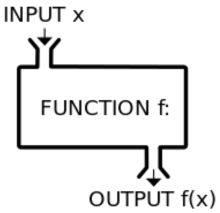
Using ("calling") a Function

```
len("hello")          len("")
round(2.718)          round(3.14)
pow(2, 3)             range(1, 5)
math.sin(0)           math.sin(math.pi / 2)
```

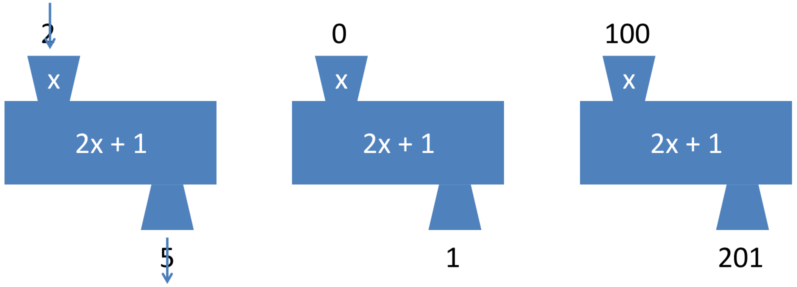
- Some need no input:
  `random.random()`

- All produce output

A Function is a Machine



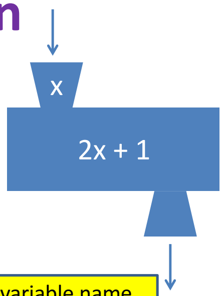
- You give it input
- It produces a result (output)



In math: func(x) = 2x + 1

Creating a Function

Define the machine,
including the input and the result



Name of the function.
Like “y = 5” for a variable

Keyword that means:
I am **def**ining a function

Input variable name,
or “formal parameter”

```
def dbl_plus(x):
  return 2*x + 1
```

Keyword that means:
This is the result

Return expression
(part of the **return** statement)

More Function Examples

Define the machine, including the input and the result

```
def square(x):
  return x * x

def fahr_to_cent(fahr):
  return (fahr – 32) / 9.0 * 5

def cent_to_fahr(cent):
  result = cent / 5.0 * 9 + 32
  return result

def abs(x):
  if x < 0:
    return – x
  else:
    return x
```

```
def print_hello():
  print("Hello, world")
```

No `return` statement
Returns the value `None`
Are also called ‘procedures’

```
def print_fahr_to_cent(fahr):
  result = fahr_to_cent(fahr)
  print(result)
```

What is the result of:

```
x = 42
square(3) + square(4)
print(x)
boiling = fahr_to_cent(212)
cold = cent_to_fahr(-40)
print(result)
print(abs(-22))
print(print_fahr_to_cent(32))
```

Python Interpreter

- An expression evaluates to a value
  - Which can be used by the containing expression or statement
- `print("test")` statement writes text to the screen
- The Python interpreter (command shell) reads statements and expressions, then executes them
- If the interpreter executes an expression, it prints its value
- In a program, evaluating an expression does not print it
- In a program, printing an expression does not permit it to be used elsewhere

How Python Executes a Function Call

Function definition

```
def square(x):
  return x * x
```

Formal parameter (a variable)

```
square(3 + 4)
```

Actual argument

Function call or function invocation

Current expression:

```
1 + square(3 + 4)
1 + square(7)
```

evaluate this expression

```
return x * x
return 7 * x
return 7 * 7
return 49
```

Variables:
x: 7

```
1 + 49
50
```

1. Evaluate the argument (at the call site)
2. Assign the formal parameter name to the argument’s value
   - A *new* variable, not reuse of any existing variable of the same name
3. Evaluate the statements in the body one by one
4. At a return statement:
   - Remember the value of the expression
   - Formal parameter variable disappears – exists only during the call!
   - The call expression evaluates to the return value

Example of Function Invocation

```
def square(x):
  return x * x
                                      Variables:
square(3) + square(4)                 (none)
return x * x                          x: 3
return 3 * x                          x: 3
return 3 * 3                          x: 3
return 9                              x: 3
9 + square(4)                         (none)
    return x * x                      x: 4
    return 4 * x                      x: 4
    return 4 * 4                      x: 4
    return 16                         x: 4
9 + 16                                (none)
25                                    (none)
```

Expression with Nested Function Invocations: Only One Executes at a Time

```
def fahr_to_cent(fahr):
  return (fahr – 32) / 9.0 * 5

def cent_to_fahr(cent):
  return cent / 5.0 * 9 + 32
                                              Variables:
fahr_to_cent(cent_to_fahr(20))                (none)
             return cent / 5.0 * 9 + 32       cent: 20
             return 20 / 5.0 * 9 + 32         cent: 20
             return 68                        cent: 20
fahr_to_cent(68)                              (none)
return (fahr – 32) / 9.0 * 5                  fahr: 68
return (68 – 32) / 9.0 * 5                    fahr: 68
return 20                                     fahr: 68
20                                            (none)
```

Expression with Nested Function Invocations: Only One Executes at a Time

```
def square(x):
  return x * x
                                      Variables:
square(square(3))                     (none)
       return x * x                   x=3
       return 3 * x                   x=3
       return 3 * 3                   x=3
       return 9                       x=3
square(9)                             (none)
    return x * x                      x=9
    return 9 * x                      x=9
    return 9 * 9                      x=9
    return 81                         x=9
81                                    (none)
```

Function that Invokes Another Function: Both Function Invocations are Active

```
def square(z):
  return z*z
def hypoten_use(x, y):
  return math.sqrt(square(x) + square(y))
                                              Variables:
hypoten_use(3, 4)                             (none)
  return math.sqrt(square(x) + square(y))     x: 3   y:4
  return math.sqrt(square(3) + square(y))     x: 3   y:4
    return z*z                                z: 3
    return 3*3                                z: 3
    return 9                                  z: 3
  return math.sqrt(9 + square(y))             x: 3   y:4
  return math.sqrt(9 + square(4))             x: 3   y:4
    return z*z                                z: 4
    return 4*4                                z: 4
    return 16                                 z: 4
  return math.sqrt(9 + 16)                    x: 3   y:4
  return math.sqrt(25)                        x: 3   y:4
  return 5                                    x: 3   y:4
5                                             (none)
```

Shadowing of Formal Variable Names

```
def square(x):
  return x*x
def hypotenuse(x, y):
  return math.sqrt(square(x) + square(y))
```

Same formal parameter name

	Variables:
hypotenuse(3, 4)	(none)
return math.sqrt(square(x) + square(y))	x:3 y:4
return math.sqrt(square(3) + square(y))	x:3 y:4
return x*x	x:3
return 3*3	x:3
return 9	x:3
return math.sqrt(9 + square(y))	x:3 y:4
return math.sqrt(9 + square(4))	x:3 y:4
return x*x	x: 4
return 4*4	x: 4
return 16	x: 4
return math.sqrt(9 + 16)	x:3 y:4
return math.sqrt(25)	x:3 y:4
return 5	x:3 y:4
5	(none)

Formal parameter is a *new* variable

Shadowing of Formal Variable Names

```
def square(x):
  return x*x
def hypotenuse(x, y):
  return math.sqrt(square(x) + square(y))
```

Same diagram, with *variable scopes* or *environment frames* shown explicitly

	Variables:	
hypotenuse(3, 4)	(none)	hypotenuse()
return math.sqrt(square(x) + square(y))		x:3 y:4
return math.sqrt(square(3) + square(y))	square()	x:3 y:4
return x*x	x: 3	x:3 y:4
return 3*3	x: 3	x:3 y:4
return 9	x: 3	x:3 y:4
return math.sqrt(9 + square(y))		x:3 y:4
return math.sqrt(9 + square(4))	square()	x:3 y:4
return x*x	x: 4	x:3 y:4
return 4*4	x: 4	x:3 y:4
return 16	x: 4	x:3 y:4
return math.sqrt(9 + 16)		x:3 y:4
return math.sqrt(25)		x:3 y:4
return 5		x:3 y:4
5	(none)	

In a Function Body, Assignment Creates a Temporary Variable (like the formal parameter)

```
stored = 0
def store_it(arg):
  stored = arg
  return stored
★ y = store_it(22)
print(y)
★ print(stored)
```

Show evaluation of the starred expressions:

	Variables:	Global or top level
y = store_it(22)	store_it()	stored: 0
stored = arg; return stored	arg: 22	stored: 0
stored = 22; return stored	arg: 22	stored: 0
return stored	arg: 22 stored: 22	stored: 0
return 22	arg: 22 stored: 22	stored: 0 y: 22
y = 22		stored: 0 y: 22
print(stored)		stored: 0 y: 22
print(0)		

How to Look Up a Variable

Idea: find the nearest variable of the given name

1. Check whether the variable is defined in the local scope
2. … check any intermediate scopes (**none** in BBM 101!) …
3. Check whether the variable is defined in the global scope

If a local and a global variable have the same name, the global variable is inaccessible ("shadowed")

This is confusing; try to avoid such shadowing

```
x = 22
stored = 100
def lookup():
    x = 42
    return stored + x
lookup()
x = 5
stored = 200
lookup()
```

```
def lookup():
    x = 42
    return stored + x
x = 22
stored = 100
lookup()
x = 5
stored = 200
lookup()
```

What happens if we define `stored` *after* `lookup`?

Local Variables Exist Only while the Function is Executing

```
def cent_to_fahr(cent):
    result = cent / 5.0 * 9 + 32
    return result

tempf = cent_to_fahr(15)
print(result)

NameError: name 'result' is not defined
```

Use Only the Local and the Global Scope

```
myvar = 1

def outer():
    myvar = 1000
    return inner()

def inner():
    return myvar

print(outer())
```

The handouts have a more precise rule, which applies when you define a function inside another function.

Abstraction

- Abstraction = ignore some details
- Generalization = become usable in more contexts
- Abstraction over computations:
  - functional abstraction, a.k.a. procedural abstraction
- As long as you know what the function means, you don't care how it computes that value
  - You don't care about the *implementation* (the function body)

Defining Absolute Value

```
def abs(x):
  if val < 0:
    return -1 * val
  else:
    return 1 * val
```

```
def abs(x):
  if val < 0:
    result = - val
  else:
    result = val
  return result
```

```
def abs(x):
  if val < 0:
    return - val
  else:
    return val
```

```
def abs(x):
  return math.sqrt(x*x)
```

Defining Round (for positive numbers)

```
def round(x):
  return int(x+0.5)


def round(x):
  fraction = x - int(x)
  if fraction >= .5:
    return int(x) + 1
  else:
    return int(x)
```

Two Types of Documentation

1. Documentation for users/clients/callers
   - Document the *purpose* or *meaning* or *abstraction* that the function represents
   - Tells what the function does
   - Should be written for *every* function
2. Documentation for programmers who are reading the code
   - Document the *implementation* – specific code choices
   - Tells how the function does it
   - Only necessary for tricky or interesting bits of the code

For users: a string as the first element of the function body

For programmers: arbitrary text after #

```
def square(x):
  """Returns the square of its argument."""
  # "x*x" can be more precise than "x**2"
  return x*x
```

Multi-line Strings

- New way to write a string – surrounded by three quotes instead of just one
  - `"hello"`
  - `'hello'`
  - `"""hello"""`
  - `'''hello'''`
- Any of these works for a documentation string
- Triple-quote version:
  - can include newlines (carriage returns), so the string can span multiple lines
  - can include quotation marks

Don’t Write Useless Comments

- Comments should give information that is not apparent from the code
- Here is a counter-productive comment that merely clutters the code, which makes the code *harder* to read:

```
# increment the value of x
x = x + 1
```

Where to Write Comments

- By convention, write a comment *above* the code that it describes (or, more rarely, on the same line)
  - First, a reader sees the English intuition or explanation, then the possibly-confusing code

```
# The following code is adapted from
# "Introduction to Algorithms", by Cormen et al.,
# section 14.22.
while (n > i):
    ...
```

- A comment may appear anywhere in your program, including at the end of a line:

```
x = y + x    # a comment about this line
```

- For a line that starts with **#**, indentation must be consistent with surrounding code

Each Variable Should Represent One Thing

```
def atm_to_mbar(pressure):
    return pressure * 1013.25

def mbar_to_mmHg(pressure):
    return pressure * 0.75006

# Confusing
pressure = 1.2 # in atmospheres
pressure = atm_to_mbar(pressure)
pressure = mbar_to_mmHg(pressure)
print(pressure)

# Better
in_atm = 1.2
in_mbar = atm_to_mbar(in_atm)
in_mmHg = mbar_to_mmHg(in_mbar)
print(in_mmHg)
```

```
# Best
def atm_to_mmHg(pressure):
    in_mbar = atm_to_mbar(pressure)
    in_mmHg = mbar_to_mmHg(in_mbar)
    return in_mmHg
print(atm_to_mmHg(1.2))
```

Corollary: Each variable should contain values of only one type

```
# Legal, but confusing: don't do this!
x = 3
…
x = "hello"
…
x = [3, 1, 4, 1, 5]
…
```

If you use a descriptive variable name, you are unlikely to make these mistakes

Exercises

```
def cent_to_fahr(c):
  print(cent / 5.0 * 9 + 32)

print(cent_to_fahr(20))
```

```
def myfunc(n):
  total = 0
  for i in range(n):
    total = total + i
  return total

print(myfunc(4))
```

```
def c_to_f(c):
    print("c_to_f")
    return c / 5.0 * 9 + 32

def make_message(temp):
    print("make_message")
    return ("The temperature is "
+ str(temp))

for tempc in [-40,0,37]:
    tempf = c_to_f(tempc)
    message = make_message(tempf)
    print(message)
```

```
double(7)
```

```
abs(-20 - 2) + 20
```

Use the Python Tutor:
http://pythontutor.com/

What Does This Print?

```
def cent_to_fahr(cent):
    print (cent / 5.0 * 9 + 32)

print (cent_to_fahr(20))
```

```
68.0
None
```

What Does This Print?

```
def myfunc(n):
    total = 0
    for i in range(n):
        total = total + i
    return total

print (myfunc(4))
```

6

What Does This Print?

```
def c_to_f(c):
    print ("c_to_f")
    return c / 5.0 * 9 + 32

def make_message(temp):
    print ("make_message")
    return "The temperature is " + str(temp)

for tempc in [-40,0,37]:
    tempf = c_to_f(tempc)
    message = make_message(tempf)
    print (message)
```

```
c_to_f
make_message
The temperature is -40.0
c_to_f
make_message
The temperature is 32.0
c_to_f
make_message
The temperature is 98.6
```

Decomposing a Problem

- Breaking down a program into functions is *__the fundamental activity__* of programming!
- How do you decide when to use a function?
  - One rule: DRY (Don’t Repeat Yourself)
  - Whenever you are tempted to copy and paste code, don’t!
- Now, how do you design a function?

How to Design a Function

1. Wishful thinking: Write the program as if the function already exists

2. Write a specification: Describe the inputs and output, including their types

No implementation yet!

3. Write tests: Example inputs and outputs

4. Write the function body (the implementation)

First, write your plan in English, then translate to Python

```
print("Temperature in Farenheit:", tempf)
tempc = fahr_to_cent(tempf)
print("Temperature in Celsius:", tempc)

def fahr_to_cent(f):
   """Input: a number representing degrees Farenheit
   Return value: a number representing degrees
centigrade
   """
   result = (f – 32) / 9.0 * 5
   return result

assert fahr_to_cent(32) == 0
assert fahr_to_cent(212) == 100
assert fahr_to_cent(98.6) == 37
assert fahr_to_cent(-40) == -40
```

Review: How to Evaluate a Function Call

1. Evaluate the function and its arguments to values
   - If the function value is not a function, execution terminates with an error
2. Create a new stack frame
   - The parent frame is the one where the function is defined
     - In CSE 140, this is always the global frame
   - A frame has bindings from variables to values
   - Looking up a variable starts here
     - Proceeds to the next older frame if no match here
     - The oldest frame is the “global” frame
     - All the frames together are called the “environment”
   - Assignments happen here
3. Assign the actual argument values to the formal parameter variable
   - In the new stack frame
4. Evaluate the body
   - At a return statement, remember the value and exit
   - If at end of the body, return `None`
5. Remove the stack frame
6. The call evaluates to the returned value

Functions are Values
The Function can be an Expression

```
import math
def double(x):
    return 2*x
print(double)
myfns = [math.sqrt, int, double, math.cos]
myfns[1](3.14)
myfns[2](3.14)
myfns[3](3.14)

def doubler():
    return double
doubler()(2.718)
```