



Functions and abstraction

Michael Ernst

UW CSE 140

Winter 2013



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)
- **Don't Repeat Yourself** (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)
```

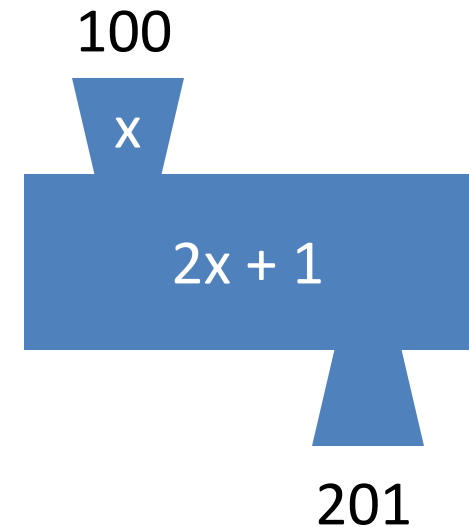
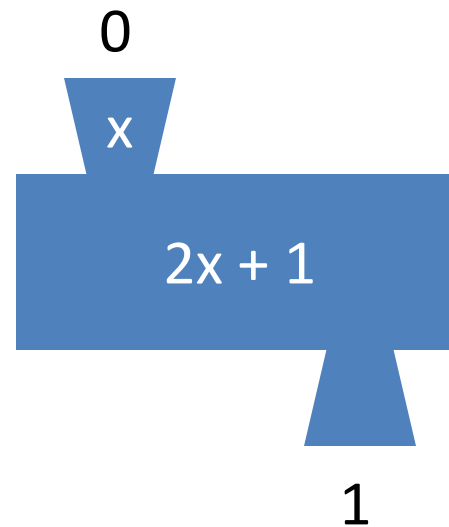
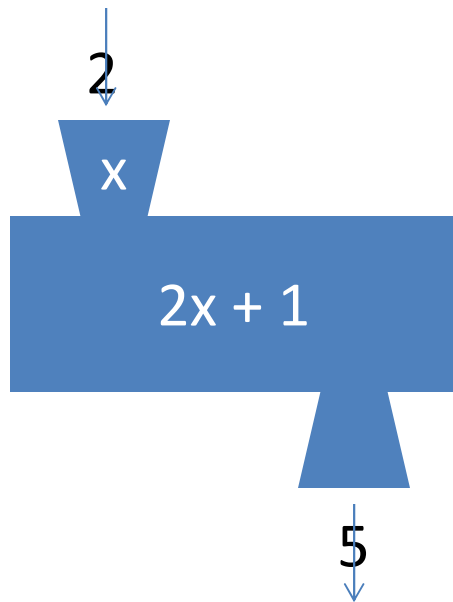
```
math.sin(0)
```

```
math.sin(math.pi / 2)
```

- Some need no input: `random.random()`
- All produce output
- What happens if you forget the parentheses on a function call? `random.random`
 - Functions are values too
 - Types we know about:
int, float, str, bool, list, function

A function is a machine

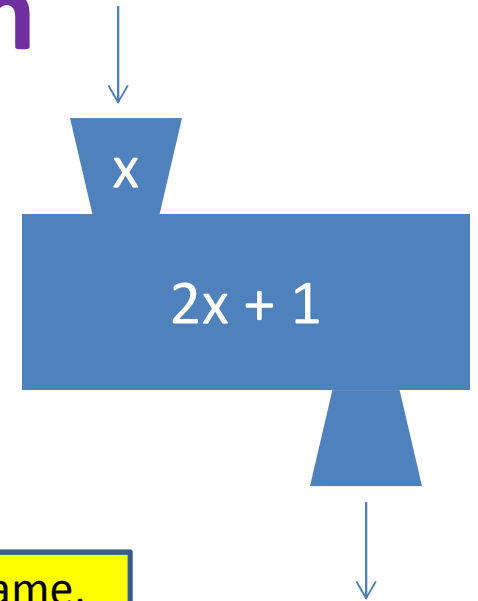
- You give it input
- It produces a result



In math: $\text{func}(x) = 2x + 1$

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 func(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**
Executed for side effect

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

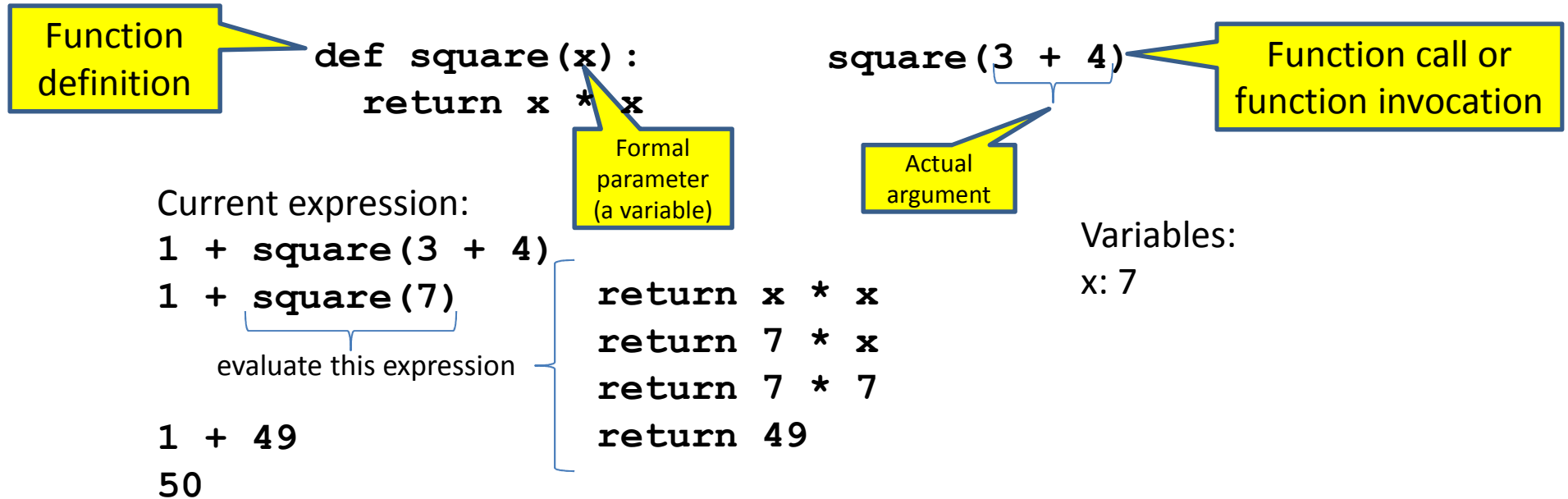
What is the result of:

```
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)
```

Digression: Two types of output

- An expression evaluates to a value
 - Which can be used by the containing expression or statement
- A **print** 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



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
```

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

```
return x * x
```

```
return 3 * 3
```

```
return 3 * 3
```

```
return 9
```

```
9 + square(4)
```

```
return x * x
```

```
return 4 * 4
```

```
return 4 * 4
```

```
return 16
```

```
9 + 16
```

```
25
```

Variables:

(none)

x: 3

x: 3

x: 3

x: 3

(none)

x: 4

x: 4

x: 4

x: 4

(none)

(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
```

```
fahr_to_cent(cent_to_fahr(20))  
    return cent / 5.0 * 9 + 32  
    return 20 / 5.0 * 9 + 32  
    return 68
```

```
fahr_to_cent(68)  
return (fahr - 32) / 9.0 * 5  
return (68 - 32) / 9.0 * 5  
return 20
```

20

Variables:

(none)
cent: 20
cent: 20
cent: 20
(none)
fahr: 68
fahr: 68
fahr: 68
(none)

Expression with nested function invocations: Only one executes at a time

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

```
square(square(3))
```

```
    return x * x  
    return 3 * x  
    return 3 * 3  
    return 9
```

```
square(9)
```

```
    return x * x  
    return 9 * x  
    return 9 * 9  
    return 81
```

81

Variables:

(none)

x: 3

x: 3

x: 3

x: 3

(none)

x: 9

x: 9

x: 9

x: 9

(none)

Function that invokes another function:

Both function invocations are active

```
def square(z):
```

```
    return z*z
```

```
def hypotenuse(x, y):
```

```
    return math.sqrt(square(x) + square(y))
```

```
hypotenuse(3, 4)
```

```
    return math.sqrt(square(x) + square(y))
```

```
    return math.sqrt(square(3) + square(y))
```

```
        return z*z
```

```
        return 3*3
```

```
        return 9
```

```
    return math.sqrt(9 + square(y))
```

```
    return math.sqrt(9 + square(4))
```

```
        return z*z
```

```
        return 4*4
```

```
        return 16
```

```
    return math.sqrt(9 + 16)
```

```
    return math.sqrt(25)
```

```
    return 5
```

Variables:

(none)

x: 3 y:4

x: 3 y:4

z: 3 x: 3 y:4

z: 3 x: 3 y:4

z: 3 x: 3 y:4

x: 3 y:4

x: 3 y:4

z: 4 x: 3 y:4

z: 4 x: 3 y:4

z: 4 x: 3 y:4

x: 3 y:4

x: 3 y:4

x: 3 y:4

(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

```
hypotenuse(3, 4)  
    return math.sqrt(square(x) + square(y))  
    return math.sqrt(square(3) + square(y))  
        return x*x  
        return 3*3  
        return 9  
    return math.sqrt(9 + square(y))  
    return math.sqrt(9 + square(4))  
        return x*x  
        return 4*4  
        return 16  
    return math.sqrt(9 + 16)  
    return math.sqrt(25)  
    return 5
```

Variables:

```
(none)  
x:3 y:4  
x:3 y:4  
x:3 x:3 y:4  
x:3 x:3 y:4  
x:3 x:3 y:4  
x:3 y:4  
x:3 y:4  
x:4 x:3 y:4  
x:4 x:3 y:4  
x:4 x:3 y:4  
x:3 y:4  
x:3 y:4  
x:3 y:4  
(none)
```

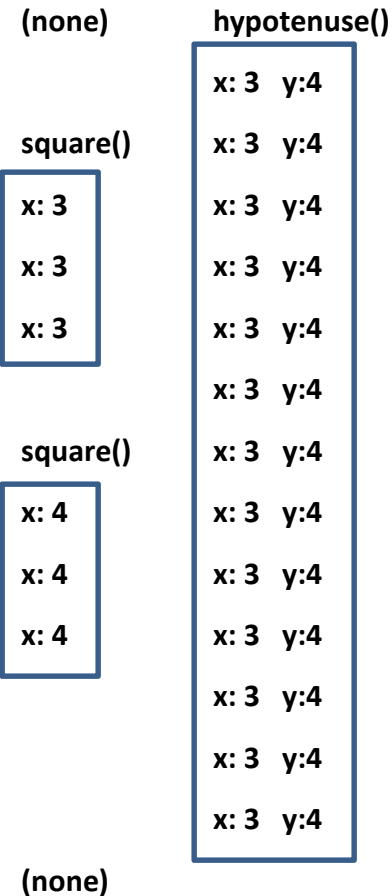
Formal parameter
is a *new* variable

Shadowing of formal variable names

```
def square(x):  
    return x*x  
def hypotenuse(x, y):  
    return math.sqrt(square(x) + square(y))  
  
hypotenuse(3, 4)  
    return math.sqrt(square(x) + square(y))  
    return math.sqrt(square(3) + square(y))  
        return x*x  
        return 3*3  
        return 9  
    return math.sqrt(9 + square(y))  
    return math.sqrt(9 + square(4))  
        return x*x  
        return 4*4  
        return 16  
    return math.sqrt(9 + 16)  
    return math.sqrt(25)  
    return 5
```

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

Variables:



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:

```
y = store_it(22)
    stored = arg; return stored
    stored = 22; return stored
    return stored
    return 22
y = 22
print stored
print 0
```

Variables:

Global or top level

store_it()

```
arg: 22
arg: 22
arg: 22 stored: 22
arg: 22 stored: 22
```

```
stored: 0
stored: 0
stored: 0
stored: 0
stored: 0 y: 22
stored: 0 y: 22
stored: 0 y: 22
```

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 CSE 140!) ...
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
```

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:  
        return - val  
    else:  
        return val
```

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

```
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(5)
```

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)
```

What does this print?

```
def myfunc (n) :  
    total = 0  
    for i in range (n) :  
        total = total + i  
    return total  
  
print myfunc (4)
```

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

```
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)
```