Functions and abstraction

Michael Ernst

UW CSE 190p

Summer 2012

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

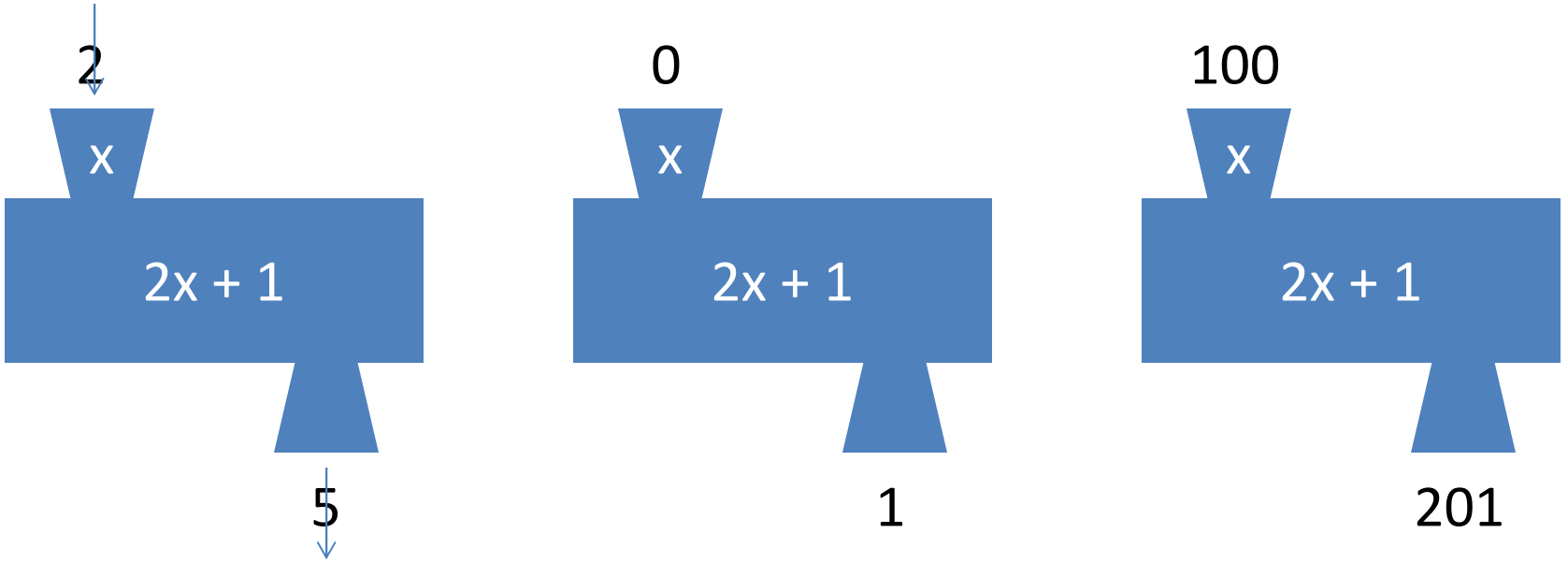
Using ("calling") a function

```
len("hello")        len("")
round(2.718)        round(3.14)
pow(2, 3)
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

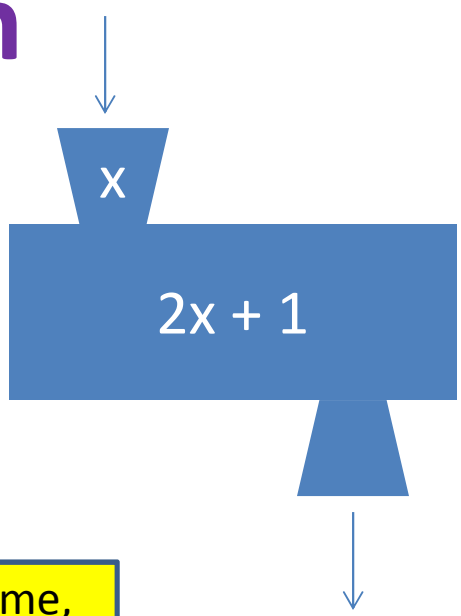
- You give it input
- It produces a result



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

Creating a function

Define the machine,
including the input and the result



Name of the function.
Like "x = 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 print_hello():
  print "Hello, world"

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

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

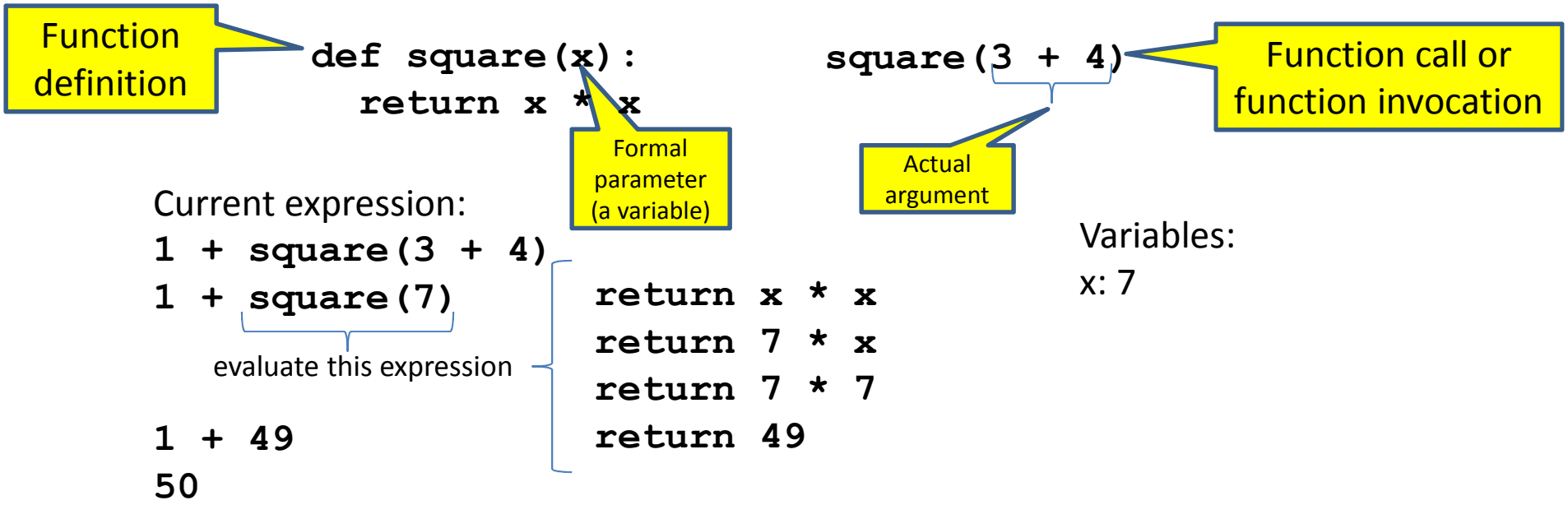
Returns the value **None**

Digression: Two types of output

- An expression evaluates to a value
  - Which can be used by the rest of the program
- 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

Examples of function invocation

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

```
square(3) + square(4)
return x * x
return 3 * x
return 3 * 3
return 9
9 + square(4)
      return x * x
      return 4 * x
      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)

Examples of function invocation

```
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)

Examples of function invocation

```
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)

Examples of function invocation

```
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
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)

Examples of function invocation

```
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
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

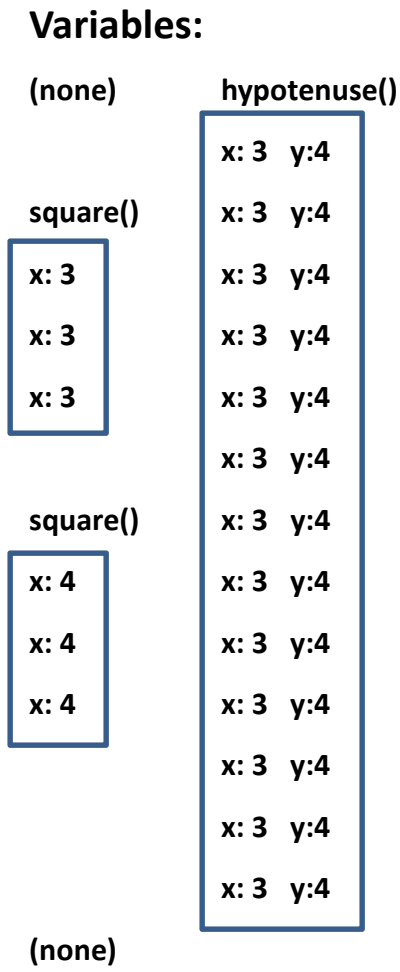
x: 3 y:4

(none)

Examples of function invocation

```
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
5
```



In a function body, assignment creates a temporary variable (like the formal parameter)

```
stored = 0
def store_it(arg):
  stored = arg
  return stored
print store_it(22)                    # prints 22
print stored                          # prints 0

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

Variables:

store_it()		Global or top level
x: 22		stored: 0
x: 22		stored: 0
x: 22	stored: 22	stored: 0
x: 22	stored: 22	stored: 0
		stored: 0
		stored: 0

A variable use finds the nearest variable of the given name

Looking up a global variable works if no local of the same name exists

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

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
   - 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

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 it *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