CS61A Lecture 2
Functions and Applicative Model of Computation

Tom Magrino and Jon Kotker
UC Berkeley EECS
June 19, 2012

CS in the News

Teaching goes social!


HOME PAGE | TODAY'S PAPER | VIDEO | MOST POPULAR | Edition: U.S. / Global

The New York Times — U.S.

WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION

POLITICS EDUCATION TEXAS

Teachers' Union to Open Lesson-Sharing Web Site

By MOTOKO RICH
Published: June 18, 2012

Teachers seeking lesson plans on anything from fractions and multiplication tables to Shakespeare or the Civil War can take to the Internet for a boundless array of work sheets, reading lists and classroom tips, available free or for a fee.

Connect With Us on Twitter
Follow @NYTNational for breaking news and headlines.
Twitter List: Reporters and Editors

Now the American Federation of Teachers is forming a partnership with TSL Education, the British publisher of the weekly Times Educational Supplement, to create a Web site where teachers can share curriculum materials with one another.

FACEBOOK
TWITTER
GOOGLE+
EMAIL
SHARE
PRINT
REPRINTS

http://www.nytimes.com/2012/06/19/us/teachers-union-to-open-lesson-sharing-web-site.html?_r=1&ref=education

Announcements

- Project 1 is out, due 6/29.
- Homework 1 is out, due 6/22.
- Lab 1 solutions are up.
  - They will always be up the day after the lab.
- You should all have lab accounts by now.
  - If you don't, please talk to your TA.
- CSUA will hold a UNIX help session on Tuesday, June 26 at 8pm in 310 Soda.

Today

- Expressions and Statements
- Functions
- A Basic Model of Evaluation
- Other Features

The Elements of Programming

- Primitive Expressions and Statements
  - *Simplest building blocks of language*
- Means of Combination
  - *Compound elements are built from simpler ones*
- Means of Abstraction
  - *Compound elements can be named and manipulated as units*

The Elements of Programming

"Computer science deals with the theoretical foundations of **DATA** *and* **FUNCTIONS,** *together with practical techniques for the implementation and application of these foundations"*

– Wikipedia

Functions and Data

- **Data:** Stuff we want to manipulate

"The Art of Computer Programming"

33

Alan Turing

This slide

- **Functions:** Rules for manipulating data

Count the words in a line of text

Pronounce someone's name

Display a dancing bear

Load the next slide

7

Expressions

An **expression** describes a *computation* and evaluates to a *value.*

18 + 69

$\sqrt{3493161}$

f(x)

$\sum_{i=1}^{100} i$

sin(π)

$|-1984|$

$\binom{69}{18}$

$f\left(f\left(f(x)\right)\right)+g(x)$

8

Expressions

There are two kinds of expressions: *primitive expressions* and *compound expressions*.

9

Expressions

- **Primitive Expressions** – expressions that directly describe their value.

2 — Numbers

x — Identifiers

"The rain in Spain" — Strings

10

Expressions

- **Compound Expressions** – expressions that are composed of simpler expressions.

add(2, 3)

mul(6, add(2, 3))

6 * (2 + 3)

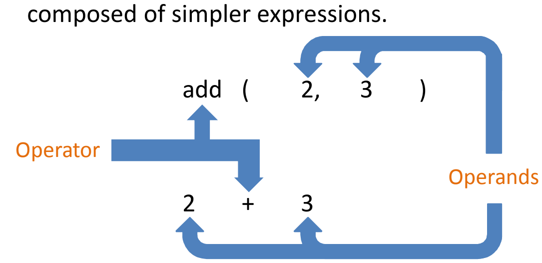
Infix Expressions

Function Calls (Call Expressions)

11

Expressions

- **Compound Expressions** – expressions that are composed of simpler expressions.



12

Evaluating Expressions

If we are evaluating a **primitive expression**, simply use the value it represents.

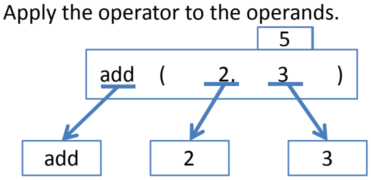
3 → 3

x → <whatever x holds>

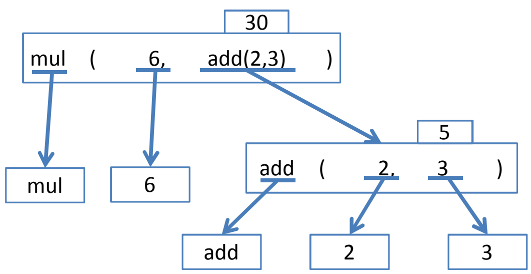
Evaluating Expressions

If we are evaluating a **compound expression**:

1. Evaluate the operator and operands.
2. Apply the operator to the operands.



Evaluating Expressions



Statements

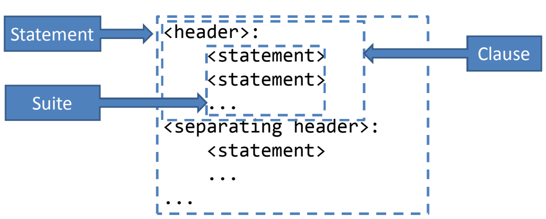
A **statement** is executed by the program to perform an *action*. It can either be a single line of code (simple) or a series of lines (compound).

x = 6 # assignment statement

print("5 is 2 + 3") # expression statement

Compound Statements

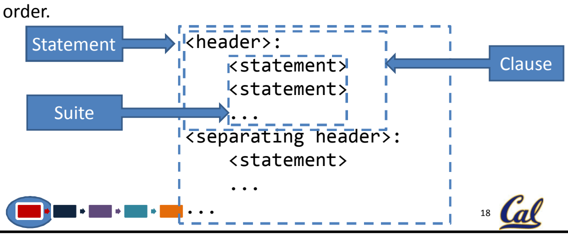
Compound statements are statements that are composed of *multiple* statements.



Evaluating Compound Statements

The first **header** determines the type of compound statement we have. The way a compound statement is evaluated depends on the type of compound statement you are using.

A **suite** is a series of statements, which are evaluated one by one in order.



The `while` statement

```
while <boolean expression>:
    <suite>
```


The `while` statement

Predict the output of the following:

```
>>> n = 7
>>> while n > 0:
...     n = n - 1
...     print(n)
...
```


The `while` statement

Predict the output of the following:

```
>>> n = 0
>>> while n < 0:
...     n = n + 1
...     print(n)
...
```


The `while` statement

Predict the output of the following:

```
>>> n = 1
>>> while n > 0:
...     n = n + 1
...     print(n)
...
```


The `if` statement

```
if <boolean expression>:
    <suite>
elif <boolean expression>:        Optional
    <suite>                        Can have multiple elifs
...
else:
    <suite>                        Optional
```


The `if` statement

Predict the output of the following:

```
>>> n = 7
>>> while n > 0:
...     if n % 2 == 0:
...         n = n + 3
...     else:
...         n = n – 3
...     print(n)
...
```

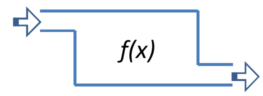
The `if` statement

Predict the output of the following:

```
>>> n = 1
>>> if n > 0:
...     n = n + 1
...     print("A")
... elif n > 1:
...     n = n – 2
...     print("B")
... else:
...     print("C")
...
```


Functions

We've already seen a few!



Defining Functions

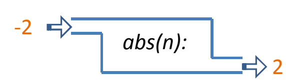
```
def <function name>(<argument list>):
    <suite>
```

The **body** can include a *return statement*, where a function can return a value and stop.

```
def add_three_nums(x, y, z):
    return x + y + z
```


Evaluating User-Defined Functions

A function takes a series of *arguments* and returns some value. Calling a function evaluates to the value it returns.* We can approximate this by taking the argument value and substituting it in the body of the function for the argument name.**




```
def abs( -2 ):
    if -2 > 0:
        return -2
    elif -2 == 0:
        return 0
    else:
        return - -2
```

* If you don't return something, the function returns the None value by default.
** Later in the course we'll learn a more accurate model to account for state.

Functions

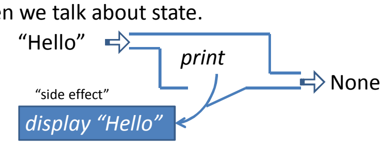
Are these functions the same? ***No!***

```
def add_one(x):
    return x + 1
def add_one(x):
    x + 1
def add_one(x):
    print(x + 1)
```


Non-Pure Functions

A function is called *pure* if it does not have any "side-effects." Otherwise, the function is *non-pure*.

The one example we've seen so far is the print function. We'll see this again later and talk about it more when we talk about state.



Extras

Sometimes we may have too much for one lecture but we would like you to see the material nonetheless. These slides are for material that we might not have time for but that you are responsible for.

Extras – Optional Arguments

We can have optional arguments for our functions:

```
def greet(name="Human"):
     print("Greetings", name)

>>> greet()
Greetings Human
>>> greet("Tom")
Greetings Tom
```


Extras – Multiple Assignment

We can actually assign values to multiple variables at once, which is very useful in some cases:

```
>>> x, y = 6, 7
>>> x
6
>>> y
7

def fib(n):
        """Compute the nth Fibonacci number, for n >= 2."""
        pred, curr = 0, 1
        k = 2
        while k < n:
                pred, curr = curr, pred + curr
                k = k + 1
        return curr
```


Extras – Multiple Return Values

You can actually return multiple values at the same time from a function:

```
from math import sqrt
def square_and_sqrt(x):
      return x ** 2, sqrt(x)
two_sqrd, sqrt_two = square_and_sqrt(2)
```

If you do not put the right number of variables on the left hand side of the assignment, then you can run into an error. If you put only **one** variable on the left, then you'll end up with a **tuple,** which we will talk about at the end of next week.

Extras – Docstrings

Whenever you program, it's always a good idea to make a note explaining your functions in case others read your code (or if you read your code again after a long time). In Python, we do this by putting a "docstring" at the beginning of the function body:

```
def add_one(x):
      """Adds one to x."""
      return x + 1
```

You will see us do this with all code we hand you and encourage you to do the same: it is a good habit for programmers!

Extras – Doctests

In your docstrings, you can even put some examples of how your code is supposed to work. If you do that, you can use the **doctest** module to test to make sure your code behaves as you said:

```
def add_one(x):
        """Adds one to x.

        >>> add_one(5)
        6
        """
        return x + 1
```

If you put this in a file named add_one.py, you can run the following in your terminal:

python3 –m doctest add_one.py

And it will run the code and indicate whether any of your examples don't match what the code actually does in those cases!