Lect 3 – Functions, Iteration

Rob Capra

INLS 490-172

Debugging

- Avoid (major) debugging by:
    - Start small
    - Keep it working / small victories
    - Example from INTPY
- Hints
    - Test boundary conditions
    - Know your error messages
        - 90% = ParseError, TypeError, NameError, ValueError
        - Examples from INTPY

Debugging Error Types

- ParseError – syntax error
  - Ex: missing parens, quotes, commas
  - Try: comment out line, see what errors change
  - Try: narrow the source of the error
- TypeError – incompatible objects
  - Ex: try to add an int and str
  - Often math/expression statements
  - Try: print values

Debugging Error Types

- NameError – use a var before it has a value
    - Often caused by typos, speeling mistaches, mis-remembering var/function name
    - Try: use search feature of editor
- ValueError – pass wrong type parameter to a function

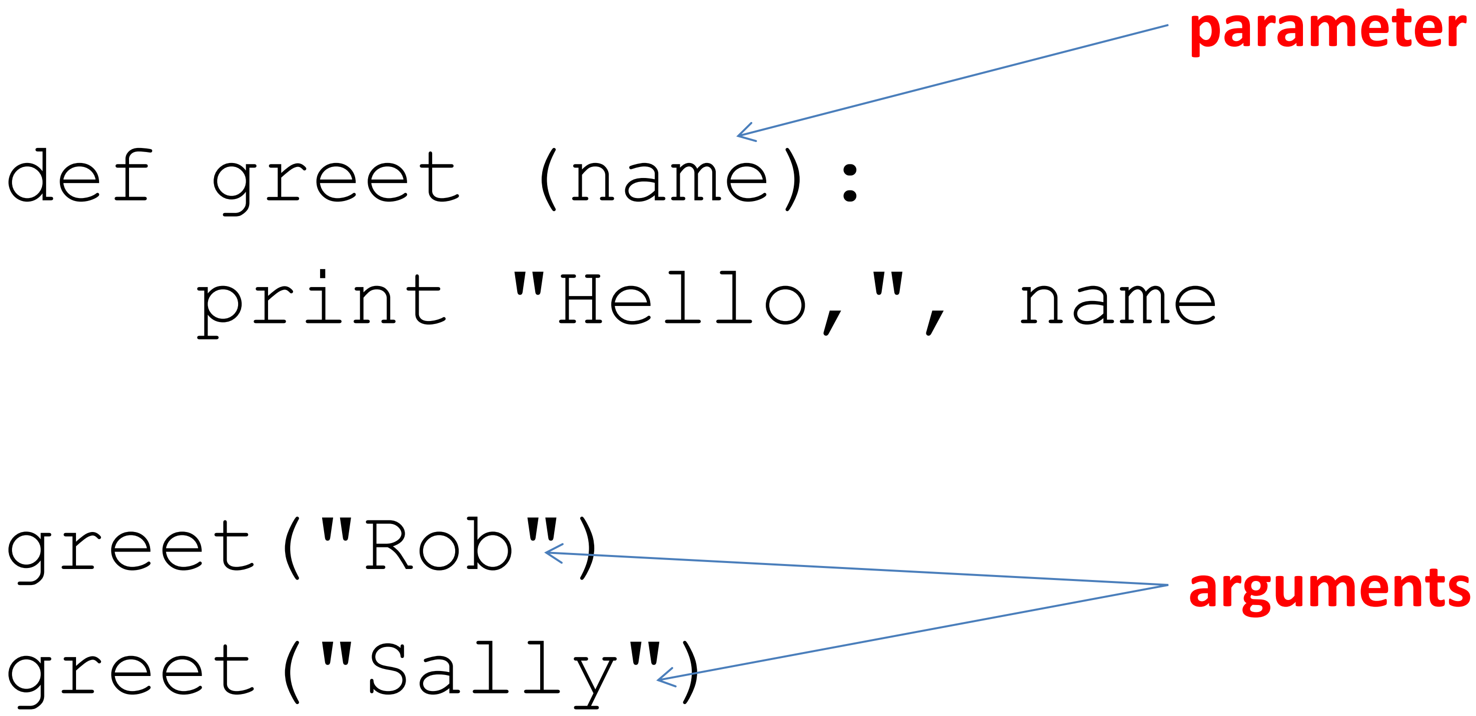
Functions

- Function definition:

```
def name( parameters ):
    statements
```

- Compound statements:
  - A **header** line that ends with a colon
  - A **body** that is indented 4 spaces and has one or more other Python statements

Parameters & Arguments



```
def greet (name):
    print "Hello,", name

greet("Rob")
greet("Sally")
```

parameter

arguments

docstring

```
def greet (name):
    '''Print a greeting to name.'''
    print "Hello,", name
```

In Python shell:

```
>>> greet.__doc__
```

Return Values (fruitful fns)

- A *fruitful* function is one that returns a value
- Use keyword **return**:

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

result = square(4)
print result
```

- Step through flow of execution

Variable Scope / Frames

- Global variables
  - created in the main code, outside of functions
  - (possibly) available anywhere
    - Beware of **shadow** variables
- Local variables
  - created inside a function
  - only available within the scope of the function
- Look carefully at INTPY examples

Local Variables

- y is a local variable that exists only within the scope of the square function.

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

Attempting to access y outside square →ERROR

```
print y
```

Return None

- If you don’t include a return, the function will return a value of **None**.

```
def square(x):
    y = x * x
    print y      # Bad!

answer = square(4)
print answer
```

Global Variables

- Global vars can be accessed from within functions
- But you should not!

```
def square(x):
    y = num * num
    return y

num = 4
answer = square(num)
print answer
```

- First, Python looks for a variable in the local scope of the function.
- If it finds it there, it will use that one.
- If not, then it will look in the global scope.

Local Cannot Change Global

- Assignment statements in the local function cannot change the value of a variable defined outside the function.

```
def square():
    x2 = x1 * x1   # yuck!

x2 = 0
x1 = 3
square()
print x2   # What is printed & why?
```

Shadow Variables

- Shadow variable – a variable in a function with the same name as a global variable.
- Avoid shadow variables

```
def square():
    x2 = x1 * x1     # x2 shadow var

x2 = 0
x1 = 3
square()
print x2    # What is printed & why?
```

Functional Abstraction

- Functions should provide a well-defined output for a given set of inputs.
- How the function “works” – the algorithm – is not known to the outside and could change.

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


num = 4
answer = square(num)
print answer
```

```
def square(x):
    total = 0
    for i in range(x):
        total = total + x
    return total


num = 4
answer = square(num)
print answer
```

Accumulator Pattern

- Initialize an accumulator (e.g. total)
- Loop through a set of items
- Inside the loop, update the accumulator

```
def square(x):
    total = 0
    for i in range(x):
        total = total + x
    return total

num = 4
answer = square(num)
print answer
```

Functions Calling Functions

- Functions can call other functions.

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

def sum_of_squares(x,y,z):
    return square(x)+square(y)+square(z)

print sum_of_squares(1,2,3)
```

Functions Calling Functions

- Functions can call other functions.

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

def sum_of_squares(x,y,z):
    return square(x)+square(y)+square(z)

print sum_of_squares(1,2,3)
```

For loop – Iteration

- For loop processes each item in a list
- In turn, each item is assigned to the loop var
- Then the body of the loop is executed

```
for name in ["Amy", "Brad", "Cathy"]:
   print "Hi,", name, "!!!"
```

For loop – range()

- range(n) – returns a list [0 .. n-1]
- range(n,m) – returns a list [n .. m-1]

```
for i in range (3):
    print i, "squared =", i*i
```

More Iteration – While loop

```
def sumTo(aBound):
    """ Return the sum of 1+2+3 ... n """
    theSum  = 0
    aNumber = 1
    while aNumber <= aBound:
        theSum = theSum + aNumber
        aNumber = aNumber + 1
    return theSum

print(sumTo(4))
print(sumTo(1000))
```

While vs. For

- Use for if you know the number of times you need to iterate
  - Traversing a list of elements
  - Do something 10 times (e.g. can use range)
  - Definite iteration
- Use while if you need to iterate until some condition is met
  - Indefinite iteration

Break… A simple way out

- break can be used to exit a loop

```
def find_brad(namelist):
    for name in namelist:
        print name
        if name == "Brad":
            print "Found Brad!"
            break;

find_brad(["Amy", "Brad", "Cathy"])
```

Break with an infinite loop

```
def type_hello():
    while True:
        line = raw_input ("Please type
hello: ")
        if line == "hello":
            break

type_hello();
```