Organizing your code with functions

Terence Parr
MSDS program
University of San Francisco

See notebook: https://github.com/parrt/msds501/blob/master/notes/functions.ipynb

What's a function?

```
def pi():
    return 3.14159
```

- We're already familiar with functions from mathematics like sin, cos, max, etc…
- A function is just a sequence of operations grouped into a single, named entity that we can invoke to perform a task
- Functions are like mini programs or subprograms that we can build just like full programs
- Just like a book is organized into multiple chapters, programs are best organized into multiple functions; the main program can then just call the appropriate functions

Black boxes

- Think of functions as black boxes that:
  - perform some task
  - possibly taking some input
  - possibly returning output
  - possibly causing side effects
- Don't worry about their guts, just worry about how to call them
- Reduce cognitive load

no args

```
import datetime as dt
dt.date.today()
```

```
datetime.date(2021, 6, 15)
```

one arg

```
import math
math.cos(math.pi)
```

```
-1.0
```

multiple args

```
min(9,4)
```

```
4
```

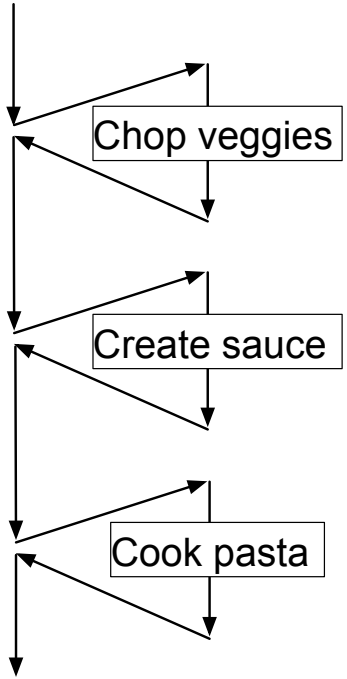
arbitrary number of args

```
print("I have", 2, "cats")
```

```
I have 2 cats
```

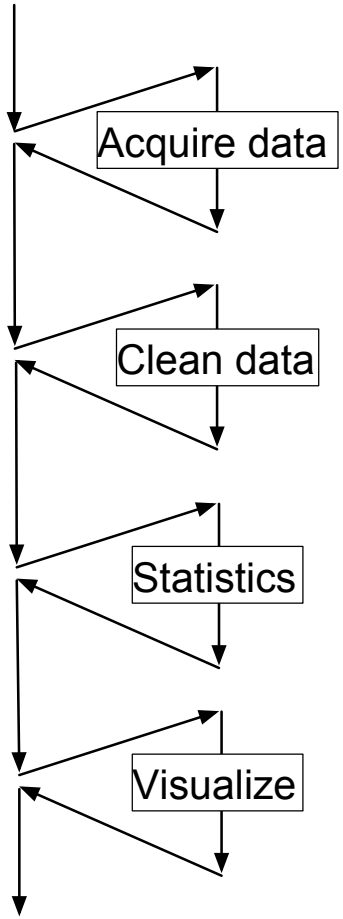
A cooking analogy to functions

- A pasta recipe might have several high level tasks:
  1. Chop veggies
  2. Create sauce
  3. Cook pasta
- As we proceed through the recipe we have to go off and perform the indicated task, come back, and continue to the next task
- We jump from the main path to the subtask and back just like the computer processor in code



Data science example

- The overall program is often a sequence of function calls that perform the subtasks; you might have something like:
  1. Acquire data
  2. Clean data
  3. Compute statistics
  4. Visualize results
- Top-down design: solve overall problem with high-level tasks, then design those subtasks
- Subtasks might be broken into subsubtasks etc…



The motivation to define functions

- Helps organize our programs, which really helps readability
- Fosters code reuse, thus, increasing productivity
- Lets us focus on just the behavior inside the function, which helps reduce what we have to think about at once
- Functions have well-established input and output (arguments and return values), which can make debugging easier and improves reusability

How to plan out a function

- First, identify:
  1. a descriptive function name
  2. the kind of value(s) it operates on (parameter types)
  3. the kind of value(s) it returns (return type)
  4. what the function does and the value(s) it returns
- If we can't specify exactly what goes in and out of the function, there's no hope of determining the processing steps, let alone Python code, to implement that function
- Write some sample function invocations to show what data goes in and what data comes out
- Then try to work out the steps, possibly working from the return value backwards
- Then write the code that implements the steps

Coding a function

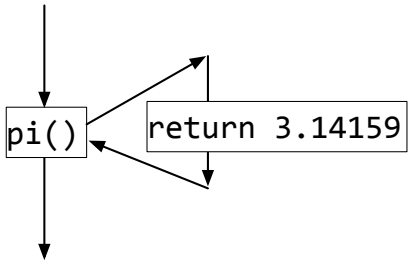
- The code template for a function with no arguments is:

```
def funcname():
    statement 1
    statement 2
    ...
    return expression
```

```
def pi():
    return 3.14159
```

- Recall: we associate statements with a function by indentation
- The function definition does not execute the code inside; it just defines the function for our use

Calling a function




```
def pi():
    return 3.14159
```

- The *definition* of a function is different than *calling* a function
- Calling a function requires the function name and any argument values; here, we don't have any arguments so we can do this:

```
pi()
```

```
3.14159
```

```
pi
```

```
<function __main__.pi()>
```

(We don't need a print statement here to see the value because we are executing inside a notebook)

Functions with side effects

- Some functions don't have return values; e.g., they might update a GUI, alter a database, delete records from a data frame, or simply print
- Such functions have *side effects*
- The **return** statement is omitted if the function does not return a value
- The value of a function w/o a **return** is **None**

```
def hi():
    print('hi')

hi()
```

```
hi
```

Experiment in pythontutor

Return values versus printing

- Functions compute and return values to their callers
- Functions do NOT print anything unless explicitly asked to do so with a **print** statement
- What does this print?

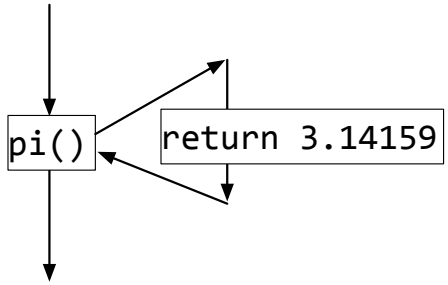
```
def pi():
    print(3.14159) # This is not a return statement!

print(pi())
```

```
3.14159
None
```

Experiment in pythontutor

Saving return values



- Every invocation of function **pi** evaluates to the value 3.14159
- We can save the return value in a variable like **x = pi()**
- Or even use it in an expression like **x = pi() * 4**
- Note: Jupyter notebooks do not print results for assignments (just for expressions)
- The **pi** function *returns* a value but *prints* nothing; e.g., even in a notebook, there is no output if we save the return value

```
[6]: x = pi()

[7]: x

[7]: 3.14159
```

Confusion point!

Functions with multiple return values

- We can return any Python object, not just numbers
- We can also return multiple values

```
def parrt():
    return "parrt", 5707

id, phone = parrt()

print(id, phone)
```

```
parrt 5707
```

Multiple return values are assigned to multiple variables

Functions with arguments

- Here is a function template with *N* arguments

```
def funcname(arg1, arg2, arg3, …, argN):
    statement 1
    statement 2
    ...
    return expression
```

- Function calls look like: `funcname(expr1,expr2,expr3,…,exprN)`
- The order of the arguments matters, matching expr_i to arg_i

Example: summation of numbers in list

- Here's a code snippet to sum the numbers in a list

```
Quantity = [6, 49, 27, 30, 19, 21, 12, 22, 21]
sum = 0
for q in Quantity:
    sum = sum + q
print(sum)
```

```
207
```

- This works, but there's an issue here; any ideas?

The code is not reusable as-is
(must copy/paste/tweak)

Encapsulating in a function; version 1

- By wrapping in a function, we strive for a reusable ”recipe”
- Add the function header, shift the statements to the right and add a return statement:

```
def sum(): # something is wrong here!
    s = 0
    for q in Quantity:
        s = s + q
    return s # this is not a print statement!

Quantity = [6, 49, 27, 30, 19, 21, 12, 22, 21]
s = sum() # call sum and save result
print(s)
```

```
207
```

What’s wrong with this version?

[Experiment in pythontutor]

Encapsulating in a function; version 2

- Functions should focus on the parameters and avoid global variables if possible
- This version now works with any list of numbers, not just **Quantity**:

```
sum([1,2,3])
```

```
6
```

```
def sum(data):
    s = 0
    for q in data:
        s = s + q
    return s # this is not a print statement!

Quantity = [6, 49, 27, 30, 19, 21, 12, 22, 21]
s = sum(Quantity) # call sum with a specific list
print(s)
s = sum(data=Quantity) # explicit arg assignment here
print(s)
```

```
207
207
```

[Experiment in pythontutor]

Example: search function

- Here's a hardcoded non-function search example

```
first=['Xue', 'Mary', 'Robert']      # our given input
target = 'Mary'                      # searching for Mary
index = -1
for i in range(len(first)):          # i is in range [0..n-1]
    if first[i]==target:
        index = i
        break
```

- The problem is that it is restricted to work with a list called **first**

Example: search function version 1

- Wrap in a function header with two arguments

```
def search(x, data):
    index = -1
    for i in range(len(data)):
        if data[i]==x:
            index = i
            break
    print(index)

first=['Xue', 'Mary', 'Robert']
search('Mary', first) # invoke search with 2 parameters
```

What's wrong with this function?

Example: search function version 2

- Wrap in a function header with two arguments

```
def search(x, data):
    for i in range(len(data)):  # i is in range [0..n-1]
        if data[i]==x:
            return i            # found, return current index i
    return -1                   # failure; we didn't find x

print(search('Mary', first))
print(search('Xue', first))
print(search('foo', first))
```

The return statement forces Python to immediately exit the function and return the specified value

Experiment in pythontutor

Restricting data accessed by functions

- Optimally, functions should be purely a function of the data passed to them as parameters---functions should be completely ignorant of any other data
- That is, functions should not access global variables

Good

```
def sum(data):
    s = 0
    for x in data:
        s += x
    return s
```

argument

Bad

```
def sum():
    s = 0
    for x in data:
        s += x
    return s
```

global

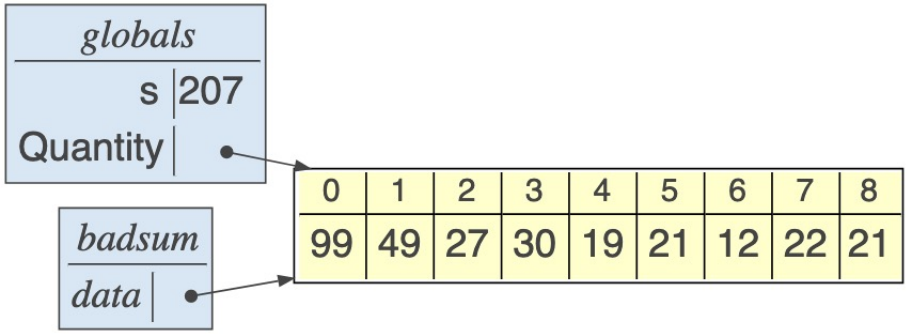
Experiment in pythontutor

Watch out for functions that modify data structure arguments

- A list argument is a reference to the list past in and so modifying the list contents modifies the caller's perspective as well

```
def badsum(data):
    data[0] = 99 # alters global variable as well
    s = 0
    for q in data:
        s = s + q
    return s
```

- (More on this in another lecture)



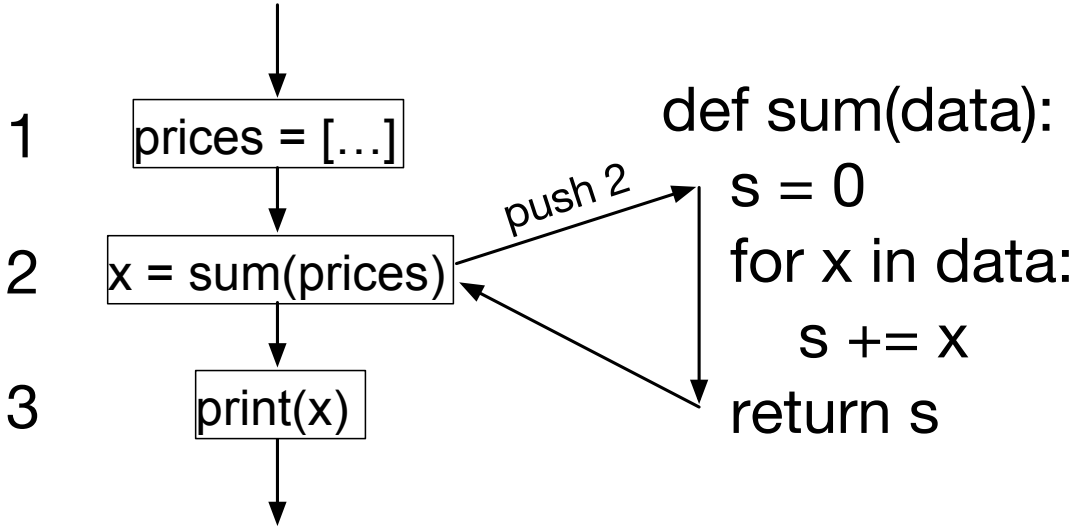
Experiment in pythontutor

Visibility rules

- Main programs cannot see variables and arguments inside functions; just because a main program can call a function, doesn't mean it can see the inner workings
- Functions can technically see global variables but don't do this as a rule; instead, pass the global variables that you need to each function as arguments

How functions return to invocation sites

- Calling a function not only jumps to the function code, it remembers the call site so it can continue where it left off

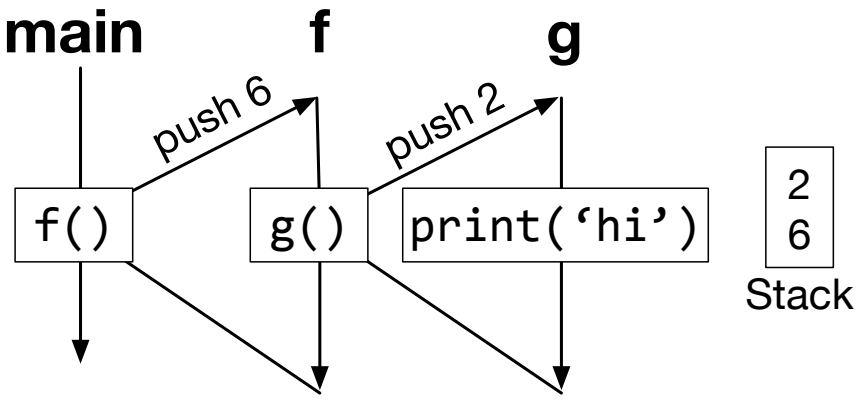


Nested function calls

- Here's a simple program with two functions, where the main program calls **f** and **f** calls **g**

```
1 def f():
2     g()
3 def g():
4     print('hi')

6 f()
```



Experiment in pythontutor

Code organization is important

- Programs quickly become an incomprehensible rat's nest if we are not strict about style and organization
- Here's a general structure for Python programs:

import any libraries
define any constants, simple data values
define any functions
main program body