Functions
parameters, scope
Functions

**function:**

- we can name a batch of statements, to call later.
- a function can accept some expected number of values to be supplied each time it is called
- it must return a single value each time it's called
- it must always return exactly to the place that called it, where program execution resumes
def std_dev(xs):
    avg = sum(xs)/len(xs)
    sig = 0
    for x in xs:
        sig += (x-avg)**2
    sd = sqrt(sig/len(xs))
    return sd

def max3 (a,b,c):
    if a>= b and a >= c:
        return a
    if b>=c:
        return b
    return c

def say_hi ( ) :
    print ("hello, friend!")

The **return statement** allows us to immediately leave a function call with some specific value. → it's like a fancy **break** for function calls
defining functions

syntax example:

```python
def some_name ( any, parameters, go, here) :
    body_of_statements
```

• def keyword, new identifier, parameters list
  – parameters list: list of variable names; indicates how many values to supply, what they will be called

• indented body of statements
  – will return a value

• executing a function definition: python will remember to use this block of code whenever it is called by name. But the definition itself doesn't perform a call!
calling functions

- function calls are expressions
- **evaluates arguments** to values *first (left-to-right)*
- **sends those values** to function definition, and parameters are implicitly assigned those values for this particular call-usage.
  - Each parameter must get exactly one value!
- function's body runs to completion, **returning a value**
- that returned value is the result of evaluating the function call expression, and the statement involving that function call expression can resume
Function Call Examples

```python
biggest = max3 (2,5,5)
print("max is " + str(biggest))

xs = [2,6,5,4,7,4,5]
print("std. dev. of xs is ", std_dev (xs))
```

Each argument is reduced to a value, and only the value is sent.
→ arguments are calculated before the function call

Each function call always returns a single value.
→ we can nest function calls, like nesting any kind of expression
functions calling functions

- order of calls can be as deeply nested as desired.
- each caller pauses, waiting for called function to return a value when finished running
Example: Functions calling Functions

```python
def func_2():
    print("\t...in func_2...")

def func_1():
    print("\t...in func_1...")
    func_2()
    print("\t...back in func_1...")
    return

def main():
    print("...in main...")
    func_1()
    print("...back in main...")
main()
```

NOTE Functions calling functions

The caller is paused, and must wait until the called function is entirely done and returns with a value. Only then does the caller get to resume, using that value.
return statements

we can immediately leave a function-call any time with a `return` statement, indicating what value to return (as an expression):

- `return msg+"\n"
- `return True`
- `return any_expression_here`
- `return # equivalent to return None`

- every function must return a value. (and only returns one value)
- no return statement reached by end of function? None is implicitly returned.
- multiple return statements allowed in one function definition. But each function-call only uses the first one reached. (b/c it leaves)
- want to return multiple things? group them into one complex value (like a list or tuple)
- `None` is a great return value if nothing needs to be returned
To Print or Return?

• Best Practice: don't do any printing inside a function unless printing is the purpose of the function.
  → printed values can't be used for any more calculations.
  → can't call such functions without printing values
  → perhaps you can return the string that might have been printed, and then the caller can choose to print it or not.

If I say "here's five dollars" versus give you five dollars, which is more useful?

• We then can choose what to print at the calling site, instead of being stuck with printing.
  → We've chosen to have "delayed decisions".
POLL – 6A

• return values
Parameters versus Arguments

- **parameters** are the named variables in a function definition. The function is declaring that each function call must ensure each parameter has a value to use in that particular call.
  - parameters are just variables, local to the function

- **arguments** are the actual expressions that are evaluated to values, used as parameters' values in a particular function call.
  - implicitly assigned as params' values for each call
**parameter list things**

**positional parameters:** order and number of parameters dictates how the **positional arguments** are matched.

**default arguments:** expressions linked to particular parameters in function definition. Only used as the value when a function-call doesn't directly provide an argument value (e.g., when no positional or keyword argument is provided).

- tricky detail: these expressions are only evaluated once, when the function is defined! Watch out for aliasing of complex values.
parameter list examples

**positional parameters**: any # of params, comma-separated.

```python
def max3( a, b, c ): ...
def findval( some_list, val_to_find ): ...
def print_menu( ): ...
```

**default arguments**: expressions assigned in parameter list.
- all default arguments must be after any positional parameters
- only used when func call *doesn't* supply an arg for this param.

```python
def sort_vals( vals, ascending = True ): ...
def replace( culprit, vals, max_replacements = 1 ): ...
def normalize( data, low = 0, high = 1 ): ...
def int( x, base = 10 ): ...
```
argument lists

• argument lists are a function call's provided expressions that get fed to the parameters

• each function call must provide zero or more argument expressions: comma-separated, in ()'s.

• each parameter needs to receive exactly one value for the call to begin. Many ways to provide the value:
  – supply arguments positionally
  – use keyword arguments to pair arg. to named parameter
  – rely upon default arguments
many calls to a function

- each function call supplies new arguments.
  - like how each loop-iteration uses next value from given sequence: loop-variables' assignments are implicit each iteration, and parameters' assignments are implicitly part of each call.

```python
def add3(a, b, c):
    return a + b + c

ans1 = add3(2, 4, 6)
ans2 = add3(1, 3, 5)
print(ans1 * ans2)
```

```python
a = 2
b = 4
c = 6
ans1 = a + b + c
a = 1
b = 3
c = 5
ans2 = a + b + c
print(ans1 * ans2)
```

same 'assignments' to a, b, c
POLL – 6B

• calling functions
**positional arguments**: expressions assigned in-order to the function's parameters.

- must be front of args list. Often, the entire args list.
- when matched parameter has a default value, this provided positional one is used (default is ignored for this call).

```python
max3(15, 400, x+17)  # three args provided
std_dev([95, 91, 94, 87])  # one argument
print_menu()  # zero positional args expected or provided
int("234", 10)  # 2nd arg: expected base of 1st arg.
```
default arguments

**default args**: no positional/keyword arg given, so function uses the default argument of its definition.

- triggers use of parameter's default value
- fails when a parameter didn't have a default value

```python
print("hello")  # end defaulted to '\n', sep defaulted to ' '  
int("354")     # base defaulted to 10
```
POLL – 6C

- default arguments
**keyword arguments**: regardless of order, these named arguments only pair with the parameter of that name.

- Must be after all positional arguments.
- Look confusingly similar to default arguments, but:
  - Keyword arguments are in the function *call*
  - Default args are in the function *definition*

```python
print("some","dramatic","args","here", end="!
", sep=".")
int("1101", base=2)
open("example.txt", mode="w")
```
def func_1(x=5, y=10):
    return x * y

def main():
    print (func_1(x=2, y=3))
    print (func_1(y=2, x=12))
    print (func_1(y=25))
    print (func_1(25))

main()
more parameter styles

Advanced parameters styles:

• "varargs": single-starred argument that accepts any # of positional arguments, grouping them into a tuple (assigned to this param's name)
  – def max_many ( * many_vals ) :
  – def record_grades ( grades_list , *entries ) :

• "keyword args": double-starred argument that accepts any # of keyword arguments, storing them into a dictionary (assigned to this param's name)
  – def format ( * args, **kwargs ) :
  – def print ( *args, **kwargs) :
multiple scope levels

- **closest**: local scope: things defined in the current function
  - all parameters of current function
  - all variables assigned anywhere in current function
  - next level of scope: whatever encloses our function (could be another function if we nest our definitions – tricky!)

- **further out**: global scope (all top-level def's in this file: functions and other vars)
  - all things created at no-indentation
  - anything explicitly made global with a `global` statement

- **furthest out**: built-in scope
  - available unless closer definition with same name hides it
  - `print()`, `int()`, etc.
scope definition

each definition is only available in parts of your code.

- **built-in** definitions are always available
- **global** definitions are visible in entire file ("top-level" defn's)
- **local** definitions in a function are only accessible during the call. they only live during the function call, dying when we return from the function.
  - parameters and any variables assigned in the function are all local

- thus multiple functions can re-use convenient names without interfering with each other's definitions
Scope

• The most local version of some identifier is always used when the same name was chosen for a global and local identifier. → This can be convenient or confusing…

• When we define local variables in a function, these *local variables "die"* when we return from (leave) the function. → Parameters are also local variables.
while in a more-local scope (in add3), we can still read global `badness`

badness = 0  # global variable. bad style!
def add3(x,y,z):
    # x, y, z's scope is inside add3. add3's scope is global.
    ans = x+y+z  # ans's scope is local to add3.
    ans += badness  # badness not local, but found in outer (global) scope
    return ans  # x, y, z, and ans all die. A value is returned.

print(add3(2,4,6))  # finds global add3, finds built-in print (further-out scope)
badness = 100  # finds global variable, badness
print(add3(2,4,6))  # finds global add3, finds built-in print (further-out scope)
scope example #2

- clashing local versus global name? local version hides the global version (doesn't use global version)

```python
foo = 5  # global
def add3(x,y,z):
    foo = x+y+z  # local foo; global foo currently hidden.
    return foo
print(foo)  # prints 5
print(add3(1,3,5))  # prints 9
print(foo)  # still prints 5 (add3 call didn't affect global foo)
```
scope example #3

- both have local definitions for `xs` and `ans`.
- but these two `xs`'s never coexist! These functions don't happen to call each other.
- same for `ans` variables.
**scope example #4**

```
def avg(xs):
    ans = sum(xs) / len(xs)
    return ans

def std_dev(xs):
    mean = avg(xs)
    ans = 0
    for x in xs:
        ans += (x-mean)**2
    ans = (ans/len(xs))**0.5
    return ans
```

- even now, the two ans variables are separate.

- what about the two xs?
  - while std_dev is calling avg, there are two distinct variables, each in their own scope.
  - as an "engineered coincidence", they are aliases in this example.

- run in the visualizer.
Mutability across function calls
Mutability and Function Calls

- **The only thing** that ever gets **sent** from function call to function definition **is a value**. (never a variable, expression, etc).
  → a reference-to-a-list may be the passed value!
  → likewise, **only a value is returned**.

- **Mutability** is then the same as always – aliases might exist, and might modify the same spot in memory where others point.
using mutability with functions

• create functions whose purpose is to directly modify lists and other complex values
• argument will be a reference to the value
• keep updating via your parameter.
  – reassigning your parameter loses the aliasing!
• no need to return anything; caller sees the updates directly.
def change_me(xs, i):
    xs[i] = 99

def main():
    vals = [1, 2, 3, 4]
    change_me(vals, 2)
    print(vals)

    # sends a copy to change_me
    data = [1, 2, 3, 4]
    change_me(data[:], 3)
    print(data)

    # reference sent, not xs name
    xs = [1, 2, 3, 4]
    change_me(xs, 0)
    print(xs)

• notice – change_me doesn't return the list, but main() still witnesses the update

• remember – only values are sent to function call, never a variable name. (it's actually an expression, not a variable, and it's reduced to a value before the call is made)

output:
[1, 2, 99, 4]
[1, 2, 3, 4]
[99, 2, 3, 4]
def no_change(xs, i):
    vals = xs[:]
    vals[i] = 99

def doesnt_empty_list(xs):
    xs = []

def relocate_forever(xs, ys):
    while len(xs) > 0:
        ys.append(xs.pop(0))
    vals = [1, 2, 3, 4]
    relocate_forever(vals, vals)

• modifies a copy; vals dies upon returning, we lose the modified copy.
   → don't work on a lost copy

• local variable xs is reassigned to empty list (breaking alias with caller's version). Didn't update, lose our copy when xs dies.
   → always update to preserve changes

• relocate function assumed id(xs) ≠ id(ys). oops! Probably caller's fault, but still a pitfall to avoid.
   → be mindful of your assumptions
Practice Problem

Write a function, \texttt{stagger_case}, that uppercases every other character in a string (starting with the first character).

→ what should the return type be?
→ how does mutability affect your solution?
Practice Problem

Write a function, **flatten**, that takes a two-dimensional list of values and returns a one-dimensional list of values, all in the (row-major) order from the original list.

→ what should the return type be?
→ how does mutability affect your solution?
Globals
globals – a Bad Idea

• globals are variables defined at global scope (e.g., 'top-level', no indentation assignments)

• using one top-level variable throughout many functions is a bad idea.
  – our assumptions about when and in what order it will be modified are very hard to predict/verify/record/enforce

• alternative:
  – instead of a global variable, add more parameters to any function that needs to use the shared information
  – rely upon aliasing to share any updates
  – if shared data is a simple value (e.g., an int), you might need to return the updated value from the function (perhaps even tupling up with other things that need returning).
Getting Rid of Globals

Transform globals into parameters. Share updates via aliasing or explicitly returning.

Before

```python
simple = 5  # global 😞
complex = [1,2,3]  # global 😞
def useThem():
    global simple, complex
    simple += 1
    complex[0] = 99
def main():
    global simple, complex
    simple = 5
    complex = [4,5,6]
    useThem()
    print(simple, complex)
main()
```

After

```python
def useThem(s, c):
    # get as params😊
    s += 1
    c[0] = 99
    return s
def main():
    simple = 5  # local 😊
    complex = [4,5,6]  # local 😊
    simple = useThem(simple, complex)
    print(simple, complex)
main()
```
Extra Materials – Stars and Dictionaries in our parameter lists

It made more sense to put this material with our functions slides than delaying them after we've covered dictionaries.

But we will come back to these once we've covered dictionaries.
vararg parameters

Allows varied number of arguments; all arguments will be packed into a sequence of the given name.

– The * designates that the formal parameter is a sequence (tuple) of whatever remaining positional parameters are passed to the function.

Syntax:
```python
def function_name( *sequence_name ):
    < function body >
```
def biggest( * xs ):
    if len(xs)==0:
        return None
    big = xs[0]
    for x in xs:
        if big < x:
            big = x
    return big

def main():
    print (biggest(1,3,2))
    print (biggest(5))
    print (biggest(4,2,6,1,3,5))
    x = 5
    y = 10
    print (biggest(x,y))
    print (biggest())

main()
def f1 (*xs):
    print(type(xs))
    for x in xs:
        print(x)

def f2 (**keyparams):
    print(type(keyparams))
    for k in keyparams:
        print(k, keyparams[k])

def main():
    f1(1, 2, 3)
    f1(1)
    f2(a=1, b=2, c=3)
    f2(x=24, y=25)
**Single Star:** arbitrary number of positional arguments are accepted by this function, packed into a tuple and given the supplied name.

**Double Star:** collects all supplied keyword arguments into a dictionary of this supplied parameter name.

→ keys: keyword identifiers (as strings).
→ values: supplied values.

→ dictionaries are a special way to group values together; we will learn them soon!
Call-Site Stars

• We can also use stars at call sites.

When a function **requires multiple arguments**, we can unpack a sequence of arguments from a sequence with one star.

```python
def fiveargs(a, b, c, d, e):
    return a + b - c * d / e
```

```python
def func_abc(a, b, c):
    return a + b - c
```

```python
d = {"a":1, "b":2, "c":3}
def_abc(**d)
```

When a function **requires multiple keyword arguments**, we can create a dictionary with them all, and unpack it using **.

```python
def fiveargs(*xs):
    return fiveargs(*xs)
```

```python
d = {"a":1, "b":2, "c":3}
def_abc(**d)
```
def f1 ( *xs ):  
    for x in xs: print(x)

def f2 ( **kps ):  
    for k in kps: print(kps[k])

def f3 (a,b,c):  
    print(a,b,c)

def main():  
    f1(1,2,3) 
    f2(a=1,b=2,c=3) 
    nums = [1,2,3] 
    f3(*nums) 
    d = {"a":1,"b":2,"c":3} 
    f3(**d)

main()