Math 260: Python programming in math

Even more python features:
default args. and error handling

default arguments

- Often, an argument is usually a known 'default' value
- We don't want the user to have to specify the default!
- You set a default value in the function definition:

```
def bisection(func, a, b, tol=1e-8):
    ...

x = bisection(func, 0, 1) # uses tol = 1e-8
x = bisection(func, 0, 1, 1e-2) #uses tol = 1e-2
```

- As a rule, default arguments must be *last* in the definition:

```
def f(a, b=1, c=2):  # ok

def g(a, b=1, c):  # error!
```

More generally...

- There are two types of arguments:

  positional arguments: the normal kind. Assigned by order in the call.
- **keyword arguments**: Assigned by name; order does not matter.

Note that args. with defaults are positional, e.g.

`x = bisection(func,0,1,1e-2)`

Aside: default arguments

We can use `var=value` in the *input* to indicate a keyword arg:

```
def func(a, b):
    return a, b

print(func(1, 2)) # 1 2
print(func(b=2, a=1)) # also 1 2
```

These variables are set by name, not position.

- Consider using parameters with defaults as keyword args for clarity!

```
def bisection(func, a, b, tol=1e-8, max_iter=10):
    ...

x = bisection(func, 0, 1, 1e-2, 50)
x = bisection(func, 0, 1, max_iter=50, tol=1e-2) # works!
x = bisection(func, 0, 1, max_iter=50) # works!
```

- (Best practice: try to keep everything ordered anyway)
- Further aside: you can have un-ordered keyword args in function definitions also - see the `kwargs` syntax for details.

default arguments

There's one snag, however - **mutable** defaults may not do what you expect.

```
def glue(elements, base=[]):
    """ adds elements to the end of base
        and returns the reference """
    base.extend(elements)
    return base

a = [3,4]
b = glue(a) # b is [3,4] (a new copy)
b[0] = 7
oops = glue(a) # does NOT do the same as b
print(oops) # oops = [7,4,3,4]
```

- mutable defaults are set **once** and then stick around
- For all subsequent calls, the existing data is used

1) The first call is `glue(a,[])`; then `base = [3,4]` and `b = base`
2) The second `glue(a)` uses the `base` from (1)

Solution: use an immutable type (e.g. a tuple) or restructure...

Aside: default arguments

Aside: you can use this to 'save' info between function calls:

```
def func(a, record=[])
    result = a
    #... do work ...
    record.append(a)
    return result, record
```

```
a = func(1)[0]
b = func(2)[0]
c, record = func(3)
# record is [1,2,3]
```

Each time func is called, the return (a) is added to record

Error handling

When python encounters **errors** (also called '**exceptions**'), the program stops.

- But often, error are not 'fatal' to the program - we want it to notice the error, and then recover and keep going.
- An `if` will not do - we need a special environment (`try`)

  1) The program 'tries' to execute what is in the `try` block

  2) if an error occurs, it skips directly to the `except` block

  - If the error type matches an `except`, it **catches** the error, executes that clause, then the program continues.

  - If there is no match, the error just occurs for real.

```
while not done:
    try:
        x = input('enter an int: ')
        y = int(x)
        done = true
    except ValueError:
        print('Wrong! Try again.')
    except KeyboardInterrupt:
        print('UNACCEPTABLE.')
```

Without `try`, the command `int('a')` just gives an error:

```
In [2]: int('a')
Traceback (most recent call last):

  File "<ipython-input-2-233884bacd4e>", line 1, in <module>
    int('a')

ValueError: invalid literal for int() with base 10: 'a'
```

Error handling

You can cause an error to occur using `raise`:

- This 'raises' an error of that type as if it had occurred for real
- `raise(str)` defines the associated error message `str`
- You can define your own Exception types (classes) - more on this soon

```
def bisection(f, a, b):
    if sign(f(a))*sign(f(b)) > 0:
        raise(ValueError('Endpoints must bracket a root.'))
    #...
```

Clean-up: after an error occurs, you may want to have some 'clean-up' code:

- The `finally` clause always executes if an error occurred in the try block

```
try:
    f = open('myfile.txt','r')
except SomeFileError:
    # ....
except (OtherErrors, MoreErrors):
    # ....
finally:
    print('Unable to open file!')
    return
#... continue the function....
```

What happens when an error occurs?

- Reminder: when a function is called, it is 'put on the stack'. When a line of code is executed, it lives on top of a stack of calling functions.

```
def inner(a, k);
    y = a[k] #**
    return y
def outer(a, k):
    return inner(a)
def test():
    a = [1,2,3]
    outer(a, 15)
```

The stack at line (**):

- `y = a[k]`
- ...called by `inner(a,k)`
- ...called by `outer(a,k)`
- ...called by `test()`

- Errors propagate 'up the stack'. An error can be caught by any of the functions. The program **fails if nothing catches the error**.

```
def outer(a, k):
    try:
        y = inner(a)
    except IndexError:
        #...
def inner(a, k);
    return a[k]
```

Using `outer([1,2,3],15])` ...

- `a[k]` raises `IndexError`
- `inner` fails with an error and leaves the stack
- Now the error propagates to the try block in `outer` (caught!)

A brief style note

Style: do not overuse try/except blocks:

- If you can handle the logic *without* an error, do so
- Common places to use it: where functions raise exceptions the user might want to catch (e.g. File I/O)
- Use sensible error names (if you have a unique error, name your own)

Not a good approach:

```
a = [1,2,3]
k=input('index? ')
try:
    v = a[k]
except IndexError:
    print('try again!')
#... etc...
```

More clear:

```
a = [1,2,3]
k=input('index? ')
if k < 0 or k >= len(a):
    print('try again!')
#... etc...
```