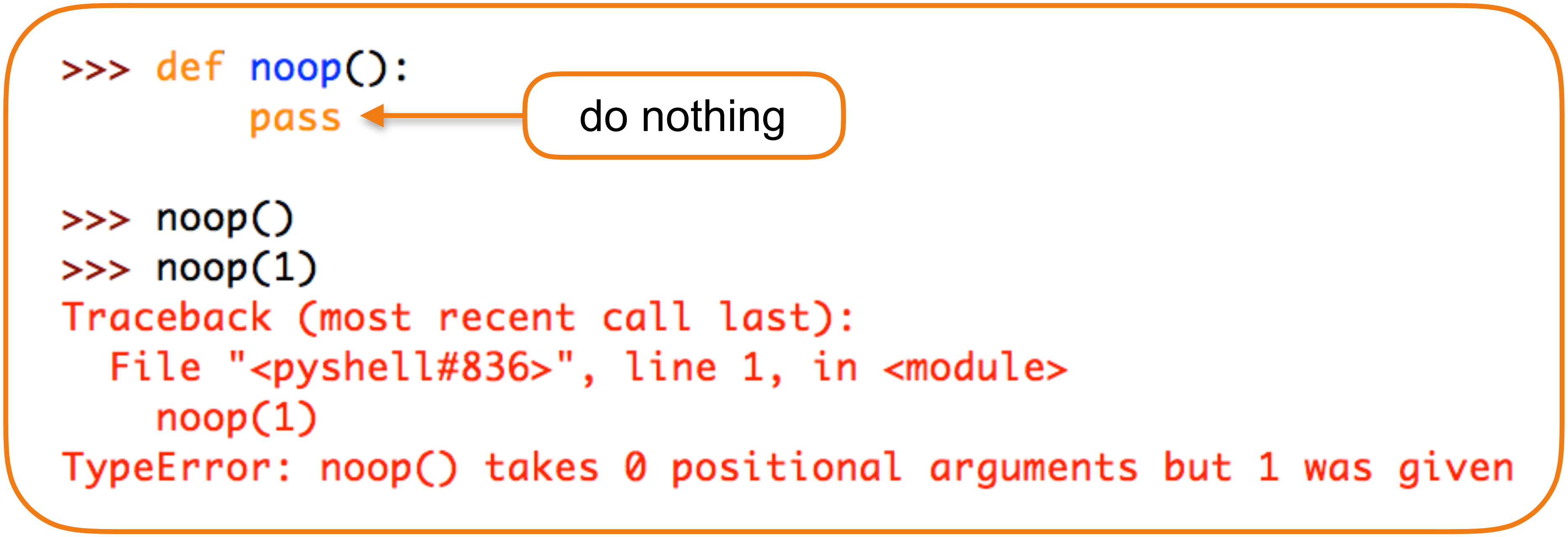
Day 3 Agenda

- functions
- exceptions
- decorators
- command line arguments
- modules
- developer modules
  - os, sys, subprocess
  - shutil, glob
- StringIO
- regular expressions
- OO programming: classes/class decorators

functions

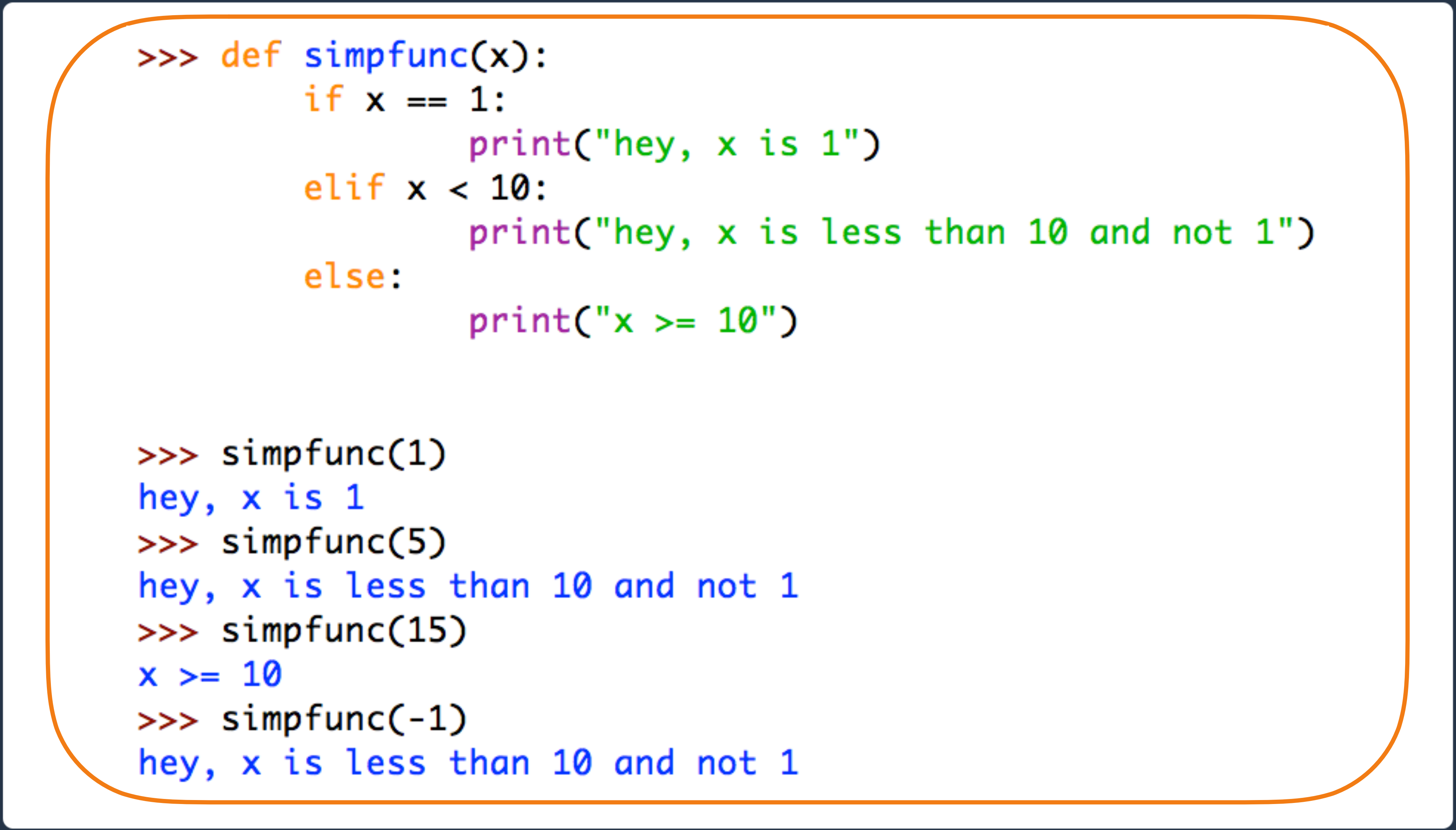
- `def` introduces a function, followed by function name, parenthesized list of args and then a colon
- body of function is indented



```
>>> def noop():
        pass

>>> noop()
>>> noop(1)
Traceback (most recent call last):
  File "<pyshell#836>", line 1, in <module>
    noop(1)
TypeError: noop() takes 0 positional arguments but 1 was given
```

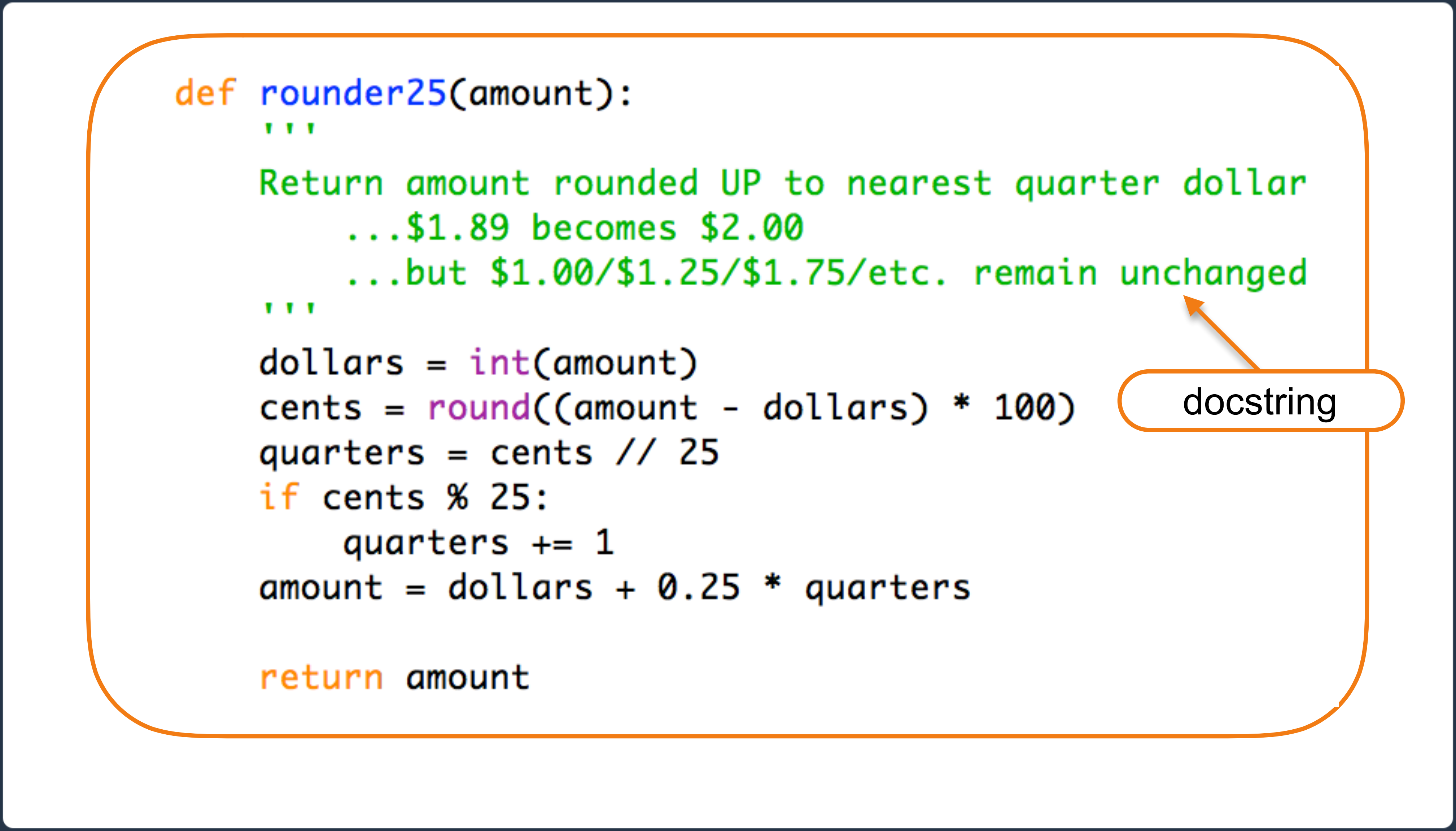
do nothing

functions (cont'd)



```
>>> def simpfunc(x):
        if x == 1:
                print("hey, x is 1")
        elif x < 10:
                print("hey, x is less than 10 and not 1")
        else:
                print("x >= 10")


>>> simpfunc(1)
hey, x is 1
>>> simpfunc(5)
hey, x is less than 10 and not 1
>>> simpfunc(15)
x >= 10
>>> simpfunc(-1)
hey, x is less than 10 and not 1
```

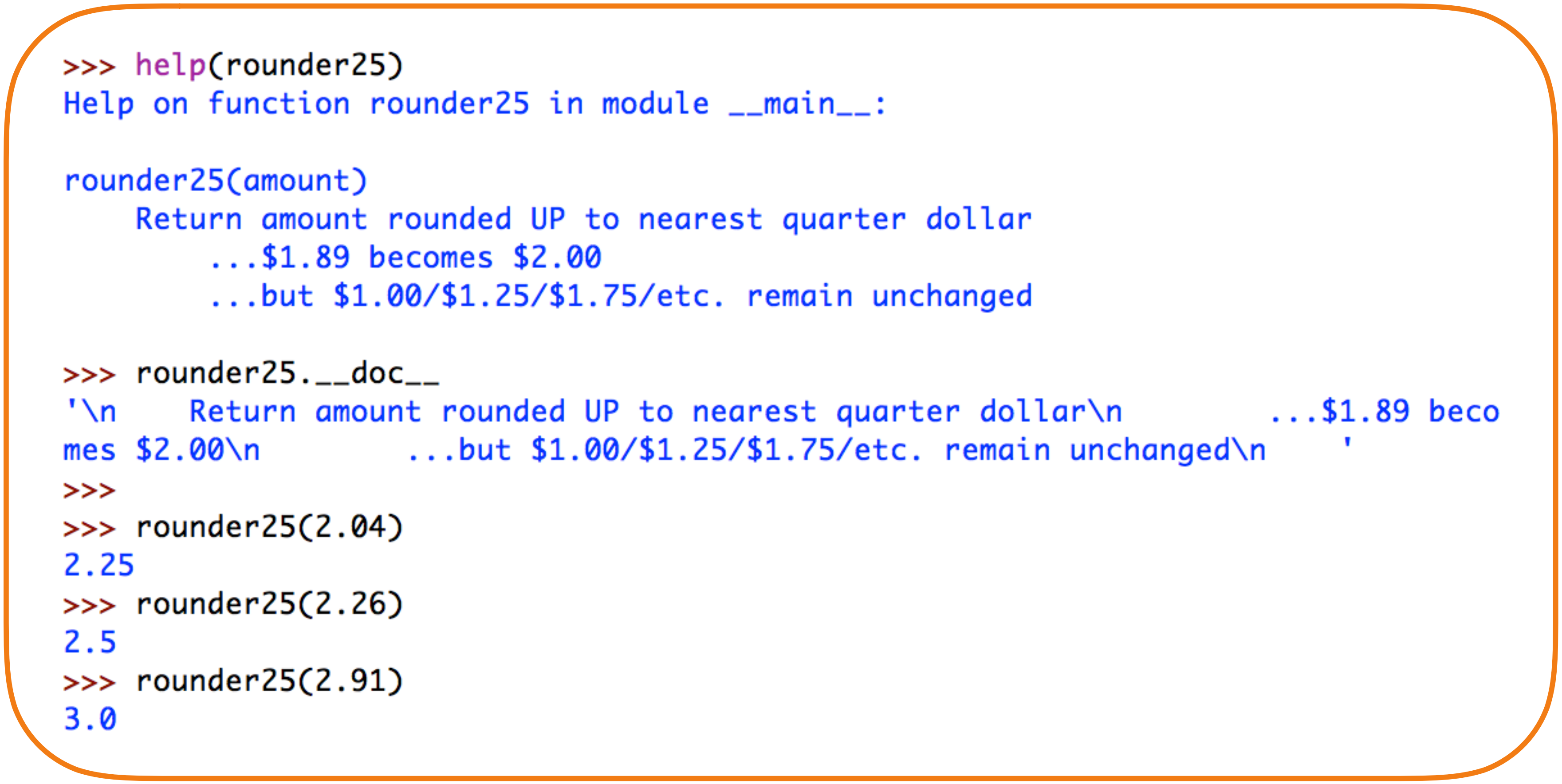
functions (cont'd)



```
def rounder25(amount):
    '''
    Return amount rounded UP to nearest quarter dollar
        ...$1.89 becomes $2.00
        ...but $1.00/$1.25/$1.75/etc. remain unchanged
    '''
    dollars = int(amount)
    cents = round((amount - dollars) * 100)
    quarters = cents // 25
    if cents % 25:
        quarters += 1
    amount = dollars + 0.25 * quarters

    return amount
```

functions (cont'd)

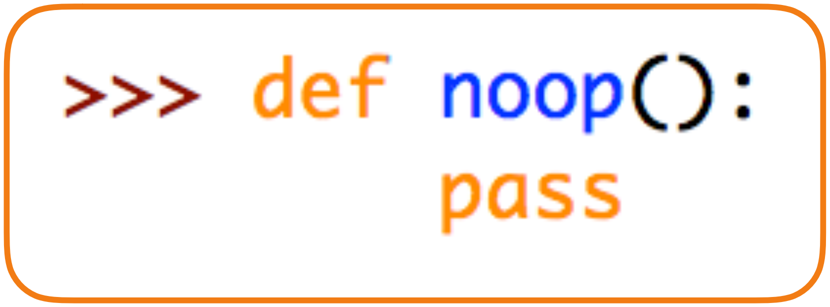
- `help(func)` prints out formatted docstring
- `func.__doc__` prints out raw docstring



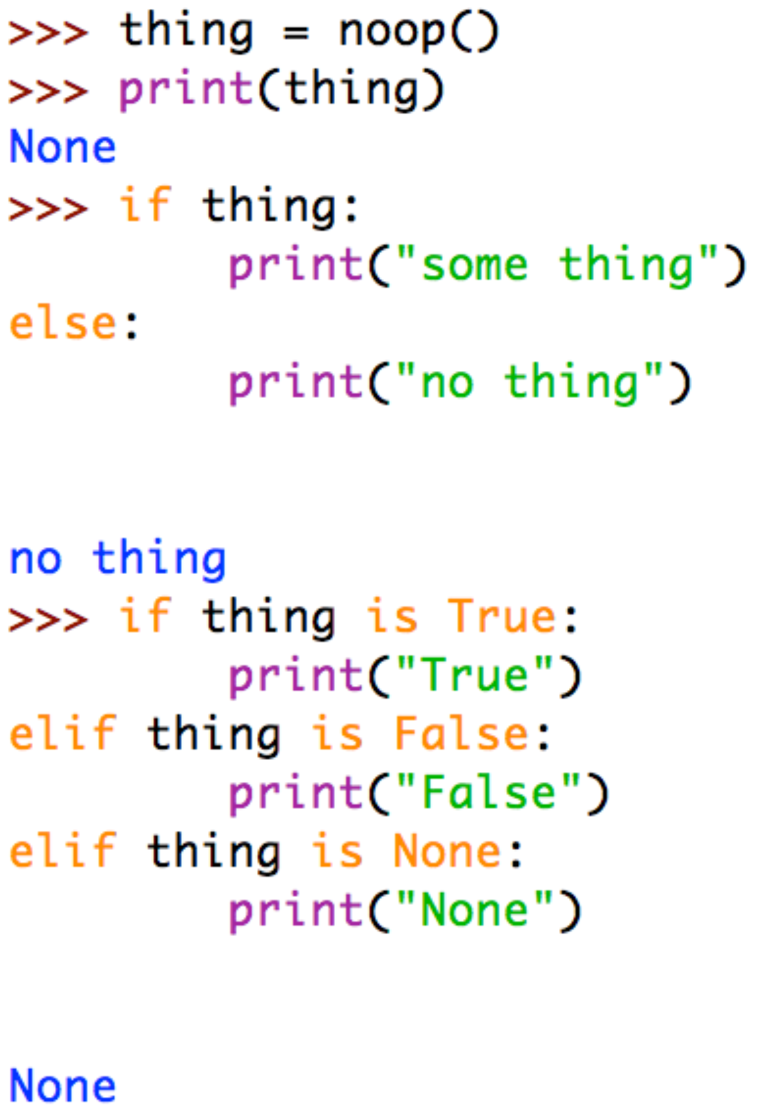
```
>>> help(rounder25)
Help on function rounder25 in module __main__:

rounder25(amount)
    Return amount rounded UP to nearest quarter dollar
        ...$1.89 becomes $2.00
        ...but $1.00/$1.25/$1.75/etc. remain unchanged

>>> rounder25.__doc__
'\n    Return amount rounded UP to nearest quarter dollar\n        ...$1.89 beco
mes $2.00\n         ...but $1.00/$1.25/$1.75/etc. remain unchanged\n    '
>>>
>>> rounder25(2.04)
2.25
>>> rounder25(2.26)
2.5
>>> rounder25(2.91)
3.0
```

functions (cont'd)

- if function doesn't call return explicitly, the special value `None` is returned
- `None` is like `NULL` in other languages
- not the same as `False`



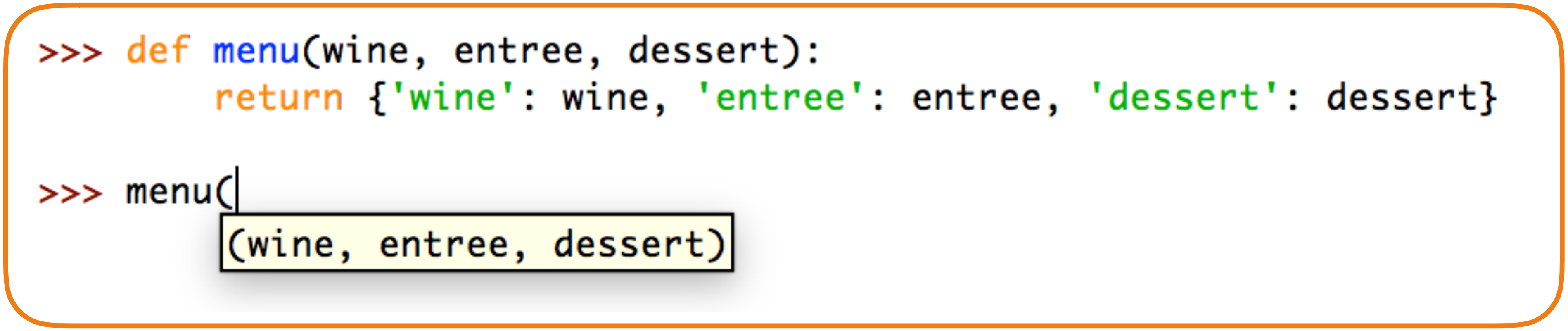
```
>>> def noop():
        pass
```



```
>>> thing = noop()
>>> print(thing)
None
>>> if thing:
        print("some thing")
else:
        print("no thing")


no thing
>>> if thing is True:
        print("True")
elif thing is False:
        print("False")
elif thing is None:
        print("None")


None
```

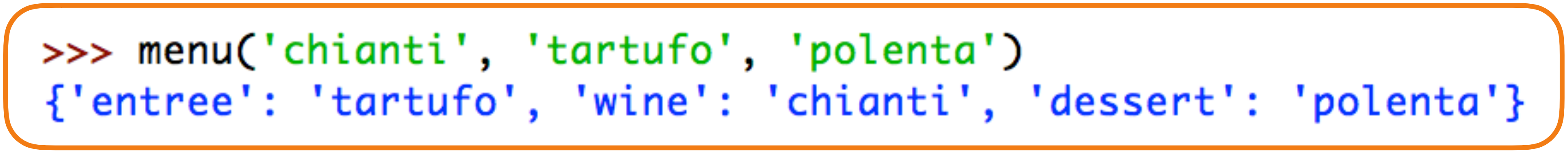
functions: positional arguments

- arguments are passed to functions in order written
- downside: you must remember meaning of each position



```
>>> def menu(wine, entree, dessert):
        return {'wine': wine, 'entree': entree, 'dessert': dessert}

>>> menu(
         (wine, entree, dessert)
```

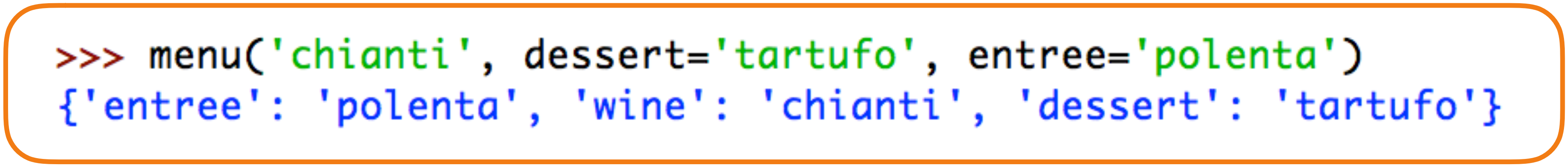
- outside an IDE, it can be difficult to remember
- if you pass args in wrong order, bad things can happen!



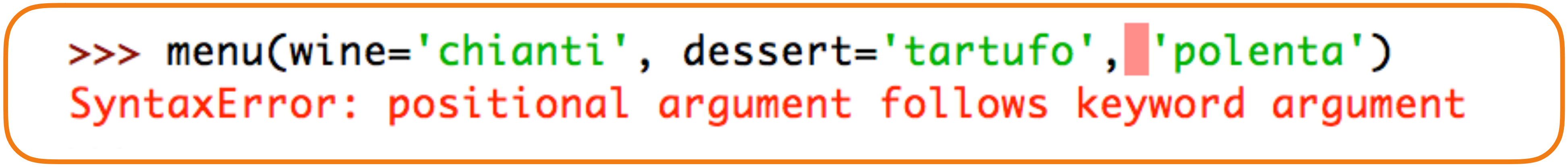
```
>>> menu('chianti', 'tartufo', 'polenta')
{'entree': 'tartufo', 'wine': 'chianti', 'dessert': 'polenta'}
```

functions: keyword arguments

- you may specify arguments by name, in any order
- once you specify a keyword argument, all arguments following it must be keyword arguments

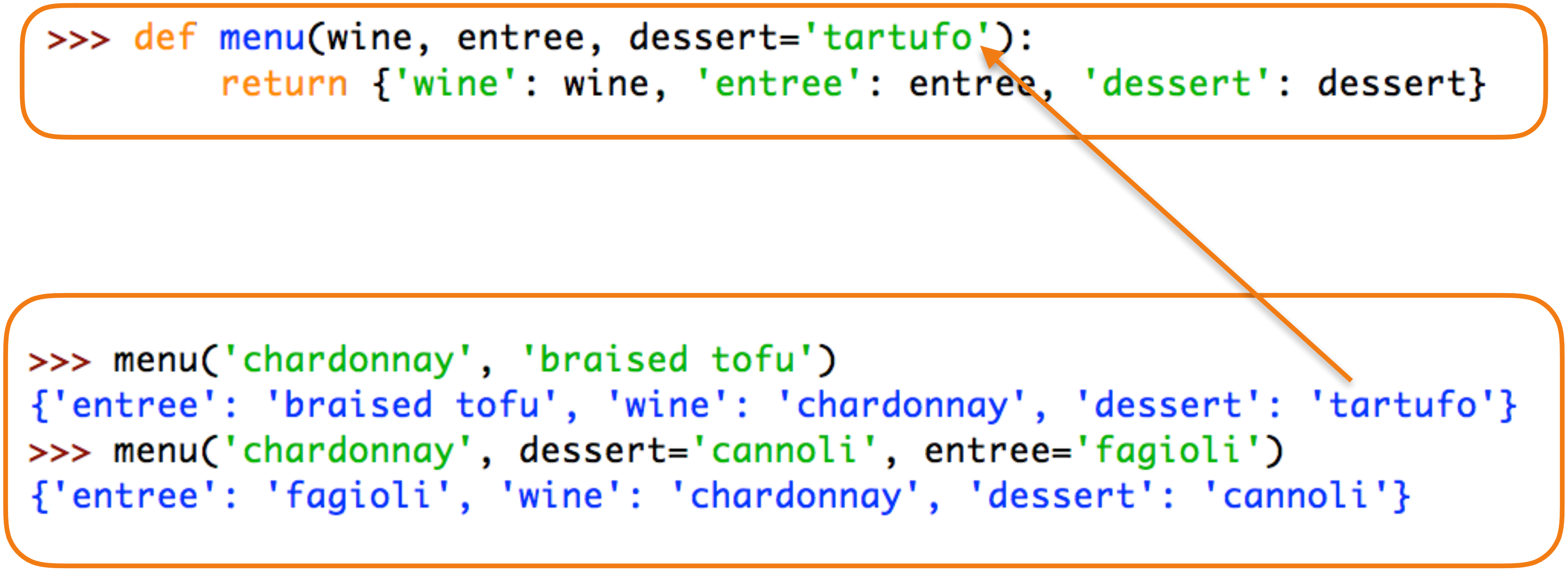


```
>>> menu('chianti', dessert='tartufo', entree='polenta')
{'entree': 'polenta', 'wine': 'chianti', 'dessert': 'tartufo'}
```



```
>>> menu(wine='chianti', dessert='tartufo', 'polenta')
SyntaxError: positional argument follows keyword argument
```

functions: default arguments



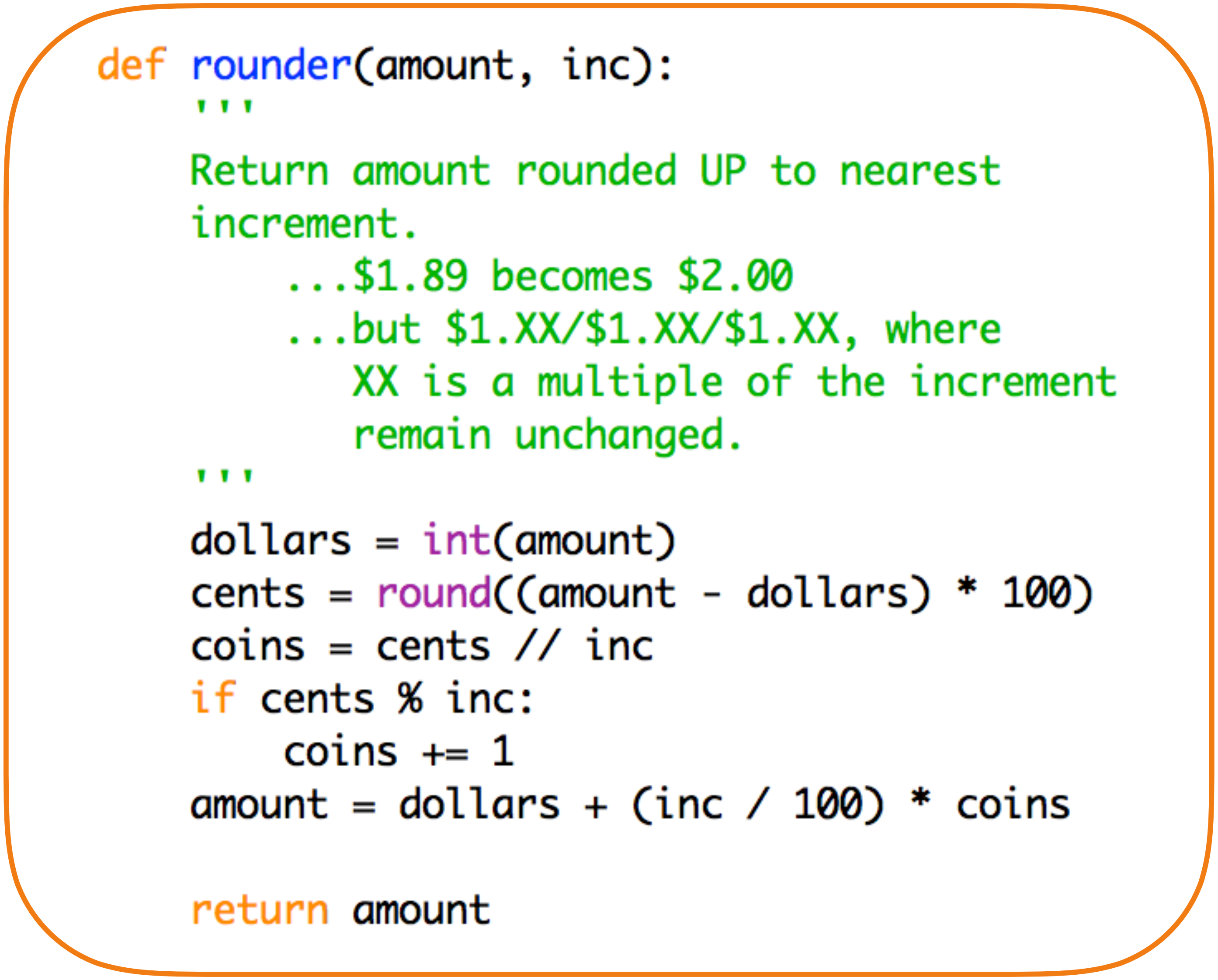
```
>>> def menu(wine, entree, dessert='tartufo'):
        return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

```
>>> menu('chardonnay', 'braised tofu')
{'entree': 'braised tofu', 'wine': 'chardonnay', 'dessert': 'tartufo'}
>>> menu('chardonnay', dessert='cannoli', entree='fagioli')
{'entree': 'fagioli', 'wine': 'chardonnay', 'dessert': 'cannoli'}
```

lab: functions

1. modify the `rounder25()` function to take an additional argument which specifies the increment to round up to (e.g., `rounder(1.37, 10)` would round $1.37 up to the next dime, or $1.40)
2. write a function `calculate` which is passed two operands and an operator and returns the calculated result, e.g. `calculate(2, 4, '+')` would return 6

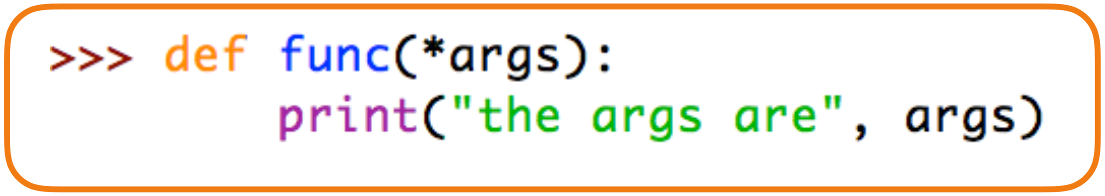
lab: functions (solution)



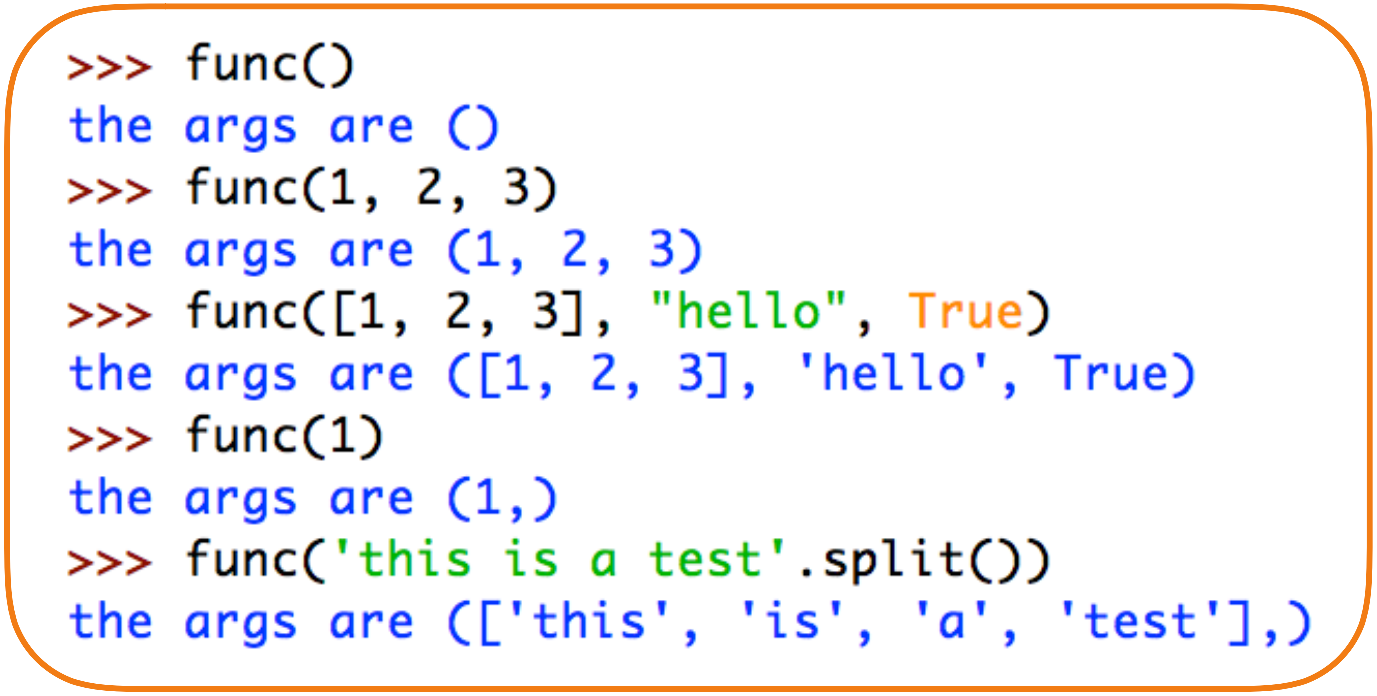
```python
def rounder(amount, inc):
    '''
    Return amount rounded UP to nearest
    increment.
        ...$1.89 becomes $2.00
        ...but $1.XX/$1.XX/$1.XX, where
           XX is a multiple of the increment
           remain unchanged.
    '''
    dollars = int(amount)
    cents = round((amount - dollars) * 100)
    coins = cents // inc
    if cents % inc:
        coins += 1
    amount = dollars + (inc / 100) * coins

    return amount
```

variable positional arguments

- sometimes we want to a function which takes a variable number of arguments (e.g., builtin `print()` function)



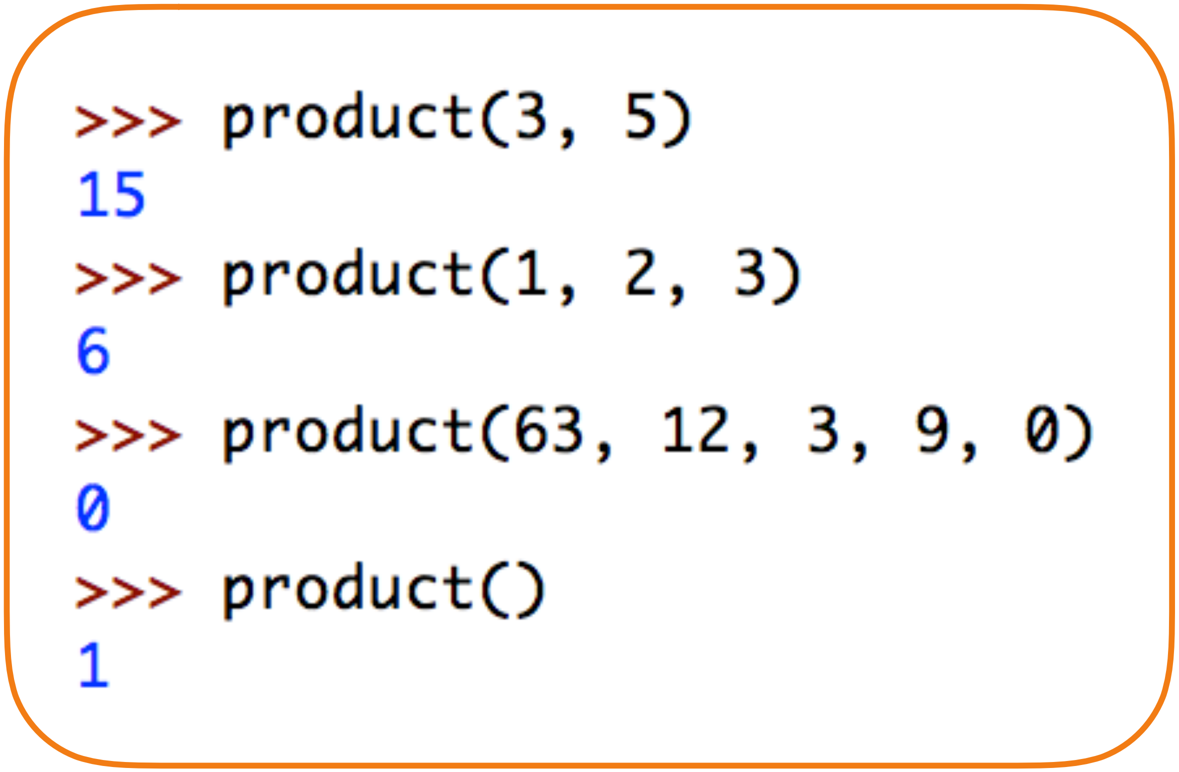
```
>>> def func(*args):
        print("the args are", args)
```



```
>>> func()
the args are ()
>>> func(1, 2, 3)
the args are (1, 2, 3)
>>> func([1, 2, 3], "hello", True)
the args are ([1, 2, 3], 'hello', True)
>>> func(1)
the args are (1,)
>>> func('this is a test'.split())
the args are (['this', 'is', 'a', 'test'],)
```

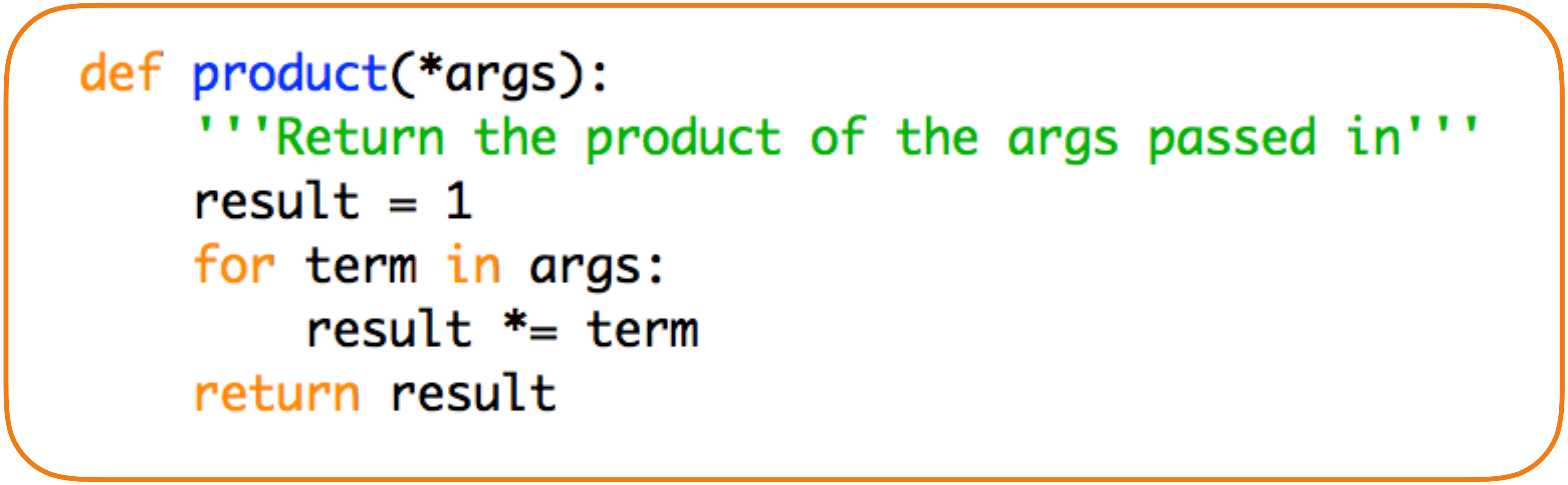
Lab: variable positional arguments

- write a function called `product` which accepts a variable number of arguments and returns the product of all of its args. With no args, `product()` should return 1



```
>>> product(3, 5)
15
>>> product(1, 2, 3)
6
>>> product(63, 12, 3, 9, 0)
0
>>> product()
1
```

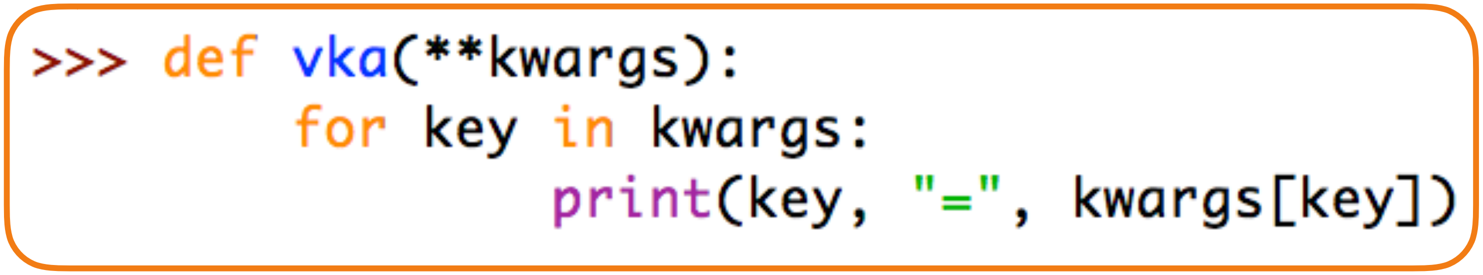
Lab: variable positional arguments (solution)



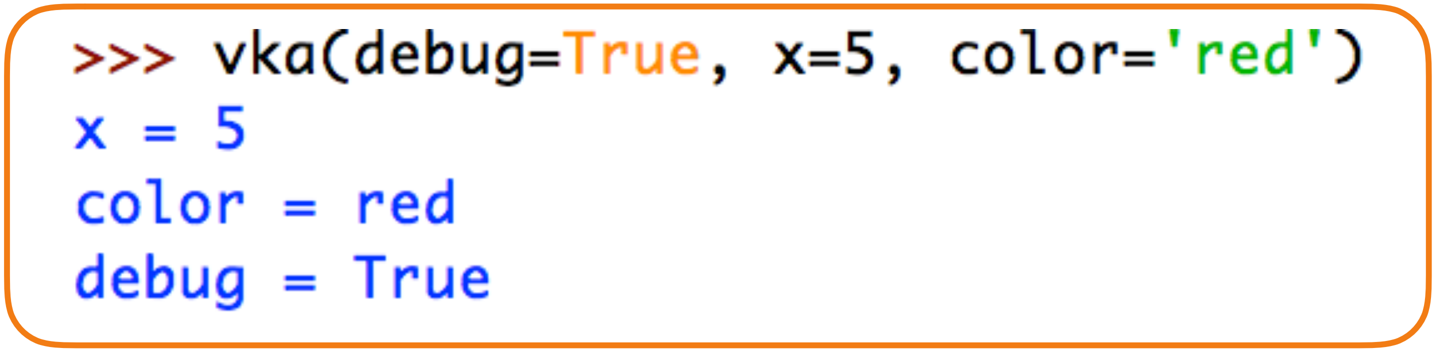
```python
def product(*args):
    '''Return the product of the args passed in'''
    result = 1
    for term in args:
        result *= term
    return result
```

variable keyword arguments

- what if a function needs a bunch of configuration options, having default values which typically aren't overridden?
  - one way to do this would be to have the function accept a dict in which these value(s) can be specified
- better way is to use variable keywords arguments



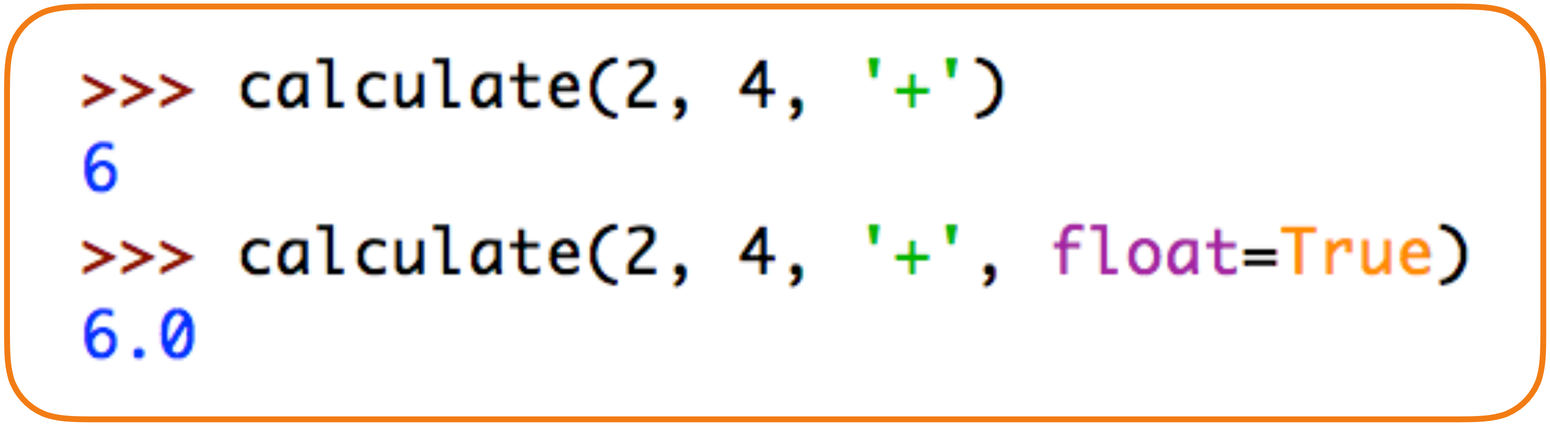
```
>>> def vka(**kwargs):
        for key in kwargs:
            print(key, "=", kwargs[key])
```



```
>>> vka(debug=True, x=5, color='red')
x = 5
color = red
debug = True
```

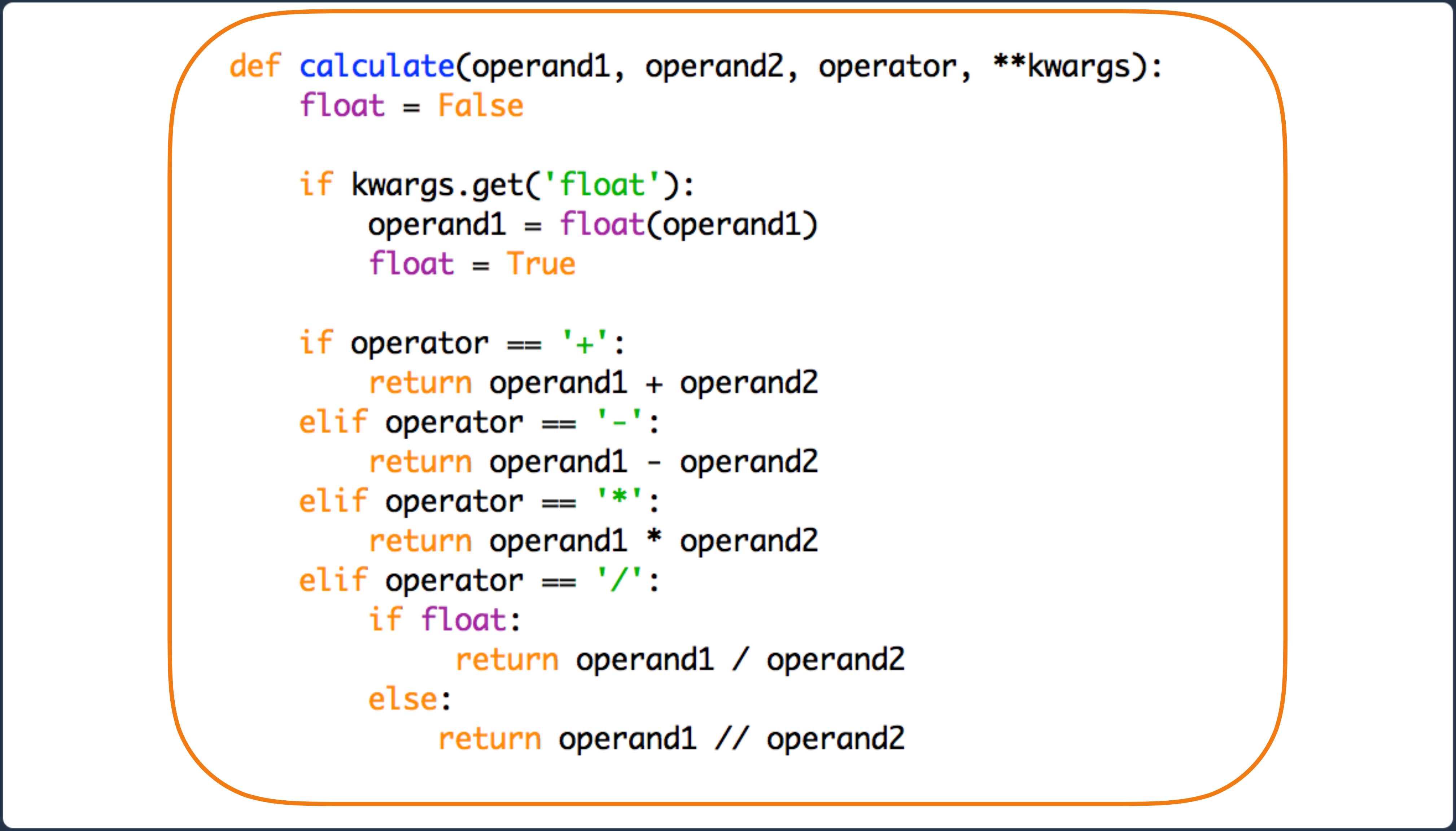
Lab: variable keyword arguments

- modify your `calculate` function by adding variable keywords arguments to it and checking whether `float = True`, and if so, the calculation should be done as floating point, rather than integer (of course this could be done with a default argument value, but don't do that)



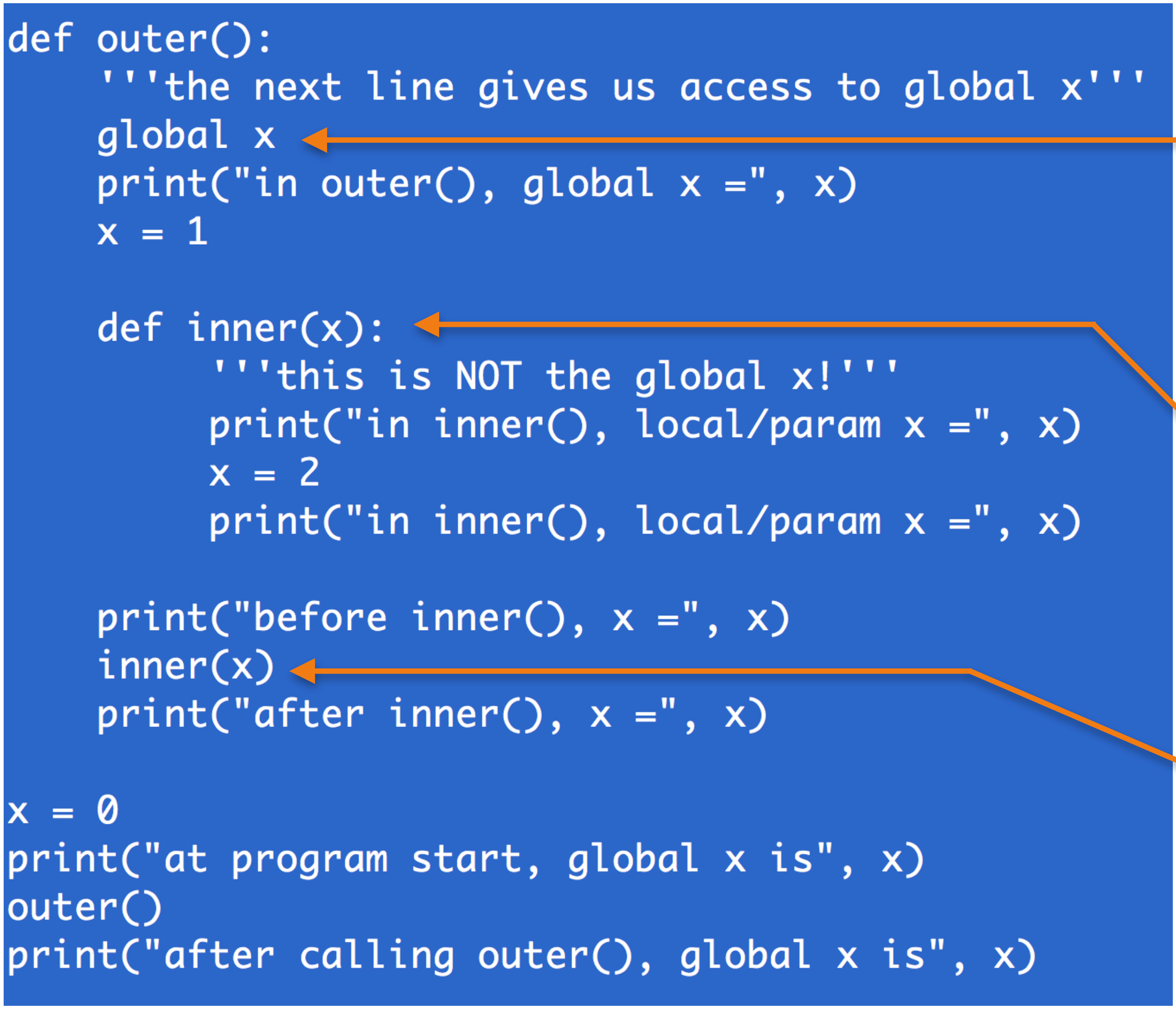
```
>>> calculate(2, 4, '+')
6
>>> calculate(2, 4, '+', float=True)
6.0
```

Lab: variable keyword arguments (solution)



```python
def calculate(operand1, operand2, operator, **kwargs):
    float = False

    if kwargs.get('float'):
        operand1 = float(operand1)
        float = True

    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    elif operator == '/':
        if float:
             return operand1 / operand2
        else:
            return operand1 // operand2
```

functions: scope/global



```
def outer():
    '''the next line gives us access to global x'''
    global x
    print("in outer(), global x =", x)
    x = 1

    def inner(x):
         '''this is NOT the global x!'''
         print("in inner(), local/param x =", x)
         x = 2
         print("in inner(), local/param x =", x)

    print("before inner(), x =", x)
    inner(x)
    print("after inner(), x =", x)

x = 0
print("at program start, global x is", x)
outer()
print("after calling outer(), global x is", x)
```

`global` keyword let us access to global vars inside a function

locally defined vars which have the same name as a var in an enclosing scope will hide the outer variable

which x is this?

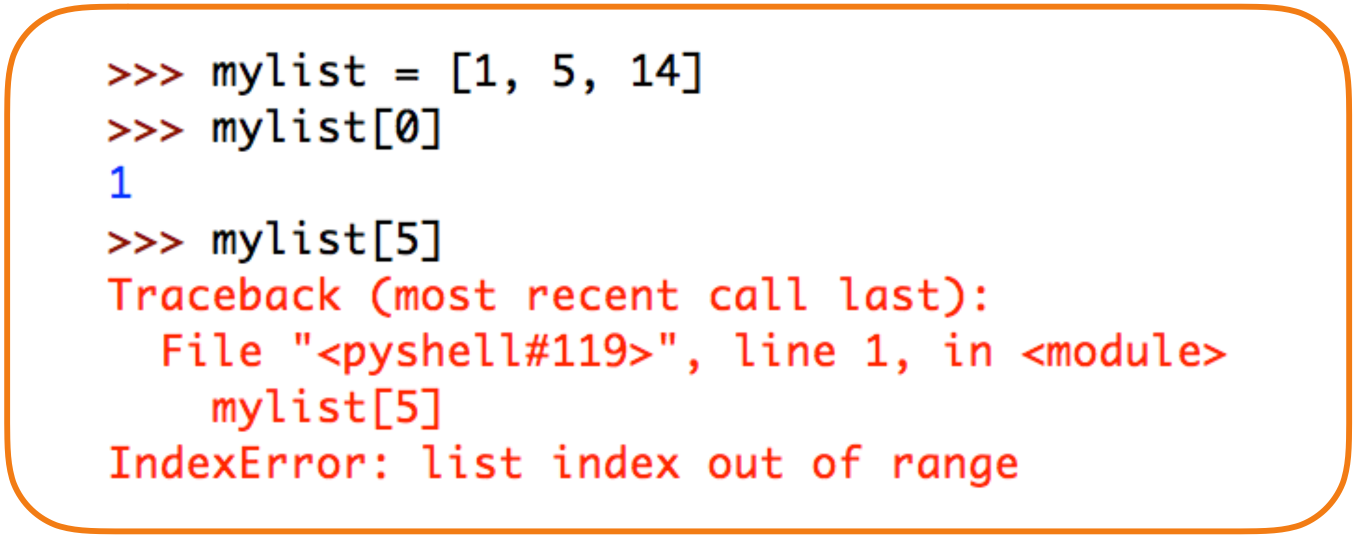
functions: recap

- Python encourages functions which support lots of arguments with default values
- ***"Explicit is better than implicit"***
  - arguments can be passed out of order ONLY if they're passed by keyword
  - keywords are more explicit than positions because the function call documents the purpose of its arguments
- variable positional args (*args)
- variable keyword args (**kwargs)

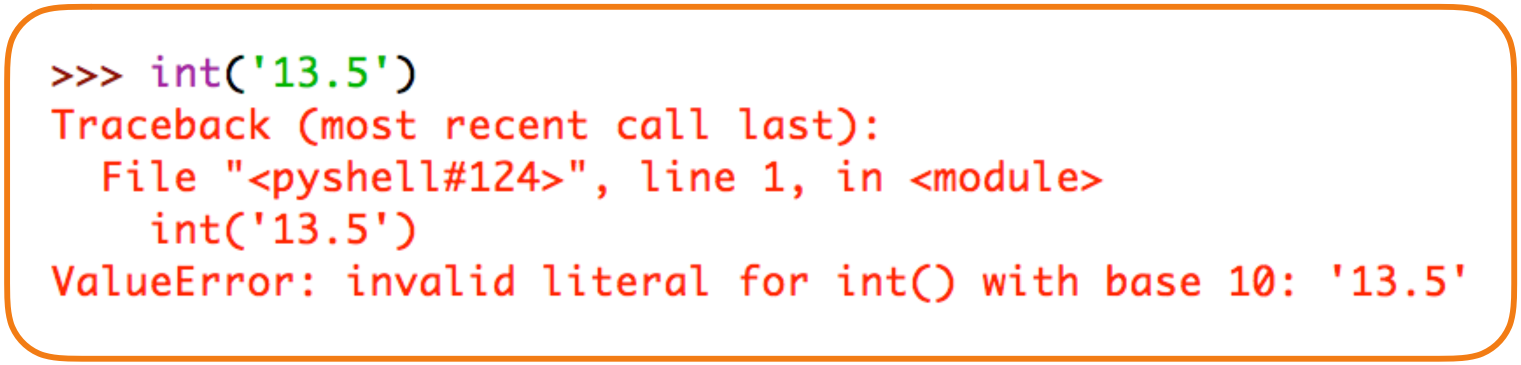
exceptions

- errors detected during execution are called *exceptions*
- exceptions are "thrown" and either "caught" by an exception handler, or propagated upward
- "…exceptions create hidden control-flow paths that are difficult for programmers to reason about" –Weimer & Necula, "Exceptional Situations and Program Reliability"

exceptions (cont'd)

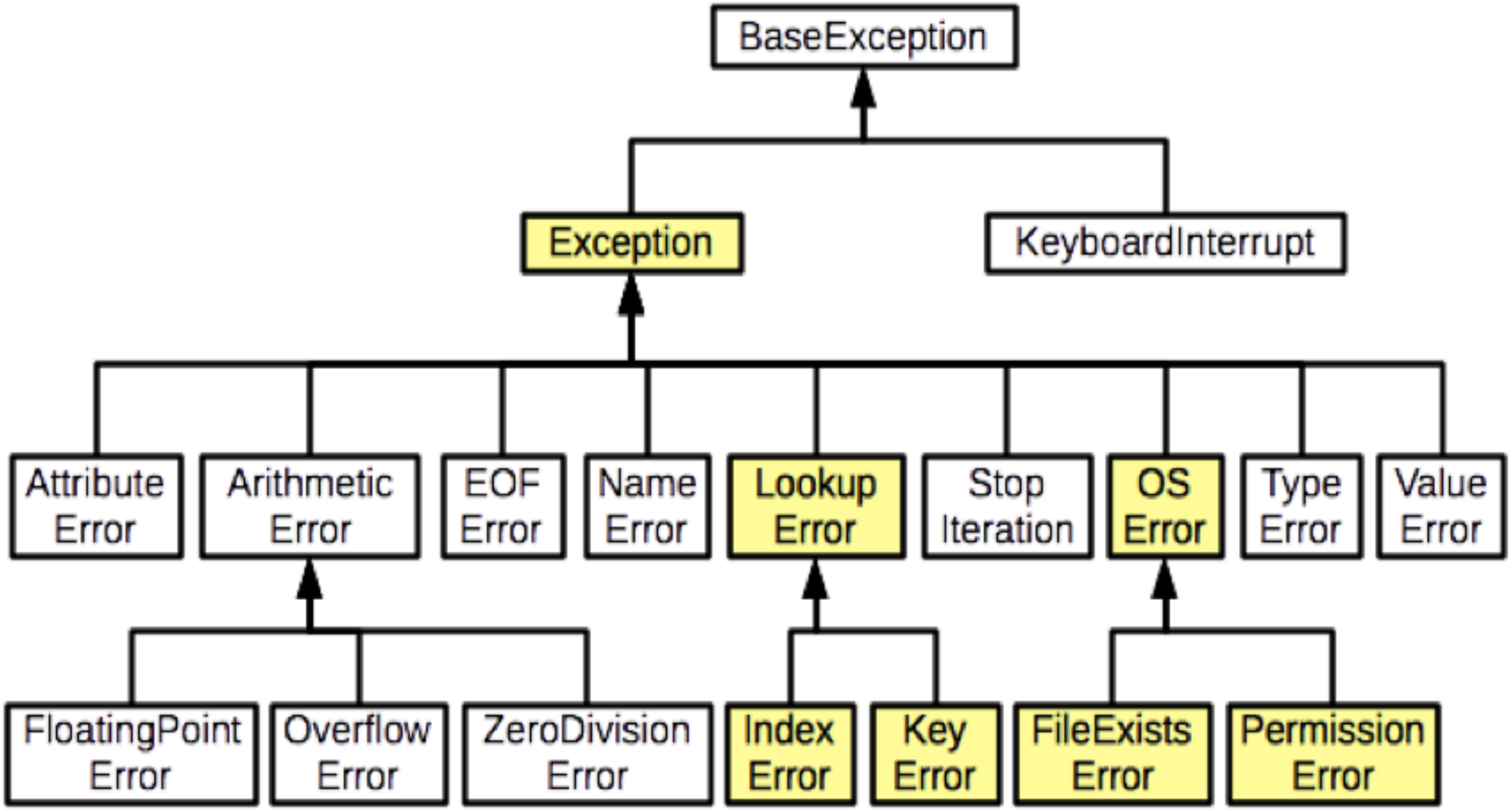


```
>>> mylist = [1, 5, 14]
>>> mylist[0]
1
>>> mylist[5]
Traceback (most recent call last):
  File "<pyshell#119>", line 1, in <module>
    mylist[5]
IndexError: list index out of range
```



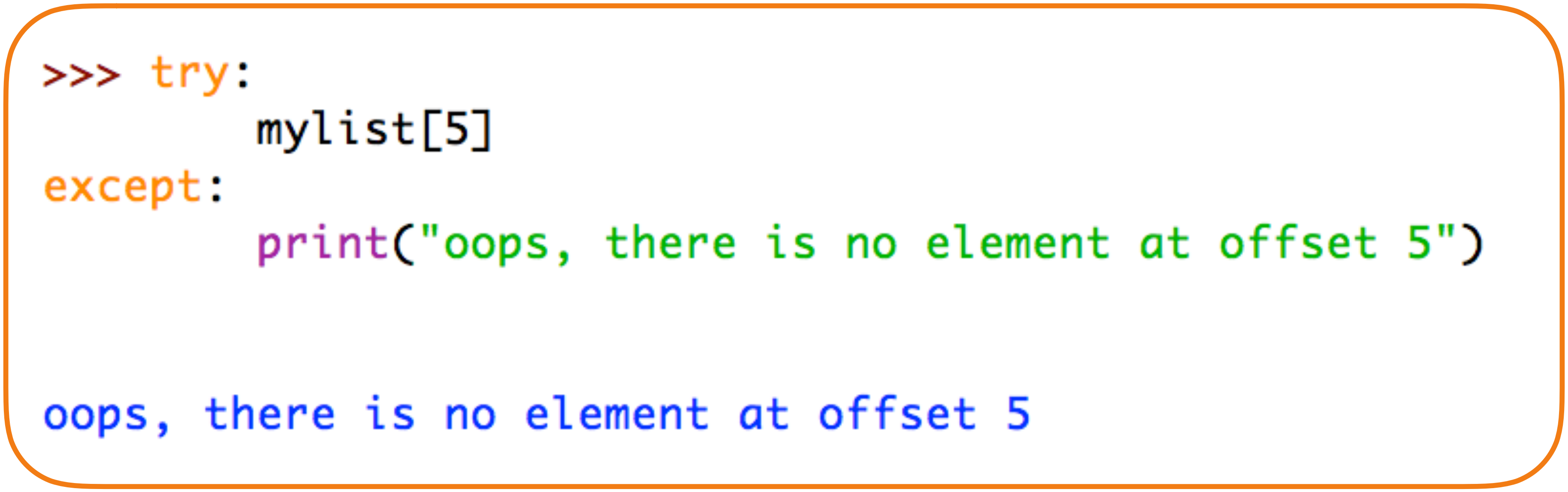
```
>>> int('13.5')
Traceback (most recent call last):
  File "<pyshell#124>", line 1, in <module>
    int('13.5')
ValueError: invalid literal for int() with base 10: '13.5'
```

exceptions (cont'd)



exceptions: try/except

- `try` block wraps code which may throw an exception, and `except` block catches exception

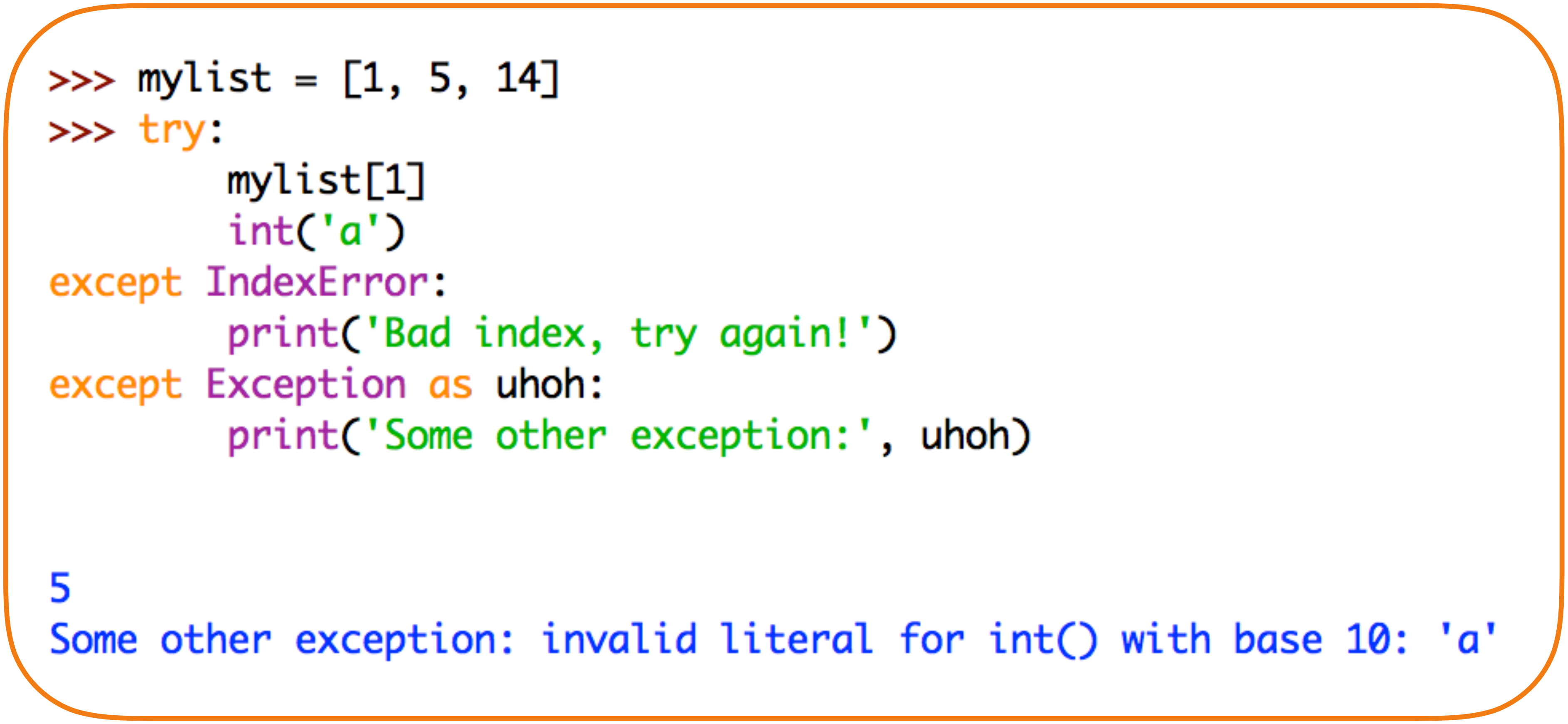


```
>>> try:
        mylist[5]
except:
        print("oops, there is no element at offset 5")


oops, there is no element at offset 5
```

- problem? above example catches ALL exceptions, not just `IndexError` we are expecting

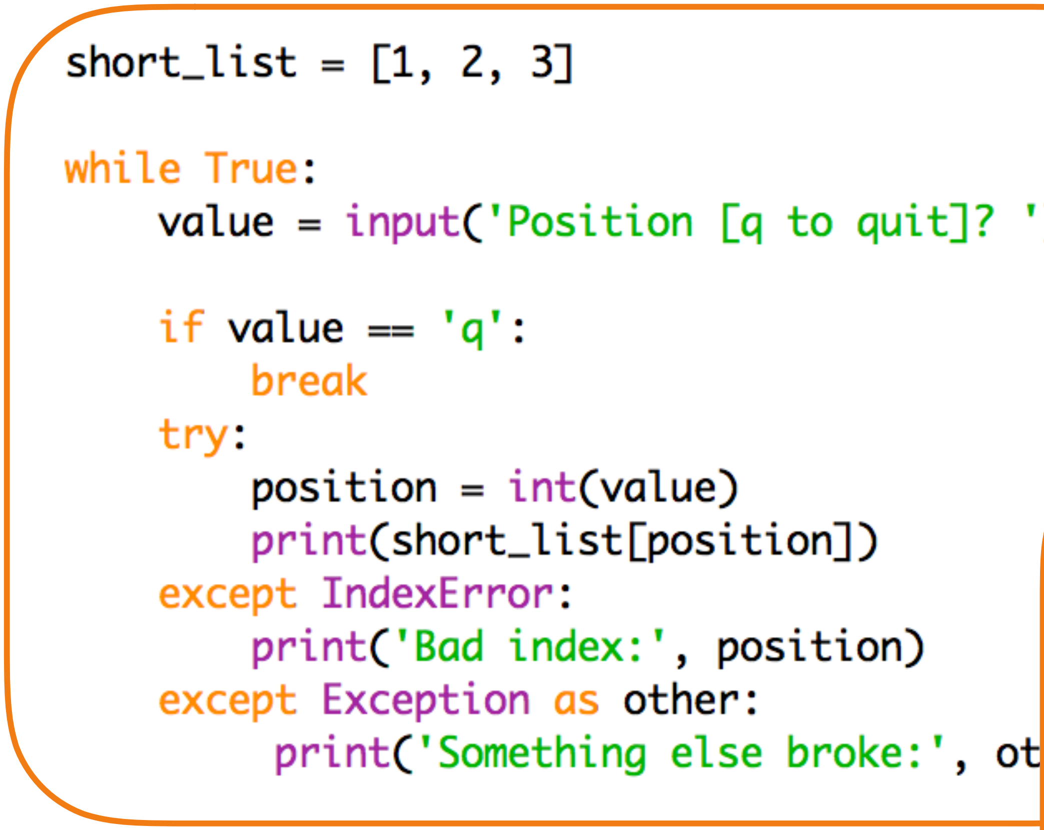
exceptions: try/except (cont'd)

- best practice is to catch expected exceptions and let unexpected ones through, so as to avoid hidden errors

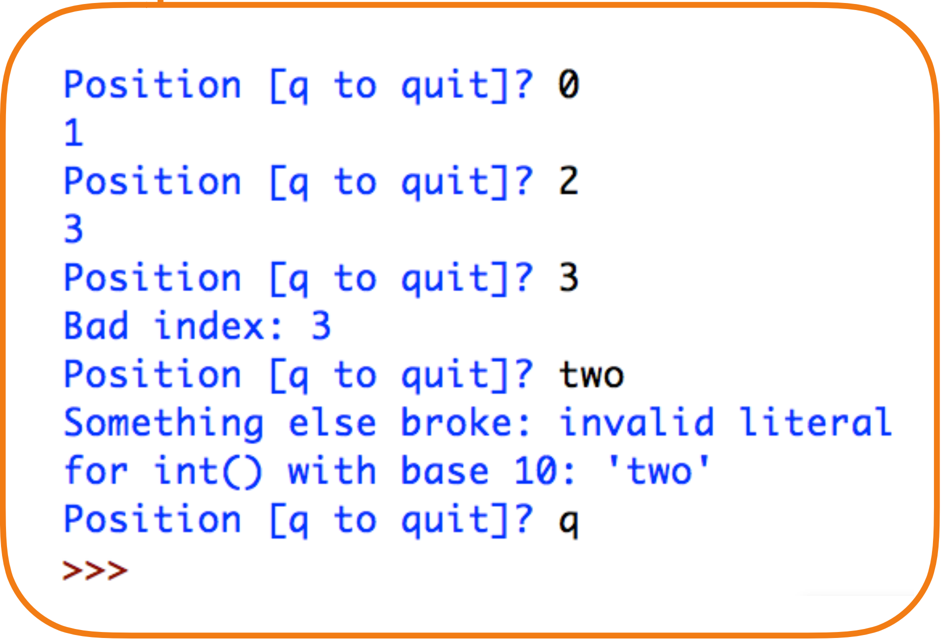


```
>>> mylist = [1, 5, 14]
>>> try:
        mylist[1]
        int('a')
except IndexError:
        print('Bad index, try again!')
except Exception as uhoh:
        print('Some other exception:', uhoh)


5
Some other exception: invalid literal for int() with base 10: 'a'
```

exceptions: try/except (cont'd)



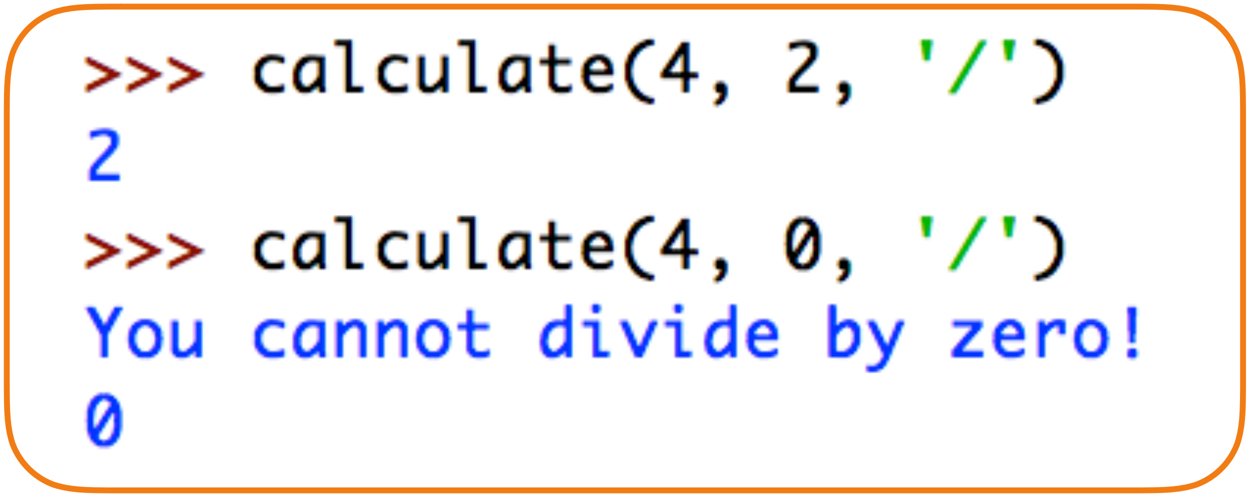
```
short_list = [1, 2, 3]

while True:
    value = input('Position [q to quit]? ')

    if value == 'q':
        break
    try:
        position = int(value)
        print(short_list[position])
    except IndexError:
        print('Bad index:', position)
    except Exception as other:
         print('Something else broke:', ot
```



```
Position [q to quit]? 0
1
Position [q to quit]? 2
3
Position [q to quit]? 3
Bad index: 3
Position [q to quit]? two
Something else broke: invalid literal
for int() with base 10: 'two'
Position [q to quit]? q
>>>
```

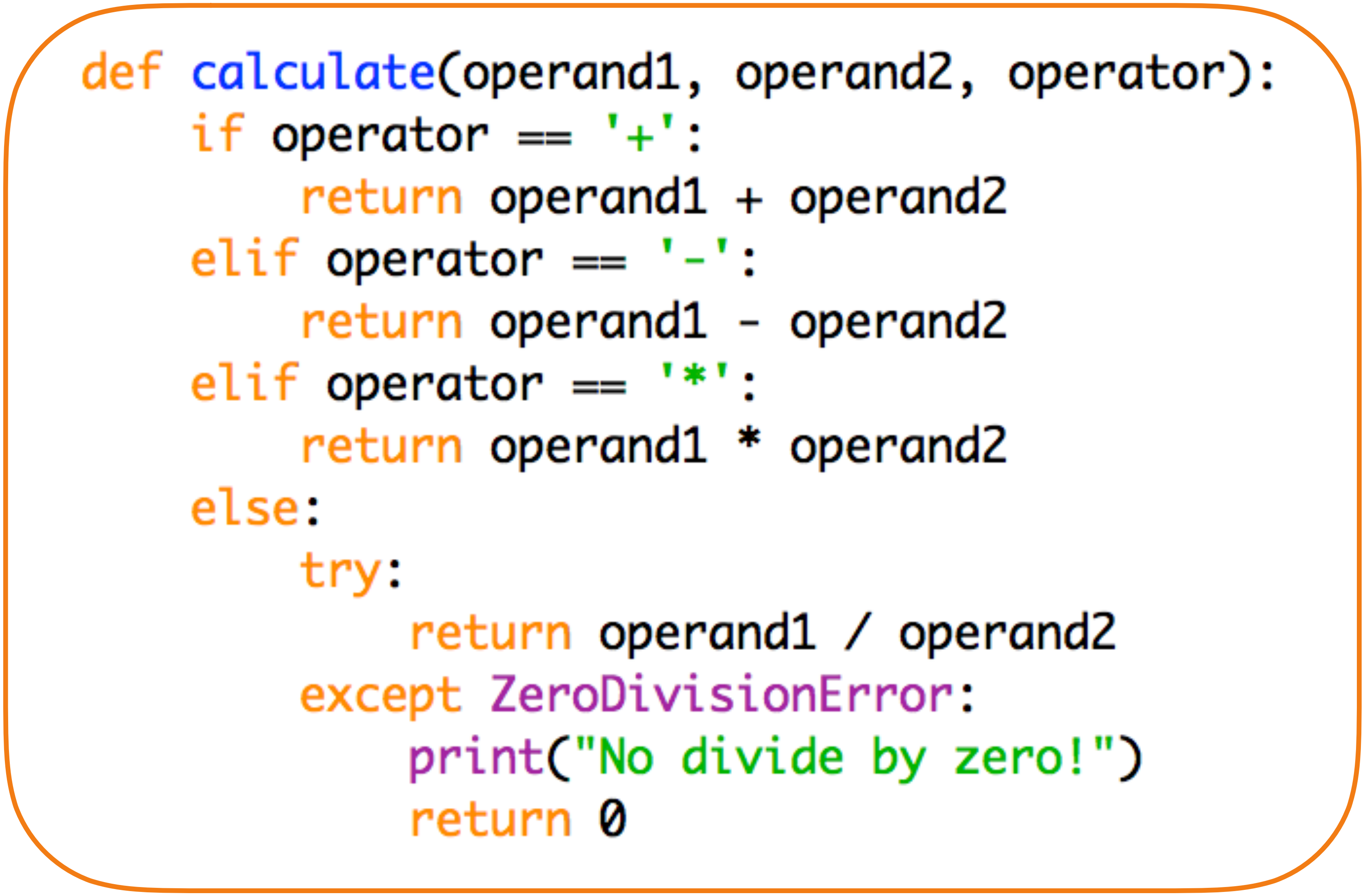
lab: exceptions

- modify your `calculate` function to catch the `ZeroDivisionError` exception and print an informative message if the user tries to divide by zero, e.g.,



```
>>> calculate(4, 2, '/')
2
>>> calculate(4, 0, '/')
You cannot divide by zero!
0
```

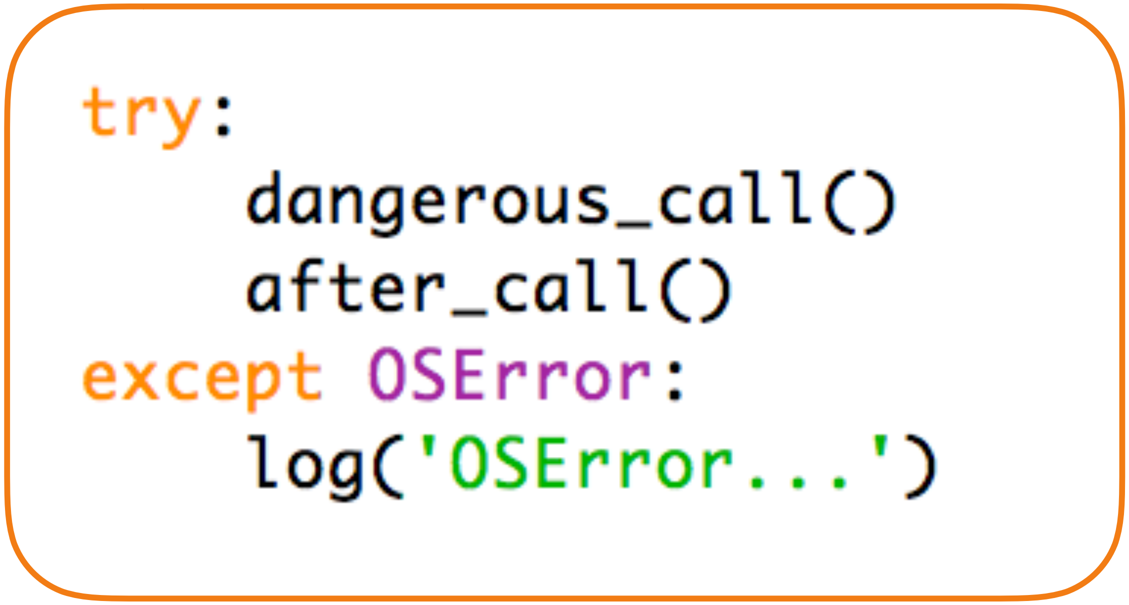
lab: exceptions (solution)



```python
def calculate(operand1, operand2, operator):
    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    else:
        try:
            return operand1 / operand2
        except ZeroDivisionError:
            print("No divide by zero!")
            return 0
```

exceptions: (cont'd)

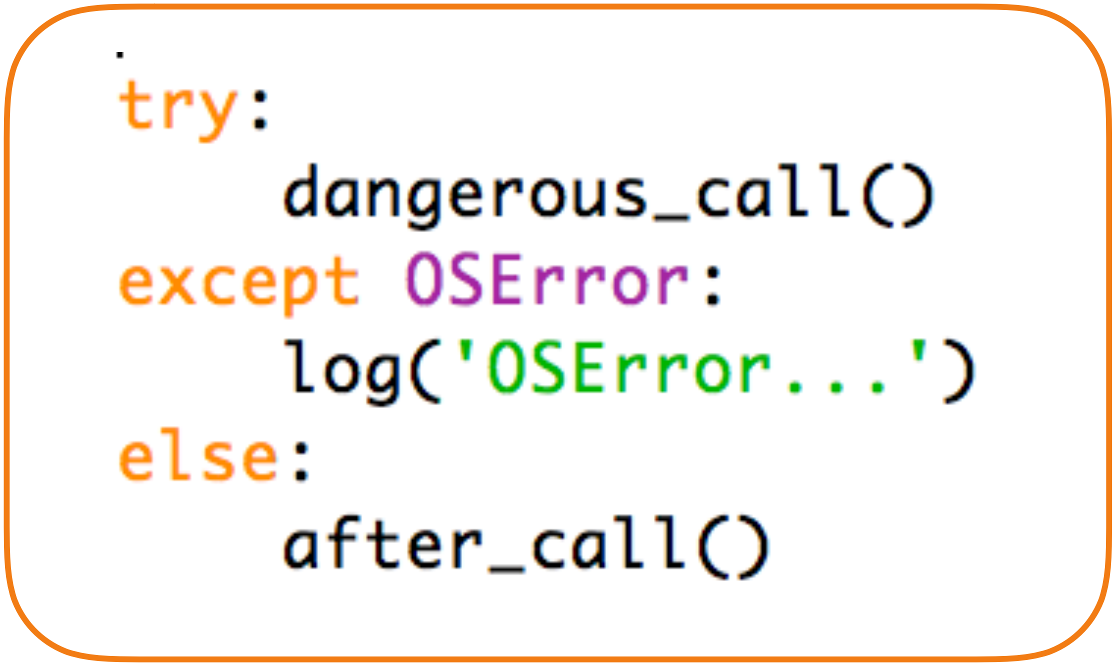
- important to minimize size of try block



```
try:
    dangerous_call()
    after_call()
except OSError:
    log('OSError...')
```

- `after_call()` will only run if `dangerous_call()` doesn't throw an exception…So what's the problem?

try/else (cont'd)



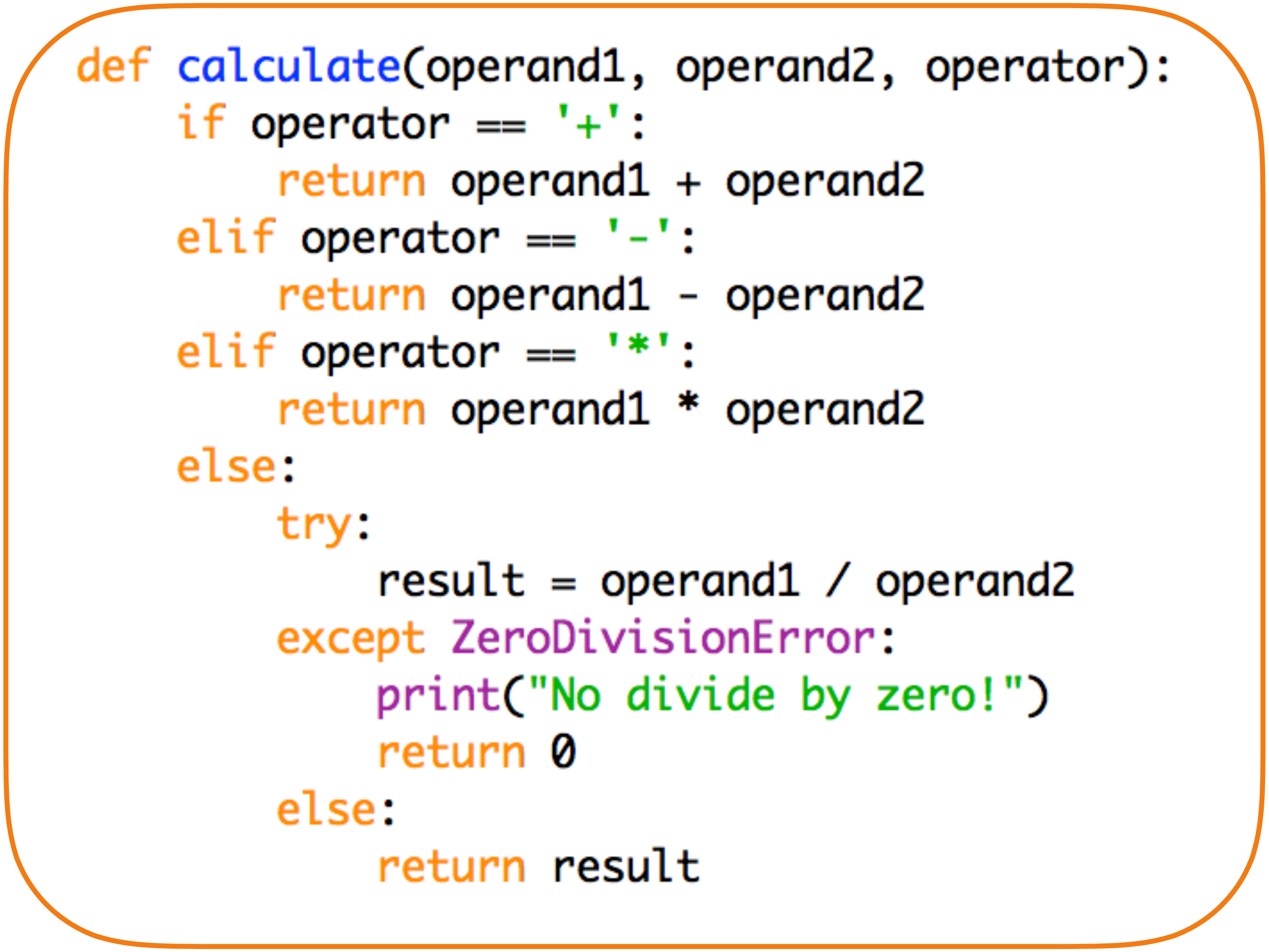
```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

- now it’s clear that `try` block is guarding against possible errors in `dangerous_call()`, not in `after_call()`
- it’s also more obvious that `after_call()` will only execute if no exceptions are raised in the `try` block

lab: exceptions

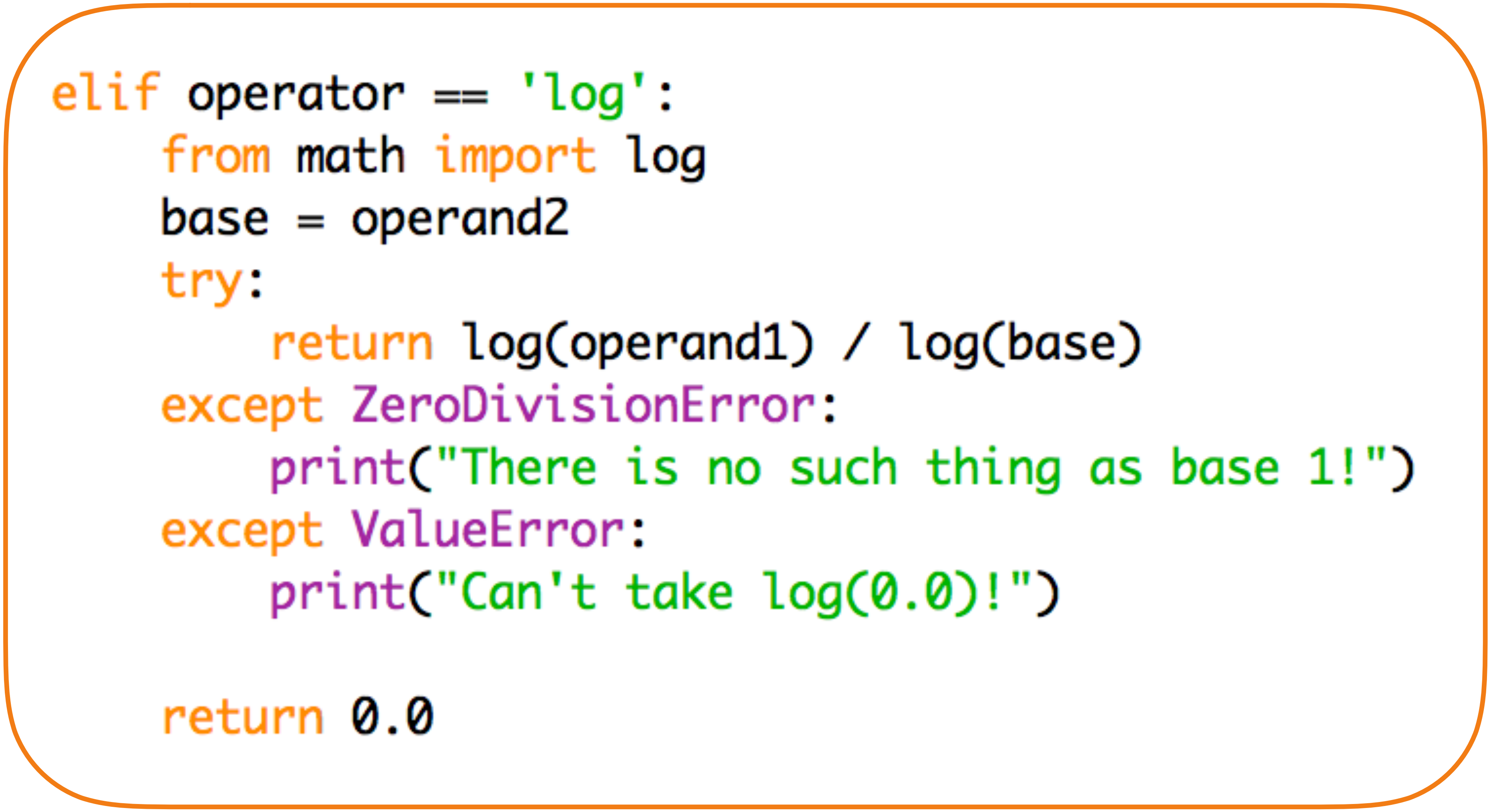
- modify the exception handler in your `calculate` function to include an `else` block and move code to the `else` block except code which may throw an exception
- extend your calculator to include a log() function where the second argument is the base, i.e,. `calculate(49.0, 7, 'log')` = $\mathtt{log}_7\mathtt{(49.0)}$ = `2.0`
- be sure you have a `try/except/else` block for your `log()` function
- (remember that $\mathtt{log}_b\mathtt{(x) = log}_a\mathtt{(x)/log(a)}$)

lab: exceptions (solution)



```python
def calculate(operand1, operand2, operator):
    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    else:
        try:
            result = operand1 / operand2
        except ZeroDivisionError:
            print("No divide by zero!")
            return 0
        else:
            return result
```

lab: exceptions (solution 2)



```
elif operator == 'log':
    from math import log
    base = operand2
    try:
        return log(operand1) / log(base)
    except ZeroDivisionError:
        print("There is no such thing as base 1!")
    except ValueError:
        print("Can't take log(0.0)!")

    return 0.0
```