Python Fundamentals
Day 5 - Functions

Objectives

After this lecture you will be able to:

- Write functions to perform a series of operations in an encapsulated way
    - Understand function syntax
    - Pass arguments
    - Name and access parameters
    - Understand variable scope
- Use if __name__ == ‘__main__:’ blocks and the python debugger to write and debug command-line executable code
- Use list comprehensions to efficiently and eloquently construct lists

Motivation

Functions are a way to:

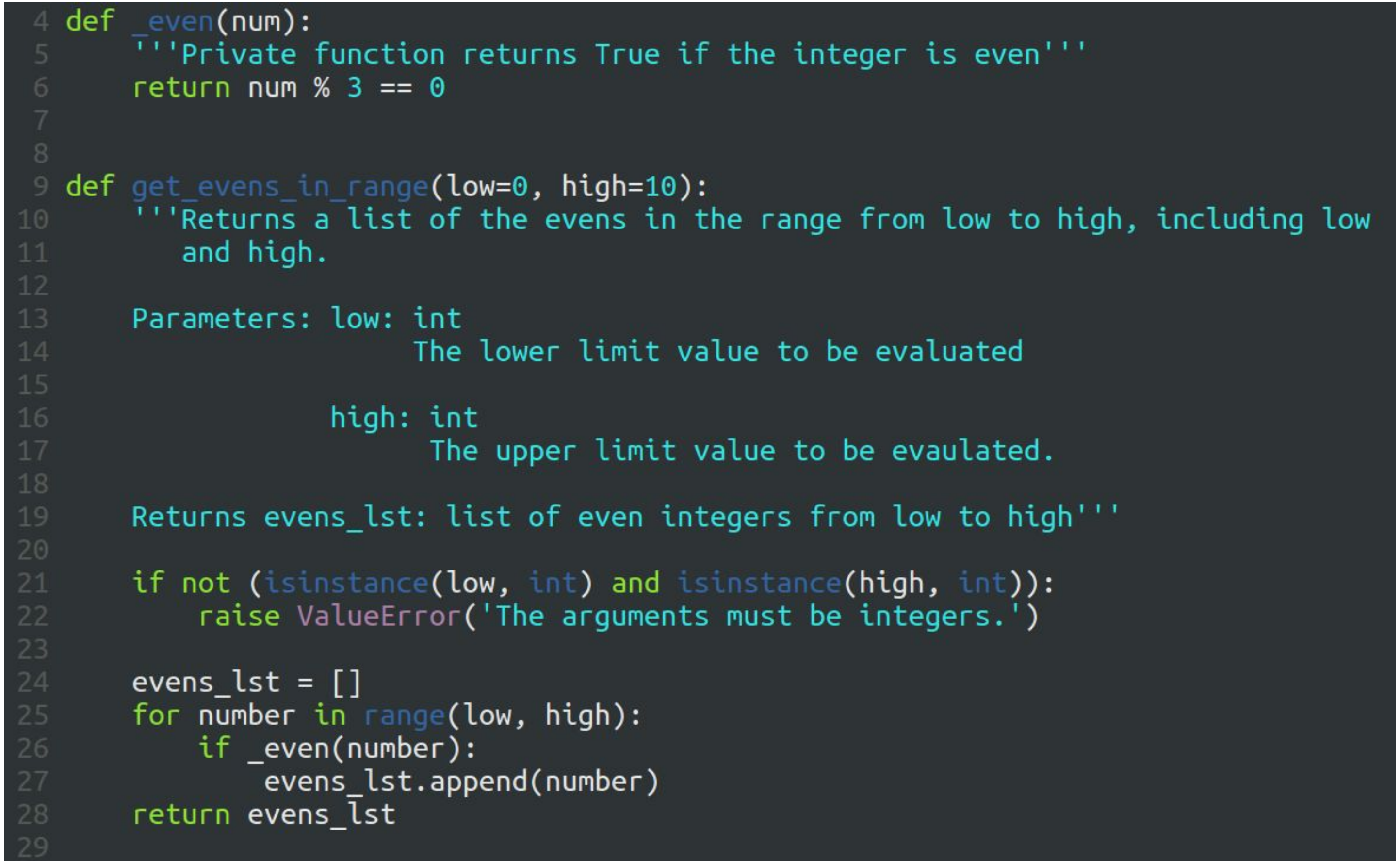
- Re-use code
  - Once your code does something useful and is debugged, why not save it for later?
- Abstract away the details
  - For you and others using it, once it works do you really need to see all the lines that go into it?
  - round example in Ipython, show code on Github
- Make your code more readable

Functions - definition

In computer science, a function is known as a *subroutine*.

A *subroutine* (function in Python) is defined as a sequence of instructions that perform a specific task, packaged together as a unit - i.e. a small, independent piece of code.

Functions - example (with some issues)



```python
def _even(num):
    '''Private function returns True if the integer is even'''
    return num % 3 == 0


def get_evens_in_range(low=0, high=10):
    '''Returns a list of the evens in the range from low to high, including low
       and high.

    Parameters: low: int
                     The lower limit value to be evaluated

                high: int
                      The upper limit value to be evaulated.

    Returns evens_lst: list of even integers from low to high'''

    if not (isinstance(low, int) and isinstance(high, int)):
        raise ValueError('The arguments must be integers.')

    evens_lst = []
    for number in range(low, high):
        if _even(number):
            evens_lst.append(number)
    return evens_lst

```

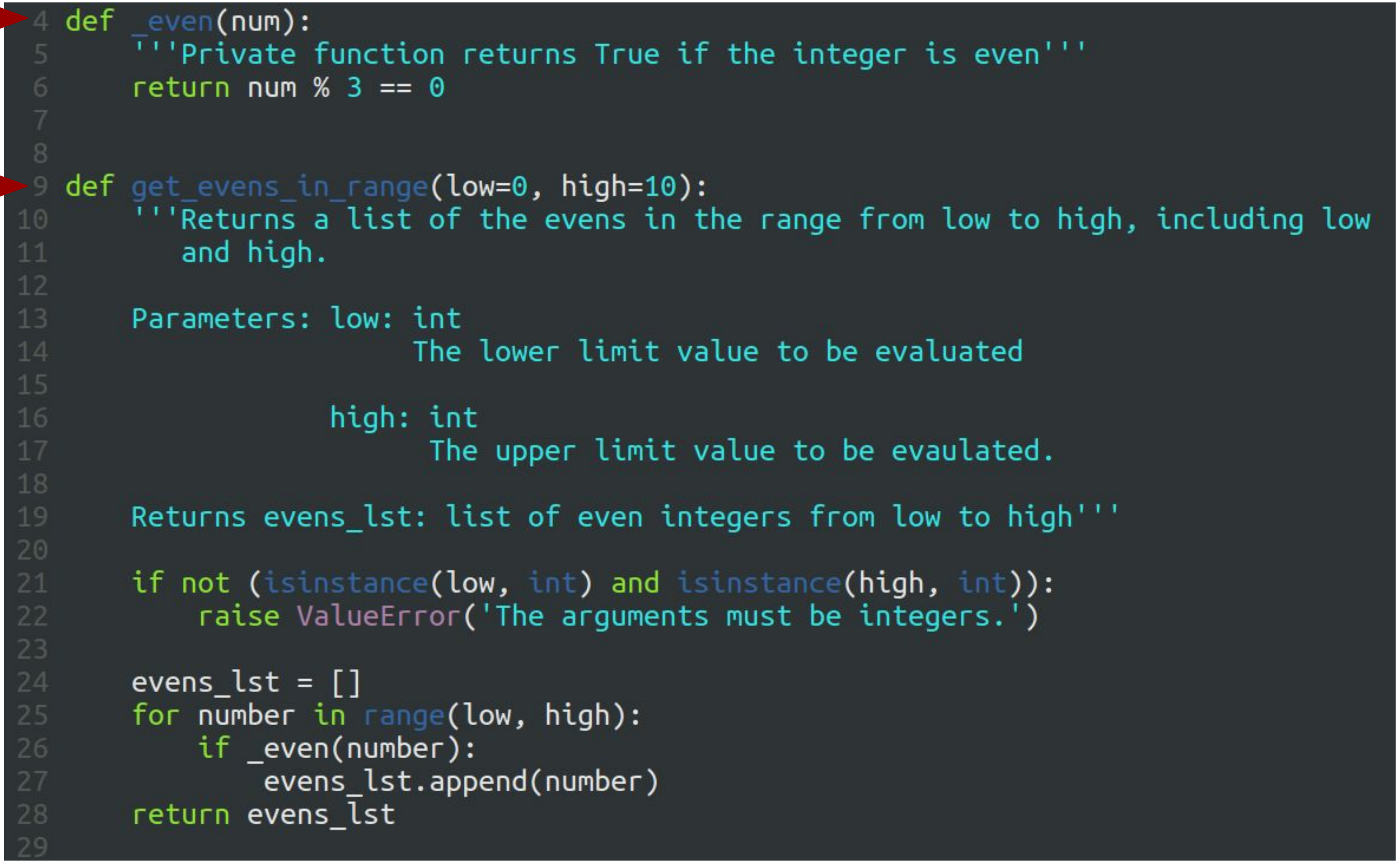
Functions - definition

Function definitions in *snakecase*.

snakecase is lower_case_separated_by_underscores

Give your functions descriptive names.

After the name are (always) parentheses where you put your parameters, if you have them.



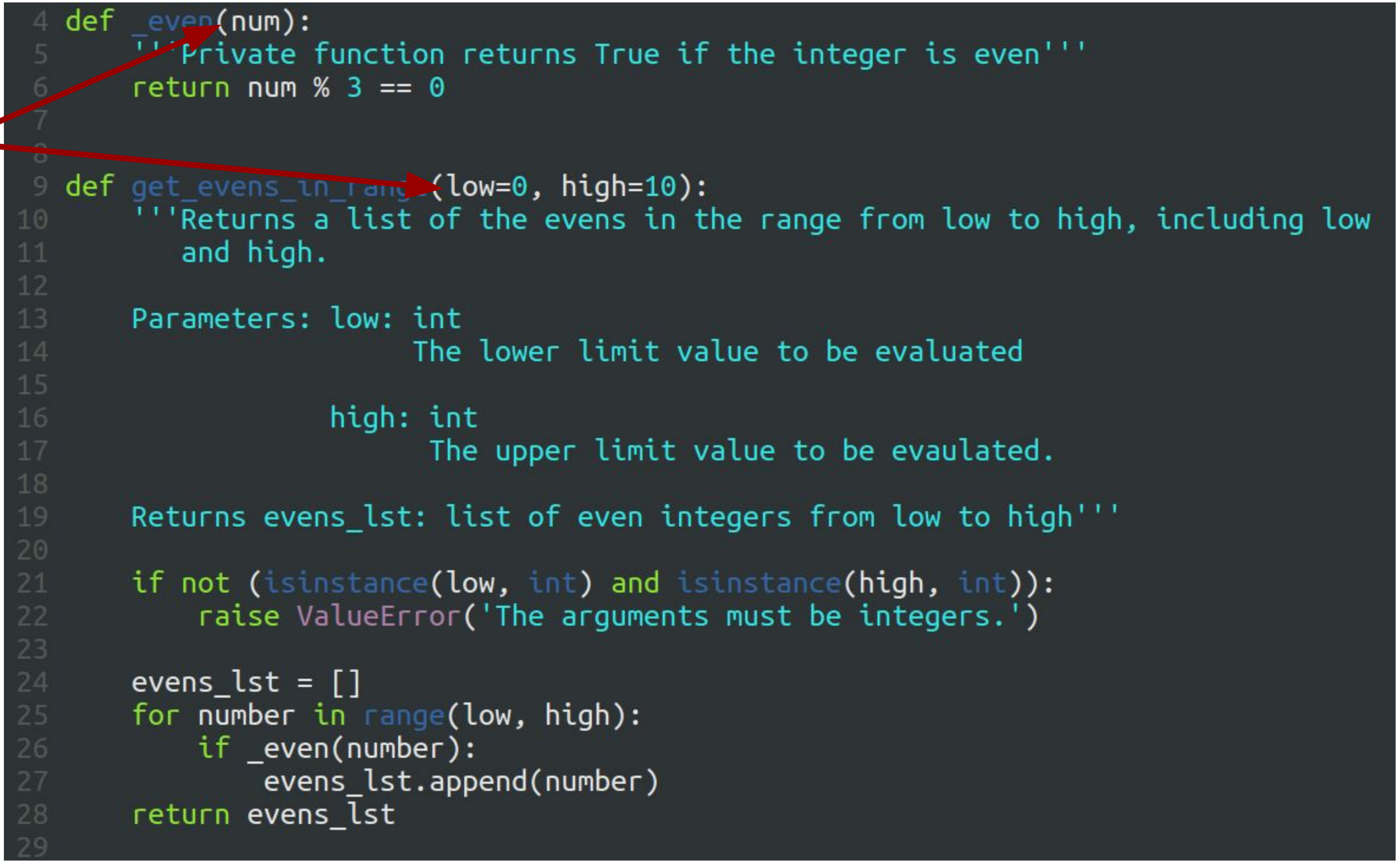
```python
def _even(num):
    '''Private function returns True if the integer is even'''
    return num % 3 == 0


def get_evens_in_range(low=0, high=10):
    '''Returns a list of the evens in the range from low to high, including low
       and high.

    Parameters: low: int
                     The lower limit value to be evaluated

                high: int
                      The upper limit value to be evaulated.

    Returns evens_lst: list of even integers from low to high'''

    if not (isinstance(low, int) and isinstance(high, int)):
        raise ValueError('The arguments must be integers.')

    evens_lst = []
    for number in range(low, high):
        if _even(number):
            evens_lst.append(number)
    return evens_lst

```

Functions - parameters

Parameters define what will be passed to the function.

Parameters can be given default values, e.g. `low=0`

Common error: putting default parameters before non-default parameters in the definition, e.g. `(low=0, high):`



```python
def _even(num):
    '''Private function returns True if the integer is even'''
    return num % 3 == 0


def get_evens_in_range(low=0, high=10):
    '''Returns a list of the evens in the range from low to high, including low
       and high.

    Parameters: low: int
                     The lower limit value to be evaluated

                high: int
                      The upper limit value to be evaulated.

    Returns evens_lst: list of even integers from low to high'''

    if not (isinstance(low, int) and isinstance(high, int)):
        raise ValueError('The arguments must be integers.')

    evens_lst = []
    for number in range(low, high):
        if _even(number):
            evens_lst.append(number)
    return evens_lst
```

Functions - doc. strings

Document strings!

You (and others) will thank you later.

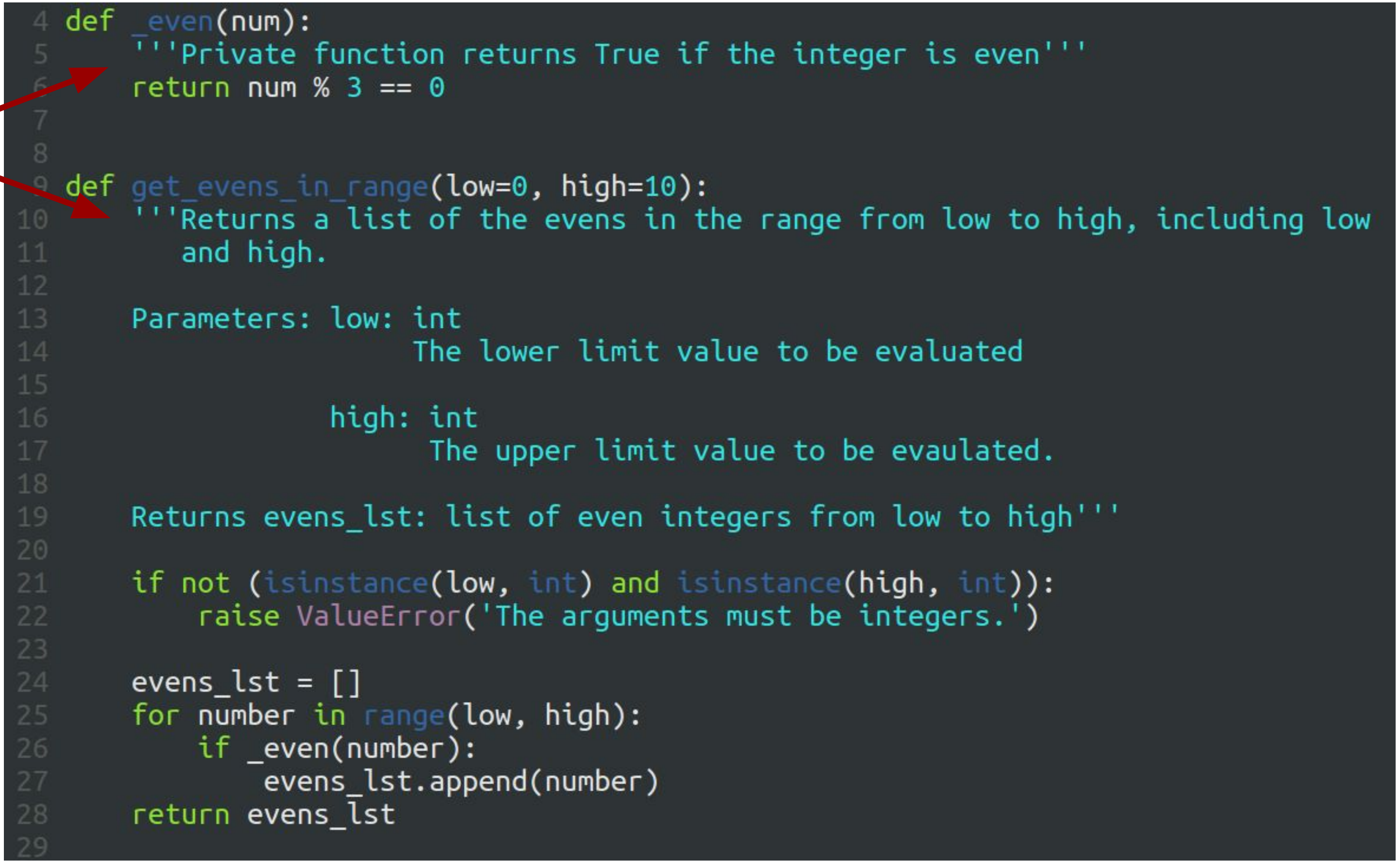
To access externally:

`In[]: get_evens_in_range?`

For built-ins:

`In[]: round?`

`$ man round`



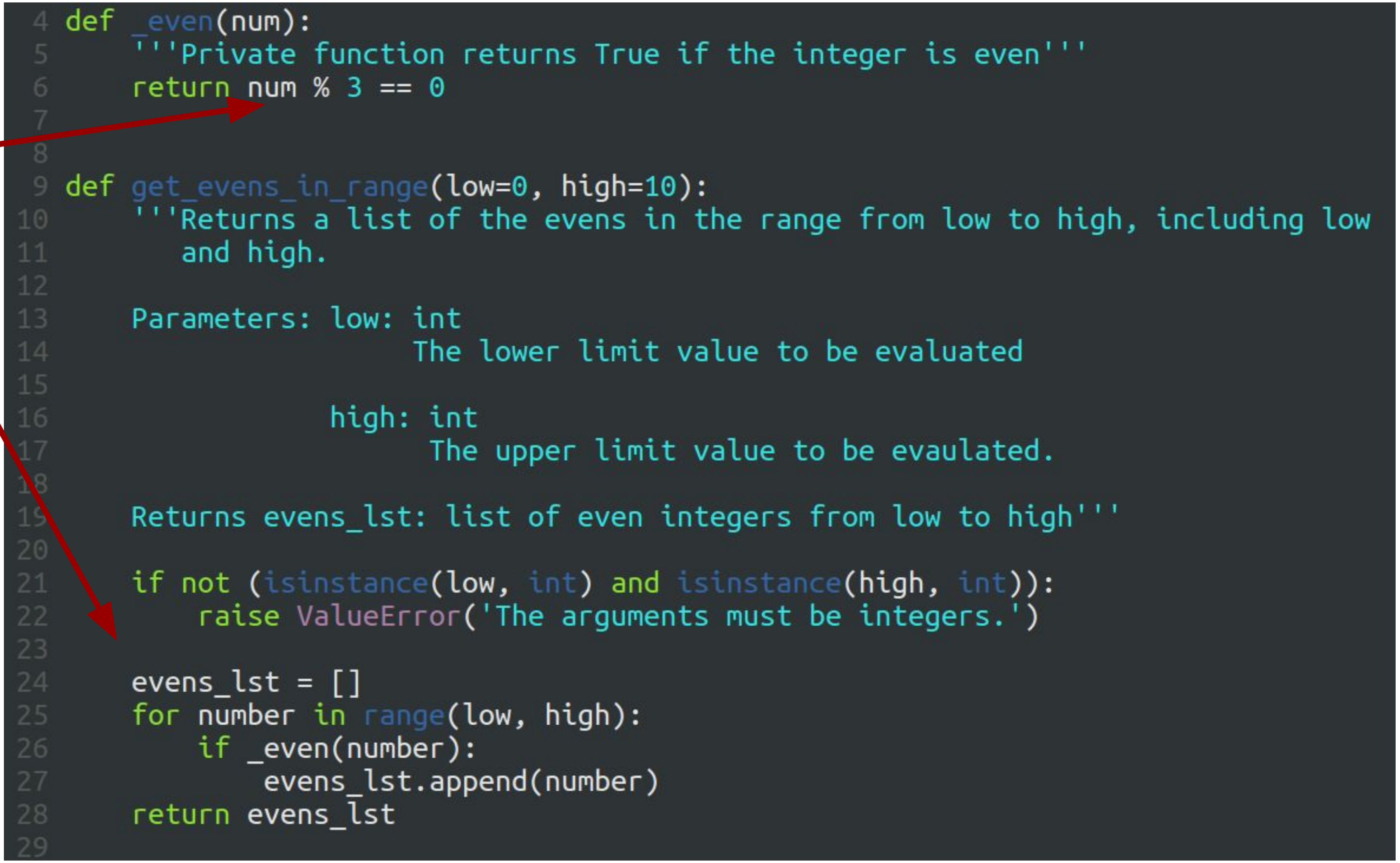
```python
def _even(num):
    '''Private function returns True if the integer is even'''
    return num % 3 == 0


def get_evens_in_range(low=0, high=10):
    '''Returns a list of the evens in the range from low to high, including low
       and high.

    Parameters: low: int
                     The lower limit value to be evaluated

                high: int
                      The upper limit value to be evaulated.

    Returns evens_lst: list of even integers from low to high'''

    if not (isinstance(low, int) and isinstance(high, int)):
        raise ValueError('The arguments must be integers.')

    evens_lst = []
    for number in range(low, high):
        if _even(number):
            evens_lst.append(number)
    return evens_lst

```

Functions - the code that does the work

Here’s where you make the functionality...of your function.

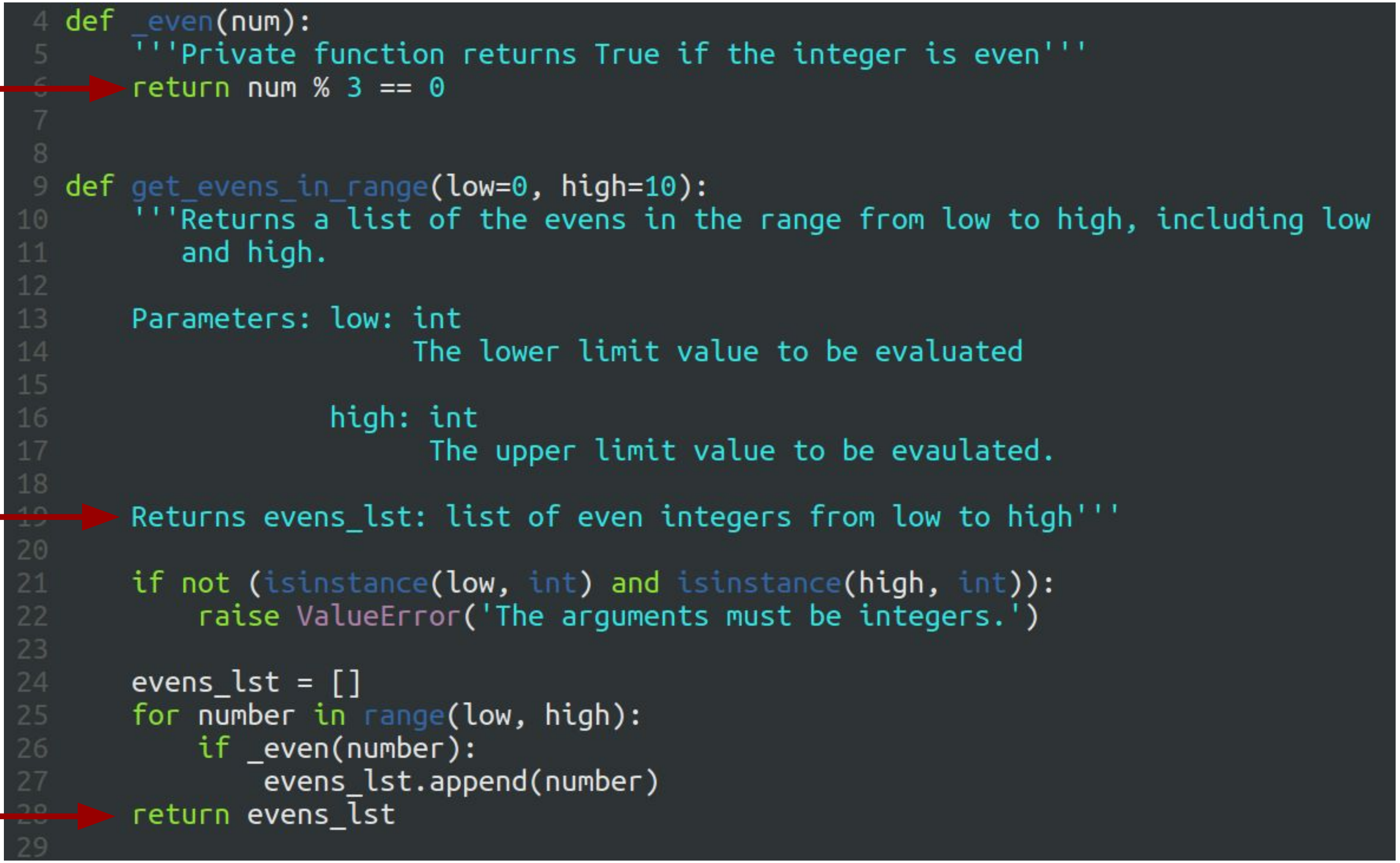
Functions can use other functions (e.g. `_evens()`). Good practice if it makes your code simpler and easier to read.

Think about this for variable names, too. A variable name *should* tell you what it is.



```python
def _even(num):
    '''Private function returns True if the integer is even'''
    return num % 3 == 0


def get_evens_in_range(low=0, high=10):
    '''Returns a list of the evens in the range from low to high, including low
       and high.

    Parameters: low: int
                     The lower limit value to be evaluated

                high: int
                      The upper limit value to be evaulated.

    Returns evens_lst: list of even integers from low to high'''

    if not (isinstance(low, int) and isinstance(high, int)):
        raise ValueError('The arguments must be integers.')

    evens_lst = []
    for number in range(low, high):
        if _even(number):
            evens_lst.append(number)
    return evens_lst

```

Functions - Return (getting something back)



```python
def _even(num):
    '''Private function returns True if the integer is even'''
    return num % 3 == 0


def get_evens_in_range(low=0, high=10):
    '''Returns a list of the evens in the range from low to high, including low
       and high.

    Parameters: low: int
                     The lower limit value to be evaluated

                high: int
                      The upper limit value to be evaulated.

    Returns evens_lst: list of even integers from low to high'''

    if not (isinstance(low, int) and isinstance(high, int)):
        raise ValueError('The arguments must be integers.')

    evens_lst = []
    for number in range(low, high):
        if _even(number):
            evens_lst.append(number)
    return evens_lst

```

Good practice, document string explicitly says what is returned.

Functions - How to use them

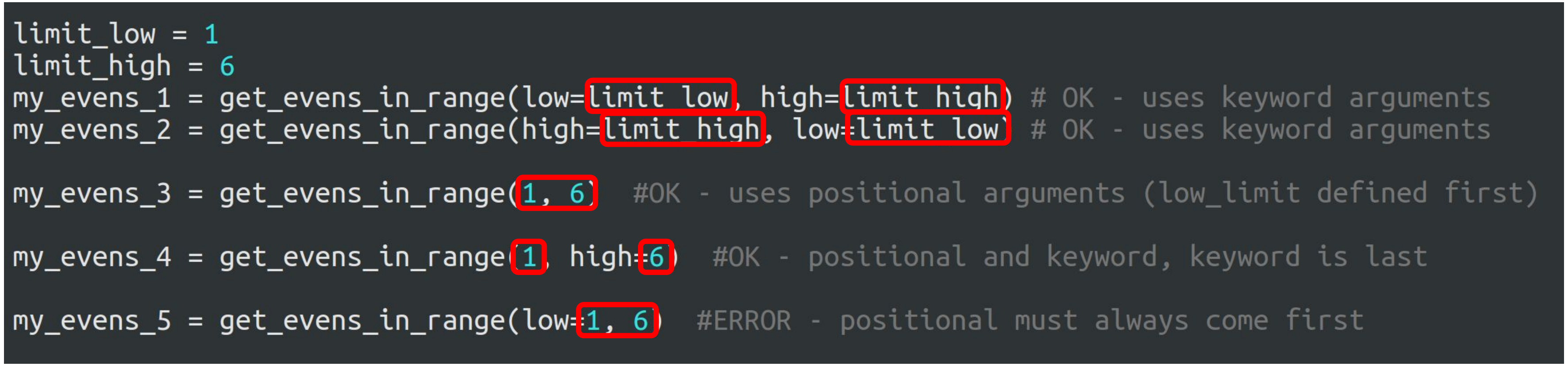
```
limit_low = 1
limit_high = 6
my_evens_1 = get_evens_in_range(low=limit_low, high=limit_high) # OK - uses keyword arguments
my_evens_2 = get_evens_in_range(high=limit_high, low=limit_low) # OK - uses keyword arguments

my_evens_3 = get_evens_in_range(1, 6)  #OK - uses positional arguments (low_limit defined first)

my_evens_4 = get_evens_in_range(1, high=6)  #OK - positional and keyword, keyword is last

my_evens_5 = get_evens_in_range(low=1, 6)  #ERROR - positional must always come first
```

An **argument** is the value that is passed to a function when it is called.

Arguments can be passed in positionally (order defined by function), by keyword (parameter name), or a mix.

But positional arguments must always come before keyword arguments.

Functions - arguments vs. parameters



```
limit_low = 1
limit_high = 6
my_evens_1 = get_evens_in_range(low=limit_low, high=limit_high) # OK - uses keyword arguments
my_evens_2 = get_evens_in_range(high=limit_high, low=limit_low) # OK - uses keyword arguments

my_evens_3 = get_evens_in_range(1, 6)  #OK - uses positional arguments (low_limit defined first)

my_evens_4 = get_evens_in_range(1, high=6)  #OK - positional and keyword, keyword is last

my_evens_5 = get_evens_in_range(low=1, 6)  #ERROR - positional must always come first
```

A **parameter** is the name of a variable given in a function definition.

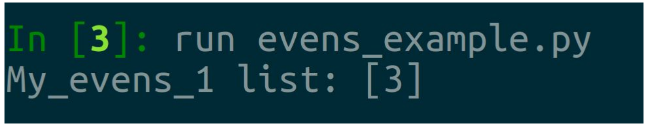
An **argument** is the value that is passed to a function when it is called.

Let’s fix `get_evens_in_range()`

My working environment:

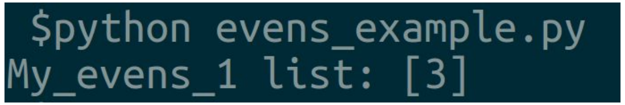
- Text editor
  - write a command-line executable script
  - script imports the python debugger
- Terminal running Ipython
  - test code snippets
  - check syntax
  - check documentation
  - where I run the script
  - check script outputs
- Terminal
  - can run script from command line
  - file navigation
  - simple bash commands (e.g. less)

Run from within Ipython:



```
In [3]: run evens_example.py
My_evens_1 list: [3]
```

Run from command line:



```
 $python evens_example.py
My_evens_1 list: [3]
```

Motivation for command line execution

Show UAV Delivery for Denver video

Then show script that made it.

Python debugger

At top of script:

```
import pdb
```

Where you want to start debugging:
pdb.set_trace()

Common commands:

n - Next line (but does not step into a function)

s - Next line (and steps into a function)

c - Continue (stops debugging and executes program)

q - Quit

Many more commands available. See documentation at:
https://docs.python.org/2/library/pdb.html