In [3]:

```
def initlog(*args):
    pass    # Remember to implement this!
```

- The keyword def introduces a function definition.
- It must be followed by the function name and the parenthesized list of formal parameters.
- The statements that form the body of the function start at the next line, and must be indented.
- The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring.

In [4]:

```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
```

In [5]:

```
fib(2000)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

In [6]:

```
fib
```

Out[6]:

```
<function __main__.fib(n)>
```

In [8]:

```
f=fib
```

In [9]:

```
f(100)
```

```
0 1 1 2 3 5 8 13 21 34 55 89
```

Functions without a return statement do return a value, this value is called None
Writing the value None is normally suppressed by the interpreter if it would be the only value written.

In [10]:

```
fib(0)
```

In [11]:

```
print(fib(0))
```

```
None
```

In [12]:

```
def fib2(n):  # return Fibonacci series up to n
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)    # see below
        a, b = b, a+b
    return result
```

In [13]:

```
f100 = fib2(100)    # call it
```

In [14]:

```
f100
```

Out[14]:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow.

In [15]:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument:

  ```
  ask_ok('Do you really want to quit?')
  ```

- giving one of the optional arguments:

  ```
  ask_ok('OK to overwrite the file?', 2)
  ```

- or even giving all arguments:

  ```
  ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
  ```

In [16]:

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

```
5
```

If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:

In [ ]:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Keyword Arguments

Functions can also be called using keyword arguments of the form kwarg=value.

In [17]:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (voltage) and three optional arguments (state, action, and type). This function can be called in any of the following ways:

In [18]:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

```
-- This parrot wouldn't voom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't voom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't jump if you put a million volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's bereft of life !
-- This parrot wouldn't voom if you put a thousand volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's pushing up the daisies !
```

but all the following calls would be invalid:

In [19]:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

```
  File "<ipython-input-19-2ac707ad11c1>", line 2
    parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword ar
gument
                       ^
SyntaxError: positional argument follows keyword argument
```

- In a function call, keyword arguments must follow positional arguments.
- All the keyword arguments passed must match one of the arguments accepted by the function (e.g. actor is not a valid argument for the parrot function), and their order is not important.
- No argument may receive a value more than once.

In [20]:

```
def function(a):pass
```

In [21]:

```
function(0, a=0)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-21-34e44d693230> in <module>
----> 1 function(0, a=0)

TypeError: function() got multiple values for argument 'a'
```

- When a final formal parameter of the form **name is present, it receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter.
- This may be combined with a formal parameter of the form *name (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (*name must occur before **name.)

In [29]:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
    print(keywords)
    print(type(keywords))
    print(arguments)
    print(type(arguments))
```

In [30]:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
{'shopkeeper': 'Michael Palin', 'client': 'John Cleese', 'sketch': 'Cheese
Shop Sketch'}
<class 'dict'>
("It's very runny, sir.", "It's really very, VERY runny, sir.")
<class 'tuple'>
```

Special parameters

- By default, arguments may be passed to a Python function either by position or explicitly by keyword.
- For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

- where / and * are optional
- If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only.
- Keyword parameters are also referred to as named parameters.

Positional-or-Keyword Arguments

If / and * are not present in the function definition, arguments may be passed to a function by position or by keyword.

In [31]:

```
def standard_arg(arg):
    print(arg)
```

In [32]:

```
standard_arg(2)
```

```
2
```

In [33]:

```
standard_arg(arg=2)
```

```
2
```

Positional-Only Parameters

- If positional-only, the parameters’ order matters, and the parameters cannot be passed by keyword.
- Positional-only parameters are placed before a / (forward-slash).
- The / is used to logically separate the positional-only parameters from the rest of the parameters.
- If there is no / in the function definition, there are no positional-only parameters.

In [1]:

```
def pos_only_arg(arg, /):
    print(arg)
```

In [2]:

```
pos_only_arg(1)
```

```
1
```

In [3]:

```
pos_only_arg(arg=1)# it gives TypeError
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-3-434db44f4ff9> in <module>
----> 1 pos_only_arg(arg=1)

TypeError: pos_only_arg() got some positional-only arguments passed as key
word arguments: 'arg'
```

Keyword-Only Arguments

To mark parameters as keyword-only, indicating the parameters must be passed by keyword argument, place an * in the arguments list just before the first keyword-only parameter.

In [4]:

```
def kwd_only_arg(*, arg):
    print(arg)
```

In [5]:

```
kwd_only_arg(3)#throws an exception
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-5-896c53ef896c> in <module>
----> 1 kwd_only_arg(3)

TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given
```

In [6]:

```
kwd_only_arg(arg=3)
```

```
3
```

In [7]:

```
def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)
```

In [8]:

```
combined_example(1, 2, 3)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-8-037a05a37207> in <module>
----> 1 combined_example(1, 2, 3)

TypeError: combined_example() takes 2 positional arguments but 3 were give
n
```

In [9]:

```
combined_example(1, 2, kwd_only=3)
```

```
1 2 3
```

In [10]:

```
combined_example(1, standard=2, kwd_only=3)
```

```
1 2 3
```

In [11]:

```
combined_example(pos_only=1, standard=2, kwd_only=3)#throw an exception
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-11-a0fe0f8338e9> in <module>
----> 1 combined_example(pos_only=1, standard=2, kwd_only=3)

TypeError: combined_example() got some positional-only arguments passed as
keyword arguments: 'pos_only'
```

In [ ]:

```

```