05-function-parameters

June 14, 2023

0.0.1 Positional Arguments

[1]:
```python
def my_func(a, b, c):
    print("a={0}, b={1}, c={2}".format(a, b, c))
```

[2]:
```python
my_func(1, 2, 3)
```

```
a=1, b=2, c=3
```

Default Values

[3]:
```python
def my_func(a, b=2, c=3):
    print("a={0}, b={1}, c={2}".format(a, b, c))
```

Note that once a parameter is assigned a default value, **all** parameters thereafter **must** be asigned a default value too!

For example, this will not work:

[4]:
```python
def fn(a, b=2, c):
    print(a, b, c)
```

```
  Cell In[4], line 1
    def fn(a, b=2, c):
                   ^
SyntaxError: non-default argument follows default argument
```

[5]:
```python
def my_func(a, b=2, c=3):
    print("a={0}, b={1}, c={2}".format(a, b, c))
```

[6]:
```python
my_func(10, 20, 30)
```

```
a=10, b=20, c=30
```

[7]:
```python
my_func(10, 20)
```

```
a=10, b=20, c=3
```

[8]:
```
my_func(10)
```

```
a=10, b=2, c=3
```

Since **a** does not have a default value, it **must** be specified:

[9]:
```
my_func()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 my_func()

TypeError: my_func() missing 1 required positional argument: 'a'
```

Keyword Arguments (named arguments) Positional arguments, can **optionally**, be specified using their corresponding parameter name.

This allows us to pass the arguments without using the positional assignment:

[ ]:
```
def my_func(a, b=2, c=3):
    print("a={0}, b={1}, c={2}".format(a, b, c))
```

[ ]:
```
my_func(c=30, b=20, a=10)
```

[ ]:
```
my_func(10, c=30, b=20)
```

Note that once a keyword argument has been used, **all** arguments thereafter **must** also be named:

[ ]:
```
my_func(10, b=20, 30)
```

However, if a parameter has a default value, it *can* be omitted from the argument list, named or not:

[ ]:
```
my_func(10, c=30)
```

[10]:
```
my_func(a=30, c=10)
```

```
a=30, b=2, c=10
```

[11]:
```
my_func(c=10, a=30)
```

```
a=30, b=2, c=10
```

0.0.2 Unpacking Iterables

Side Note on Tuples This is a tuple:

[12]:
```
a = (1, 2, 3)
```

[13]:
```
type(a)
```

[13]:
```
tuple
```

This is also a tuple:

[14]:
```
a = 1, 2, 3
```

[15]:
```
type(a)
```

[15]:
```
tuple
```

In fact what defines a tuple is not **()**, but the **,** (comma)

To create a tuple with a single element:

[16]:
```
a = (1)
```

will not work!!

[17]:
```
type(a)
```

[17]:
```
int
```

Instead, we have to use a comma:

[18]:
```
a = (1,)
```

[19]:
```
type(a)
```

[19]:
```
tuple
```

And in fact, we don't even need the **()**:

[20]:
```
a = 1,
```

[21]:
```
type(a)
```

[21]:
```
tuple
```

The only exception is to create an empty tuple:

[22]:
```
a = ()
```

[23]:
```
type(a)
```

[23]:
```
tuple
```

Or we can use the tuple constructor:

[24]:
```
a = tuple()
```

[25]:
```
type(a)
```

[25]:
```
tuple
```

Unpacking Unpacking is a way to split an iterable object into individual variables contained in a list or tuple:

[26]:
```
l = [1, 2, 3, 4]
```

[27]:
```
a, b, c, d = l
```

[28]:
```
print(a, b, c, d)
```

```
1 2 3 4
```

Strings are iterables too:

[29]:
```
a, b, c = 'XYZ'
print(a, b, c)
```

```
X Y Z
```

Swapping Two Variables Here's a quick application of unpacking to swap the values of two variables.

First we look at the "traditional" way you would have to do it in other languages such as Java:

[30]:
```
a = 10
b = 20
print("a={0}, b={1}".format(a, b))

tmp = a
a = b
b = tmp
print("a={0}, b={1}".format(a, b))
```

```
a=10, b=20
a=20, b=10
```

But using unpacking we can simplify this:

[31]:
```
a = 10
b = 20
print("a={0}, b={1}".format(a, b))

a, b = b, a
print("a={0}, b={1}".format(a, b))
```

```
a=10, b=20
a=20, b=10
```

In fact, we can even simplify the initial assignment of values to a and b as follows:

```
[32]: a, b = 10, 20
      print("a={0}, b={1}".format(a, b))

      a, b = b, a
      print("a={0}, b={1}".format(a, b))
```

```
a=10, b=20
a=20, b=10
```

Unpacking Unordered Objects

```
[33]: dict1 = {'p': 1, 'y': 2, 't': 3, 'h': 4, 'o': 5, 'n': 6}
```

```
[34]: dict1
```

```
[34]: {'p': 1, 'y': 2, 't': 3, 'h': 4, 'o': 5, 'n': 6}
```

```
[35]: for c in dict1:
          print(c)
```

```
p
y
t
h
o
n
```

```
[36]: a, b, c, d, e, f = dict1
      print(a)
      print(b)
      print(c)
      print(d)
      print(e)
      print(f)
```

```
p
y
t
h
o
n
```

Note that this order is not guaranteed. You can always use an OrderedDict if that is a requirement.

The same applies to sets.

```
[37]: s = {'p', 'y', 't', 'h', 'o', 'n'}
```

```
[38]: type(s)
```

```
[38]: set
```

```
[39]: print(s)
```

```
{'t', 'p', 'o', 'y', 'h', 'n'}
```

```
[40]: for c in s:
          print(c)
```

```
t
p
o
y
h
n
```

```
[41]: a, b, c, d, e, f = s
```

```
[42]: print(a)
      print(b)
      print(c)
      print(d)
      print(e)
      print(f)
```

```
t
p
o
y
h
n
```

0.0.3 Extended Unpacking

Let's see how we might split a list into it's first element, and "everything else" using slicing:

```
[43]: l = [1, 2, 3, 4, 5, 6]
```

```
[44]: a = l[0]
      b = l[1:]
      print(a)
      print(b)
```

```
1
[2, 3, 4, 5, 6]
```

We can even use unpacking to simplify this slightly:

[45]:
```
a, b = l[0], l[1:]
print(a)
print(b)
```

```
1
[2, 3, 4, 5, 6]
```

But we can use the * operator to achieve the same result:

[46]:
```
a, *b = l
print(a)
print(b)
```

```
1
[2, 3, 4, 5, 6]
```

Note that the * operator can only appear **once**!

Like standard unpacking, this extended unpacking will work with any iterable.

With tuples:

[47]:
```
a, *b = -10, 5, 2, 100
print(a)
print(b)
```

```
-10
[5, 2, 100]
```

With strings:

[48]:
```
a, *b = 'python'
print(a)
print(b)
```

```
p
['y', 't', 'h', 'o', 'n']
```

What about extracting the first, second, last elements and *the rest*.

Again we can use slicing:

[49]:
```
s = 'python'

a, b, c, d = s[0], s[1], s[2:-1], s[-1]
print(a)
print(b)
print(c)
print(d)
```

```
p
y
tho
n
```

But we can just as easily do it this way using unpacking:

```
[50]: a, b, *c, d = s
      print(a)
      print(b)
      print(c)
      print(d)
```

```
p
y
['t', 'h', 'o']
n
```

As you can see though, **c** is a list of characters, not a string.

It that's a problem we can easily fix it this way:

```
[51]: print(c)
      c = ''.join(c)
      print(c)
```

```
['t', 'h', 'o']
tho
```

We can also use unpacking on the right hand side of an assignment expression:

```
[52]: l1 = [1, 2, 3]
      l2 = [4, 5, 6]
      l = [*l1, *l2]
      print(l)
```

```
[1, 2, 3, 4, 5, 6]
```

```
[53]: l1 = [1, 2, 3]
      s = 'ABC'
      l = [*l1, *s]
      print(l)
```

```
[1, 2, 3, 'A', 'B', 'C']
```

This unpacking works with unordered types such as sets and dictionaries as well.

The only thing is that it may not be very useful considering there is no particular ordering, so a first or last element has no real useful meaning.

```
[54]: s = {10, -99, 3, 'd'}
```

```
[55]: for c in s:
          print(c)
```

```
10
3
d
-99
```

As you can see, the order of the elements when we created the set was not retained!

```
[56]: s = {10, -99, 3, 'd'}
      a, b, *c = s
      print(a)
      print(b)
      print(c)
```

```
10
3
['d', -99]
```

So unpacking this way is of limited use.

However consider this:

```
[57]: s = {10, -99, 3, 'd'}
      *a, = s
      print(a)
```

```
[10, 3, 'd', -99]
```

At first blush, this doesn't look terribly exciting - we simply unpacked the set values into a list.

But this is actually quite useful in both sets and dictionaries to combine things (although to be sure, there are alternative ways to do this as well - which we'll cover later in this course)

```
[58]: s1 = {1, 2, 3}
      s2 = {3, 4, 5}
```

How can we combine both these sets into a single merged set?

```
[59]: s1 + s2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[59], line 1
----> 1 s1 + s2

TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Well, + doesn't work…

We could use the built-in method for unioning sets:

[60]:
```
help(set)
```

```
Help on class set in module builtins:

class set(object)
 |  set() -> new empty set object
 |  set(iterable) -> new set object
 |
 |  Build an unordered collection of unique elements.
 |
 |  Methods defined here:
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __contains__(…)
 |      x.__contains__(y) <==> y in x.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iand__(self, value, /)
 |      Return self&=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __ior__(self, value, /)
 |      Return self|=value.
 |
 |  __isub__(self, value, /)
 |      Return self-=value.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
```

```
|
|  __ixor__(self, value, /)
|      Return self^=value.
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __reduce__(…)
|      Return state information for pickling.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(…)
|      S.__sizeof__() -> size of S in memory, in bytes
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  add(…)
|      Add an element to a set.
```

```
|
|      This has no effect if the element is already present.
|
|  clear(…)
|      Remove all elements from this set.
|
|  copy(…)
|      Return a shallow copy of a set.
|
|  difference(…)
|      Return the difference of two or more sets as a new set.
|
|      (i.e. all elements that are in this set but not the others.)
|
|  difference_update(…)
|      Remove all elements of another set from this set.
|
|  discard(…)
|      Remove an element from a set if it is a member.
|
|      Unlike set.remove(), the discard() method does not raise
|      an exception when an element is missing from the set.
|
|  intersection(…)
|      Return the intersection of two sets as a new set.
|
|      (i.e. all elements that are in both sets.)
|
|  intersection_update(…)
|      Update a set with the intersection of itself and another.
|
|  isdisjoint(…)
|      Return True if two sets have a null intersection.
|
|  issubset(…)
|      Report whether another set contains this set.
|
|  issuperset(…)
|      Report whether this set contains another set.
|
|  pop(…)
|      Remove and return an arbitrary set element.
|      Raises KeyError if the set is empty.
|
|  remove(…)
|      Remove an element from a set; it must be a member.
|
|      If the element is not a member, raise a KeyError.
```

```
 |
 |  symmetric_difference(…)
 |      Return the symmetric difference of two sets as a new set.
 |
 |      (i.e. all elements that are in exactly one of the sets.)
 |
 |  symmetric_difference_update(…)
 |      Update a set with the symmetric difference of itself and another.
 |
 |  union(…)
 |      Return the union of sets as a new set.
 |
 |      (i.e. all elements that are in either set.)
 |
 |  update(…)
 |      Update a set with the union of itself and others.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(…) from builtins.type
 |      See PEP 585
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

[61]:
```
print(s1)
print(s2)
s1.union(s2)
```

```
{1, 2, 3}
{3, 4, 5}
```

[61]:
```
{1, 2, 3, 4, 5}
```

What about joining 4 different sets?

[62]:
```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
```

```
s3 = {5, 6, 7}
s4 = {7, 8, 9}
print(s1.union(s2).union(s3).union(s4))
print(s1.union(s2, s3, s4))
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Or we could use unpacking in this way:

[63]:
```
{*s1, *s2, *s3, *s4}
```

[63]:
```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

What we did here was to unpack each set directly into another set!

The same works for dictionaries - just remember that * for dictionaries unpacks the keys only.

[64]:
```
d1 = {'key1': 1, 'key2': 2}
d2 = {'key2': 3, 'key3': 3}
[*d1, *d2]
```

[64]:
```
['key1', 'key2', 'key2', 'key3']
```

So, is there anything to unpack the key-value pairs for dictionaries instead of just the keys?

Yes - we can use the ** operator:

[65]:
```
d1 = {'key1': 1, 'key2': 2}
d2 = {'key2': 3, 'key3': 3}

{**d1, **d2}
```

[65]:
```
{'key1': 1, 'key2': 3, 'key3': 3}
```

Notice what happened to the value of **key2**. The value for the second occurrence of **key2** was retained (overwritten).

In fact, if we write the unpacking reversing the order of d1 and d2:

[66]:
```
{**d2, **d1}
```

[66]:
```
{'key2': 2, 'key3': 3, 'key1': 1}
```

we see that the value of **key2** is now **2**, since it was the second occurrence.

Of course, we can unpack a dictionary into a dictionary as seen above, but we can mix in our own key-value pairs as well - it is just a dictionary literal after all.

[67]:
```
{'a': 1, 'b': 2, **d1, **d2, 'c':3}
```

```
[67]: {'a': 1, 'b': 2, 'key1': 1, 'key2': 3, 'key3': 3, 'c': 3}
```

Again, if we have the same keys, only the "latest" value of the key is retained:

```
[68]: {'key1': 100, **d1, **d2, 'key3': 200}
```

```
[68]: {'key1': 1, 'key2': 3, 'key3': 200}
```

Nested Unpacking Python even supports nested unpacking:

```
[69]: a, b, (c, d) = [1, 2, ['X', 'Y']]
      print(a)
      print(b)
      print(c)
      print(d)
```

```
1
2
X
Y
```

In fact, since a string is an iterable, we can even write:

```
[70]: a, b, (c, d) = [1, 2, 'XY']
      print(a)
      print(b)
      print(c)
      print(d)
```

```
1
2
X
Y
```

We can even write something like this:

```
[71]: a, b, (c, d, *e) = [1, 2, 'python']
      print(a)
      print(b)
      print(c)
      print(d)
      print(e)
```

```
1
2
p
y
['t', 'h', 'o', 'n']
```

Remember when we said that we can use a * only **once**...

How about this then?

[72]:
```python
a, *b, (c, d, *e) = [1, 2, 3, 'python']
print(a)
print(b)
print(c)
print(d)
print(e)
```

```
1
[2, 3]
p
y
['t', 'h', 'o', 'n']
```

We can break down what happened here in multiple steps:

[73]:
```python
a, *b, tmp = [1, 2, 3, 'python']
print(a)
print(b)
print(tmp)
```

```
1
[2, 3]
python
```

[74]:
```python
c, d, *e = tmp
print(c)
print(d)
print(e)
```

```
p
y
['t', 'h', 'o', 'n']
```

So putting it together we get our original line of code:

[75]:
```python
a, *b, (c, d, *e) = [1, 2, 3, 'python']
print(a)
print(b)
print(c)
print(d)
print(e)
```

```
1
[2, 3]
p
y
['t', 'h', 'o', 'n']
```

If we wanted to do the same thing using slicing:

[76]:
```
l = [1, 2, 3, 'python']
l[0], l[1:-1], l[-1][0], l[-1][1], list(l[-1][2:])
```

[76]:
```
(1, [2, 3], 'p', 'y', ['t', 'h', 'o', 'n'])
```

[77]:
```
l = [1, 2, 3, 'python']
a, b, c, d, e = l[0], l[1:-1], l[-1][0], l[-1][1], list(l[-1][2:])
print(a)
print(b)
print(c)
print(d)
print(e)
```

```
1
[2, 3]
p
y
['t', 'h', 'o', 'n']
```

Of course, this works for arbitrary lengths and indexable sequence types:

[78]:
```
l = [1, 2, 3, 4, 'unladen swallow']
a, b, c, d, e = l[0], l[1:-1], l[-1][0], l[-1][1], list(l[-1][2:])
print(a)
print(b)
print(c)
print(d)
print(e)
```

```
1
[2, 3, 4]
u
n
['l', 'a', 'd', 'e', 'n', ' ', 's', 'w', 'a', 'l', 'l', 'o', 'w']
```

or even:

[79]:
```
l = [1, 2, 3, 4, ['a', 'b', 'c', 'd']]
a, b, c, d, e = l[0], l[1:-1], l[-1][0], l[-1][1], list(l[-1][2:])
print(a)
print(b)
print(c)
print(d)
print(e)
```

```
1
[2, 3, 4]
a
```

```
b
['c', 'd']
```

[ ]:

```python

```

0.0.4 *args

Recall from iterable unpacking:

[80]:

```python
a, b, *c = 10, 20, 'a', 'b'
```

[81]:

```python
print(a, b)
```

```
10 20
```

[82]:

```python
print(c)
```

```
['a', 'b']
```

We can use a similar concept in function definitions to allow for arbitrary numbers of **positional** parameters/arguments:

[83]:

```python
def func1(a, b, *args):
    print(a)
    print(b)
    print(args)
```

[84]:

```python
func1(1, 2, 'a', 'b')
```

```
1
2
('a', 'b')
```

A few things to note:

1. Unlike iterable unpacking, ***args** will be a **tuple**, not a list.
2. The name of the parameter **args** can be anything you prefer
3. You cannot specify positional arguments **after** the ***args** parameter - this does something different that we'll cover in the next lecture.

[85]:

```python
def func1(a, b, *my_vars):
    print(a)
    print(b)
    print(my_vars)
```

[86]:

```python
func1(10, 20, 'a', 'b', 'c')
```

```
10
20
('a', 'b', 'c')
```

[87]:
```
def func1(a, b, *c, d):
    print(a)
    print(b)
    print(c)
    print(d)
```

[88]:
```
func1(10, 20, 'a', 'b', 100)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[88], line 1
----> 1 func1(10, 20, 'a', 'b', 100)

TypeError: func1() missing 1 required keyword-only argument: 'd'
```

Let's see how we might use this to calculate the average of an arbitrary number of parameters.

[89]:
```
def avg(*args):
    count = len(args)
    total = sum(args)
    return total/count
```

[90]:
```
avg(2, 2, 4, 4)
```

[90]:
```
3.0
```

But watch what happens here:

[91]:
```
avg()
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[91], line 1
----> 1 avg()

Cell In[89], line 4, in avg(*args)
      2 count = len(args)
      3 total = sum(args)
----> 4 return total/count

ZeroDivisionError: division by zero
```

The problem is that we passed zero arguments.

We can fix this in one of two ways:

```
[92]: def avg(*args):
          count = len(args)
          total = sum(args)
          if count == 0:
              return 0
          else:
              return total/count
```

```
[93]: avg(2, 2, 4, 4)
```

```
[93]: 3.0
```

```
[94]: avg()
```

```
[94]: 0
```

But we may not want to allow specifying zero arguments, in which case we can split our parameters into a required (non-defaulted) positional argument, and the rest:

```
[95]: def avg(a, *args):
          count = len(args) + 1
          total = a + sum(args)
          return total/count
```

```
[96]: avg(2, 2, 4, 4)
```

```
[96]: 3.0
```

```
[97]: avg()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[97], line 1
----> 1 avg()

TypeError: avg() missing 1 required positional argument: 'a'
```

As you can see, an exception occurs if we do not specify at least one argument.

Unpacking an iterable into positional arguments

```
[98]: def func1(a, b, c):
          print(a)
          print(b)
          print(c)
```

[99]:
```
l = [10, 20, 30]
```

This will **not** work:

[100]:
```
func1(l)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[100], line 1
----> 1 func1(l)

TypeError: func1() missing 2 required positional arguments: 'b' and 'c'
```

The function expects three positional arguments, but we only supplied a single one (albeit a list).

But we could unpack the list, and **then** pass it to as the function arguments:

[101]:
```
*l,
```

[101]:
```
(10, 20, 30)
```

[102]:
```
func1(*l)
```

```
10
20
30
```

What about mixing positional and keyword arguments with this?

[103]:
```
def func1(a, b, c, *d):
    print(a)
    print(b)
    print(c)
    print(d)
```

[104]:
```
func1(10, c=20, b=10, 'a', 'b')
```

```
  Cell In[104], line 1
    func1(10, c=20, b=10, 'a', 'b')
                                  ^
SyntaxError: positional argument follows keyword argument
```

Recall that once a keyword argument is used in a function call, we **cannot** use positional arguments after that.

However, in the next lecture we'll look at how to address this issue.

0.0.5 Keyword Arguments

Recall: positional parameters defined in functions can also be passed as named (keyword) arguments.

[105]:
```
def func1(a, b, c):
    print(a, b, c)
```

[106]:
```
func1(10, 20, 30)
```

```
10 20 30
```

[107]:
```
func1(b=20, c=30, a=10)
```

```
10 20 30
```

[108]:
```
func1(10, c=30, b=20)
```

```
10 20 30
```

Using a named argument is optional and up to the caller.

What if we wanted to force calls to our function to use named arguments?

We can do so by **exhausting** all the positional arguments, and then adding some additional parameters in teh function definition:

[109]:
```
def func1(a, b, *args, d):
    print(a, b, args, d)
```

Now we will need at least two positional arguments, an optional (possibly even zero) number of additional arguments, and this extra argument which is supposed to go into **d**. This argument can **only** be passed to the function using a named (keyword) argument:

So, this will not work:

[110]:
```
func1(10, 20, 'a', 'b', 100)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[110], line 1
----> 1 func1(10, 20, 'a', 'b', 100)

TypeError: func1() missing 1 required keyword-only argument: 'd'
```

But this will:

[111]:
```
func1(10, 20, 'a', 'b', d=100)
```

```
10 20 ('a', 'b') 100
```

As you can see, **d** took the keyword argument, while the remaining arguments were handled as positional parameters.

We can even define a function that has only optional positional arguments and mandatory keyword arguments:

[112]:
```
def func1(*args, d):
    print(args)
    print(d)
```

[113]:
```
func1(1, 2, 3, d='hello')
```

```
(1, 2, 3)
hello
```

We can of course, not pass any positional arguments:

[114]:
```
func1(d='hello')
```

```
()
hello
```

but the positional argument is mandatory (since no default was provided in the function definition):

[115]:
```
func1()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[115], line 1
----> 1 func1()

TypeError: func1() missing 1 required keyword-only argument: 'd'
```

To make the keyword argument optional, we just need to specify a default value in the function definition:

[116]:
```
def func1(*args, d='n/a'):
    print(args)
    print(d)
```

[117]:
```
func1(1, 2, 3)
```

```
(1, 2, 3)
n/a
```

[118]:
```
func1()
```

```
()
n/a
```

Sometimes we want **only** keyword arguments, in which case we still have to exhaust the positional arguments first - but we can use the following syntax if we do not want any positional parameters passed in:

[119]:
```
def func1(*, d='hello'):
    print(d)
```

[120]:
```
func1(10, d='bye')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[120], line 1
----> 1 func1(10, d='bye')

TypeError: func1() takes 0 positional arguments but 1 positional argument (and
 ↪keyword-only argument) were given
```

[121]:
```
func1(d='bye')
```

```
bye
```

Of course, if we do not provide a default value for the keyword argument, then we effectively are forcing the caller to provide the keyword argument:

[122]:
```
def func1(*, a, b):
    print(a)
    print(b)
```

[123]:
```
func1(a=10, b=20)
```

```
10
20
```

but, the following would not work:

[124]:
```
func1(10, 20)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[124], line 1
----> 1 func1(10, 20)

TypeError: func1() takes 0 positional arguments but 2 were given
```

Unlike positional parameters, keyword arguments do not have to be defined with non-defaulted and then defaulted arguments:

[125]:
```python
def func1(a, *, b='hello', c):
    print(a, b, c)
```

[126]:
```python
func1(5, c='bye')
```

```
5 hello bye
```

We can also include positional non-defaulted (first), positional defaulted (after positional non-defaulted) followed lastly (after exhausting positional arguments) by keyword args (defaulted or non-defaulted in any order)

[127]:
```python
def func1(a, b=20, *args, d=0, e='n/a'):
    print(a, b, args, d, e)
```

[128]:
```python
func1(5, 4, 3, 2, 1, d=0, e='all engines running')
```

```
5 4 (3, 2, 1) 0 all engines running
```

[129]:
```python
func1(0, 600, d='goooood morning', e='python!')
```

```
0 600 () goooood morning python!
```

[130]:
```python
func1(11, 'm/s', 24, 'mph', d='unladen', e='swallow')
```

```
11 m/s (24, 'mph') unladen swallow
```

As you can see, defining parameters and passing arguments is extremely flexible in Python! Even more so, when you account for the fact that the parameters are not statically typed!

In the next video, we'll look at one more thing we can do with function parameters!

0.0.6 **kwargs

[131]:
```python
def func(**kwargs):
    print(kwargs)
```

[132]:
```python
func(x=100, y=200)
```

```
{'x': 100, 'y': 200}
```

We can also use it in conjunction with ***args**:

[133]:
```python
def func(*args, **kwargs):
    print(args)
    print(kwargs)
```

[134]:
```python
func(1, 2, a=100, b=200)
```

```
(1, 2)
{'a': 100, 'b': 200}
```

Note: You cannot do the following:

[135]:
```
def func(*, **kwargs):
    print(kwargs)
```

```
  Cell In[135], line 1
    def func(*, **kwargs):
             ^
SyntaxError: named arguments must follow bare *
```

There is no need to even do this, since ****kwargs** essentially indicates no more positional arguments.

[136]:
```
def func(a, b, **kwargs):
    print(a)
    print(b)
    print(kwargs)
```

[137]:
```
func(1, 2, x=100, y=200)
```

```
1
2
{'x': 100, 'y': 200}
```

Also, you cannot specify parameters **after** ****kwargs** has been used:

[138]:
```
def func(a, b, **kwargs, c):
    pass
```

```
  Cell In[138], line 1
    def func(a, b, **kwargs, c):
                             ^
SyntaxError: arguments cannot follow var-keyword argument
```

If you want to specify both specific keyword-only arguments and ****kwargs** you will need to first get to a point where you can define a keyword-only argument (i.e. exhaust the positional arguments, using either ***args** or just *****)

[139]:
```
def func(*, d, **kwargs):
    print(d)
    print(kwargs)
```

[140]:
```
func(d=1, x=100, y=200)
```

```
1
{'x': 100, 'y': 200}
```

0.0.7 Putting it all Together

Positionals Only: no extra positionals, no defaults (all positionals required)

[141]:
```
def func(a, b):
    print(a, b)
```

[142]:
```
func('hello', 'world')
```

```
hello world
```

[143]:
```
func(b='world', a='hello')
```

```
hello world
```

Positionals Only: no extra positionals, defaults (some positionals optional)

[144]:
```
def func(a, b='world', c=10):
    print(a, b, c)
```

[145]:
```
func('hello')
```

```
hello world 10
```

[146]:
```
func('hello', c='!')
```

```
hello world !
```

Positionals Only: extra positionals, no defaults (all positionals required)

[147]:
```
def func(a, b, *args):
    print(a, b, args)
```

[148]:
```
func(1, 2, 'x', 'y', 'z')
```

```
1 2 ('x', 'y', 'z')
```

Note that we cannot call the function this way:

[149]:
```
func(b=2, a=1, 'x', 'y', 'z')
```

```
  Cell In[149], line 1
    func(b=2, a=1, 'x', 'y', 'z')
                                ^
SyntaxError: positional argument follows keyword argument
```

Keywords Only: no positionals, no defaults (all keyword args required)

[150]:
```python
def func(*, a, b):
    print(a, b)
```

[151]:
```python
func(a=1, b=2)
```

```
1 2
```

Keywords Only: no positionals, some defaults (not all keyword args required)

[152]:
```python
def func(*, a=1, b):
    print(a, b)
```

[153]:
```python
func(a=10, b=20)
```

```
10 20
```

[154]:
```python
func(b=2)
```

```
1 2
```

Keywords and Positionals: some positionals (no defaults), keywords (no defaults)

[155]:
```python
def func(a, b, *, c, d):
    print(a, b, c, d)
```

[156]:
```python
func(1, 2, c=3, d=4)
```

```
1 2 3 4
```

[157]:
```python
func(1, 2, d=4, c=3)
```

```
1 2 3 4
```

[158]:
```python
func(1, c=3, d=4, b=2)
```

```
1 2 3 4
```

Keywords and Positionals: some positional defaults

[159]:
```python
def func(a, b=2, *, c, d=4):
    print(a, b, c, d)
```

[160]:
```python
func(1, c=3)
```

```
1 2 3 4
```

[161]:
```python
func(c=3, a=1)
```

```
1 2 3 4
```

[162]:
```
func(1, 2, c=3, d=4)
```

```
1 2 3 4
```

[163]:
```
func(c=3, a=1, b=2, d=4)
```

```
1 2 3 4
```

Keywords and Positionals: extra positionals

[164]:
```
def func(a, b=2, *args, c=3, d):
    print(a, b, args, c, d)
```

[165]:
```
func(1, 2, 'x', 'y', 'z', c=3, d=4)
```

```
1 2 ('x', 'y', 'z') 3 4
```

Note that if we are going to use the extra arguments, then we cannot actually use a default value for b:

[166]:
```
func(1, 'x', 'y', 'z', c=3, d=4)
```

```
1 x ('y', 'z') 3 4
```

as you can see, **b** was assigned the value **x**

Keywords and Positionals: no extra positionals, extra keywords

[167]:
```
def func(a, b, *, c, d=4, **kwargs):
    print(a, b, c, d, kwargs)
```

[168]:
```
func(1, 2, c=3, x=100, y=200, z=300)
```

```
1 2 3 4 {'x': 100, 'y': 200, 'z': 300}
```

[169]:
```
func(x=100, y=200, z=300, c=3, b=2, a=1)
```

```
1 2 3 4 {'x': 100, 'y': 200, 'z': 300}
```

Keywords and Positionals: extra positionals, extra keywords

[170]:
```
def func(a, b, *args, c, d=4, **kwargs):
    print(a, b, args, c, d, kwargs)
```

[171]:
```
func(1, 2, 'x', 'y', 'z', c=3, d=5, x=100, y=200, z=300)
```

```
1 2 ('x', 'y', 'z') 3 5 {'x': 100, 'y': 200, 'z': 300}
```

Keywords and Positionals: only extra positionals and extra keywords

[172]:
```
def func(*args, **kwargs):
    print(args, kwargs)
```

[173]:
```
func(1, 2, 3, x=100, y=200, z=300)
```

```
(1, 2, 3) {'x': 100, 'y': 200, 'z': 300}
```

The Print Function

[174]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

[175]:
```
print(1, 2, 3)
```

```
1 2 3
```

[176]:
```
print(1, 2, 3, sep='--')
```

```
1--2--3
```

[177]:
```
print(1, 2, 3, end='***\n')
```

```
1 2 3***
```

[178]:
```
print(1, 2, 3, sep='\t', end='\t***\t')
print(4, 5, 6, sep='\t', end='\t***\n')
```

```
1       2       3       ***     4       5       6       ***
```

Another Use Case

[179]:
```
def calc_hi_lo_avg(*args, log_to_console=False):
    hi = int(bool(args)) and max(args)
    lo = int(bool(args)) and min(args)
    avg = (hi + lo)/2
    if log_to_console:
        print("high={0}, low={1}, avg={2}".format(hi, lo, avg))
    return avg
```

[180]:
```
avg = calc_hi_lo_avg(1, 2, 3, 4, 5)
print(avg)
```

```
3.0
```

[181]:
```
avg = calc_hi_lo_avg(1, 2, 3, 4, 5, log_to_console=True)
print(avg)
```

```
high=5, low=1, avg=3.0
3.0
```

0.0.8 A Simple Function Timer

We want to create a simple function that can time how fast a function runs.

We want this function to be generic in the sense that it can be used to time any function (along with it's positional and keyword arguments), as well as specifying the number of the times the function should be timed, and the returns the average of the timings.

We'll call our function **time_it**, and it will need to have the following parameters:

- the function we want to time
- the positional arguments of the function we want to time (if any)
- the keyword-only arguments of the function we want to time (if any)
- the number of times we want to run this function

[182]:
```
import time
```

[183]:
```
def time_it(fn, *args, rep=5, **kwargs):
    print(args, rep, kwargs)
```

Now we could the function this way:

[184]:
```
time_it(print, 1, 2, 3, sep='-')
```

```
(1, 2, 3) 5 {'sep': '-'}
```

Let's modify our function to actually run the print function with any positional and keyword args (except for rep) passed to it:

[185]:
```
def time_it(fn, *args, rep=5, **kwargs):
    for i in range(rep):
        fn(*args, **kwargs)
```

[186]:
```
time_it(print, 1, 2, 3, sep='-')
```

```
1-2-3
1-2-3
1-2-3
1-2-3
1-2-3
```

As you can see **1, 2, 3** was passed to the **print** function's positional parameters, and the keyword_only arg **sep** was also passed to it.

We can even add more arguments:

[187]:
```
time_it(print, 1, 2, 3, sep='-', end=' *** ', rep=3)
```

```
1-2-3 *** 1-2-3 *** 1-2-3 ***
```

Now all that's really left for us to do is to time the function and return the average time:

[188]:
```
def time_it(fn, *args, rep=5, **kwargs):
    start = time.perf_counter()
    for i in range(rep):
        fn(*args, **kwargs)
    end = time.perf_counter()
    return (end - start) / rep
```

Let's write a few functions we might want to time:

We'll create three functions that all do the same thing: calculate powers of n**k for k in some range of integer values

[189]:
```
def compute_powers_1(n, *, start=1, end):
    # using a for loop
    results = []
    for i in range(start, end):
        results.append(n**i)
    return results
```

[190]:
```
def compute_powers_2(n, *, start=1, end):
    # using a list comprehension
    return [n**i for i in range(start, end)]
```

[191]:
```
def compute_powers_3(n, *, start=1, end):
    # using a generator expression
    return (n**i for i in range(start, end))
```

Let's run these functions and see the results:

[192]:
```
compute_powers_1(2, end=5)
```

[192]:
```
[2, 4, 8, 16]
```

[193]:
```
compute_powers_2(2, end=5)
```

[193]:
```
[2, 4, 8, 16]
```

[194]:
```
list(compute_powers_3(2, end=5))
```

```
[194]: [2, 4, 8, 16]
```

Finally let's run these functions through our **time_it** function and see the results:

```
[195]: time_it(compute_powers_1, n=2, end=20000, rep=4)
```

```
[195]: 0.38889572500011127
```

```
[196]: time_it(compute_powers_2, 2, end=20000, rep=4)
```

```
[196]: 0.3628432250006881
```

```
[197]: time_it(compute_powers_3, 2, end=20000, rep=4)
```

```
[197]: 1.7500005924375728e-06
```

Although the **compute_powers_3** function appears to be **much** faster than the other two, it doesn't quite do the same thing!

We'll cover generators in detail later in this course.

```
[ ]:
```

0.0.9 Default Values - Beware!

```
[198]: from datetime import datetime
```

```
[199]: print(datetime.utcnow())
```

```
2023-06-14 15:46:01.652895
```

```
[200]: def log(msg, *, dt=datetime.utcnow()):
           print('{0}: {1}'.format(dt, msg))
```

```
[201]: log('message 1')
```

```
2023-06-14 15:46:01.657710: message 1
```

```
[202]: log('message 2', dt='2001-01-01 00:00:00')
```

```
2001-01-01 00:00:00: message 2
```

```
[203]: log('message 3')
```

```
2023-06-14 15:46:01.657710: message 3
```

```
[204]: log('message 4')
```

```
2023-06-14 15:46:01.657710: message 4
```

As you can see, the default for **dt** is calculated when the function is **defined** and is **NOT** re-evaluated when the function is called.

Solution Pattern Here is one pattern we can use to achieve the desired result:

We actually set the default to None - this makes the argument optional, and we can then test for None **inside** the function and default to the current time if it is None.

[205]:
```
def log(msg, *, dt=None):
    dt = dt or datetime.utcnow()
    # above is equivalent to:
    #if not dt:
    #    dt = datetime.utcnow()
    print('{0}: {1}'.format(dt, msg))
```

[206]:
```
log('message 1')
```

```
2023-06-14 15:46:01.689504: message 1
```

[207]:
```
log('message 2')
```

```
2023-06-14 15:46:01.694657: message 2
```

[208]:
```
log('message 3', dt='2001-01-01 00:00:00')
```

```
2001-01-01 00:00:00: message 3
```

[209]:
```
log('message 4')
```

```
2023-06-14 15:46:01.705039: message 4
```

0.0.10 Parameter Defaults - Beware 2

Another gotcha with parameter defaults comes with mutable types, and is an easy trap to fall into.

Again, you have to remember that function parameter defaults are evaluated once, when the function is defined (i.e. when the module is loaded, or in this Jupyter notebook, when we "execute" the function definition), and not every time the function is called.

Consider the following scenario.

We are creating a grocery list, and we want our list to contain consistently formatted data with name, quantity and measurement unit:

`bananas (2 units) grapes (1 bunch) milk (1 liter) python (1 medium-rare)`

To make sure the data is consistent, we want to use a function that we can call to add the item to our list.

So we'll need to provide it our current grocery list as well as the item information to be added:

```
[210]: def add_item(name, quantity, unit, grocery_list):
           item_fmt = "{0} ({1} {2})".format(name, quantity, unit)
           grocery_list.append(item_fmt)
           return grocery_list
```

We have two stores we want to visit, so we set up two grocery lists:

```
[211]: store_1 = []
       store_2 = []
```

```
[212]: add_item('bananas', 2, 'units', store_1)
       add_item('grapes', 1, 'bunch', store_1)
       add_item('python', 1, 'medium-rare', store_2)
```

```
[212]: ['python (1 medium-rare)']
```

```
[213]: store_1
```

```
[213]: ['bananas (2 units)', 'grapes (1 bunch)']
```

```
[214]: store_2
```

```
[214]: ['python (1 medium-rare)']
```

Ok, working great. But let's make the function a little easier to use - if the user does not supply an existing grocery list to append the item to, let's just go ahead and default our `grocery_list` to an empty list hence starting a new shopping list:

```
[215]: def add_item(name, quantity, unit, grocery_list=[]):
           item_fmt = "{0} ({1} {2})".format(name, quantity, unit)
           grocery_list.append(item_fmt)
           return grocery_list
```

```
[216]: store_1 = add_item('bananas', 2, 'units')
       add_item('grapes', 1, 'bunch', store_1)
```

```
[216]: ['bananas (2 units)', 'grapes (1 bunch)']
```

```
[217]: store_1
```

```
[217]: ['bananas (2 units)', 'grapes (1 bunch)']
```

OK, so that seems to be working as expected.

Let's start our second list:

```
[218]: store_2 = add_item('milk', 1, 'gallon')
```

```
[219]: print(store_2)
```

```
['bananas (2 units)', 'grapes (1 bunch)', 'milk (1 gallon)']
```

??? What's going on? Our second list somehow contains the items that are in the first list.

What happened is that the returned value in the first call we made was the default grocery list - but remember that the list was created once and for all when the function was **created** not called. So everytime we call the function, that is the **same** list being used as the default.

When we started out first list, we were adding item to that default list.

When we started our second list, we were adding items to the **same** default list (since it is the same object).

We can avoid this problem using the same pattern as in the previous example we had with the default date time value. We use None as a default value instead, and generate a new empty list (hence starting a new list) if none was provided.

[220]:
```
def add_item(name, quantity, unit, grocery_list=None):
    if not grocery_list:
        grocery_list = []
    item_fmt = "{0} ({1} {2})".format(name, quantity, unit)
    grocery_list.append(item_fmt)
    return grocery_list
```

[221]:
```
store_1 = add_item('bananas', 2, 'units')
add_item('grapes', 1, 'bunch', store_1)
```

[221]:
```
['bananas (2 units)', 'grapes (1 bunch)']
```

[222]:
```
store_2 = add_item('milk', 1, 'gallon')
store_2
```

[222]:
```
['milk (1 gallon)']
```

Issue resolved!

However, there are legitimate use cases (well, almost legitimate, often we're better off using a different approach that we'll see when we look at closures), but here's a simple one.

We want our function to cache results, so that we don't recalculate something more than once.

Let's say we have a factorial function, that can be defined recursively as:

`n! = n * (n-1)!`

[223]:
```
def factorial(n):
    if n < 1:
        return 1
    else:
        print('calculating {0}!'.format(n))
        return n * factorial(n-1)
```

[224]:
```python
factorial(3)
```

```
calculating 3!
calculating 2!
calculating 1!
```

[224]:
```
6
```

[225]:
```python
factorial(3)
```

```
calculating 3!
calculating 2!
calculating 1!
```

[225]:
```
6
```

As you can see we had to recalculate all those factorials the second time around.

Let's cache the results leveraging what we saw in the previous example:

[226]:
```python
def factorial(n, cache={}):
    if n < 1:
        return 1
    elif n in cache:
        return cache[n]
    else:
        print('calculating {0}!'.format(n))
        result = n * factorial(n-1)
        cache[n] = result
        return result
```

[227]:
```python
factorial(3)
```

```
calculating 3!
calculating 2!
calculating 1!
```

[227]:
```
6
```

[228]:
```python
factorial(3)
```

[228]:
```
6
```

Now as you can see, the second time around we did not have to recalculate all the factorials. In fact, to calculate higher factorials, you'll notice that we don't need to re-run *all* the recursive calls:

[229]:
```python
factorial(5)
```

```
calculating 5!
calculating 4!
```

[229]:
```
120
```

`5!` and `4!` was calculated since they weren't cached, but since `3!` was already cached we didn't have to recalculate it - it was a quick lookup instead.

This technique is something called memoization, and we'll come back to it in much more detail when we discuss closures and decorators.

[ ]:
```

```