Passing arguments to Python functions

[I wrote this in 2006, while developing the Halalang Marangal (HALAL) software for processing election reports sent in as text messages. It may still be useful for Python programmers. I made minor changes for clarity. For the non-technical reader: Python is a programming language like BASIC, Fortran, Pascal, C, Java, etc. It was my language of choice when I was looking for a language to implement the software I'm referring to above.]

I initially found it hard to understand the use of * and ** in python functions. Now I think I do. I'd like to share my understanding with other beginners who might be struggling with these concepts.

What is the difference between the following definitions?

```
def nostar(a): print a

def onestar(*a): print a

def twostar(**a): print a
```

I was curious myself so I loaded the python interpreter and tried the following:

```
>>> nostar(1)

1
```

As expected. Formal parameter a received the value 1.

```
>>> nostar(1,2,3)

Traceback (most recent call last):

File "", line 1, in ?

TypeError: nostar() takes exactly 1 argument (3 given)
```

Of course. Nostar() only expects one argument (a). Sending it more causes an error.

```
>>> onestar(1,2,3)

(1, 2, 3)
```

Hmmm. Onestar(*a) apparently means: all (i.e., *) arguments go to the argument a. So, the form “def f(*a)” allows one to define functions with a variable number of arguments. Ok. Let’s now look at twostar(**a).

>>> twostar(1,2,3)

Traceback (most recent call last):

File “”, line 1, in ?

TypeError: twostar() takes exactly 0 arguments (3 given)

What is this? The formal argument **a, apparently means the function expects no arguments at all. What does it expect then?

>>> twostar(a=1,b=2,c=3)

{‘a’: 1, ‘c’: 3, ‘b’: 2}

So, functions defined using the form “def f(**a)” expect key/value pairs (i.e., **), which are then passed on to the receiving function as a dictionary.

To summarize: “def f(a)” is the standard format for defining a function, which is the called by passing it one argument. “def f(*a)” is useful for defining functions with a variable number of arguments, which are then passed as a single tuple to the receiving function. “def f(**a)” is useful for passing key/value pairs, not single arguments, which are then passed on as a dictionary to the receiving function. I think I get it.

By the way, the three forms can be combined as follows:

def f(a,*b,**c):

All single arguments beyond the first will end up with the tuple b, and all key/value arguments will end up in dictionary c. For example, f(1,2,3,4,5,d=6,e=7,f=8) should put 1 in a, (2,3,4,5) in b, and {‘d’:6,’e':7,’f':8} in c. Let’s see if it does. Let’s define f first:

>>> def f(a,*b,**c):

… print a,b,c

…

Now, let’s call the function:

>>> f(1,2,3,4,5,d=6,e=7,f=8)

1 (2, 3, 4, 5) {‘e’: 7, ‘d’: 6, ‘f’: 8}

Right. Note that the dictionary printed its contents an order different from the order the arguments were passed. Dictionaries do that. Unless you sort them, you can’t predict the order of the contents. (So, they shouldn’t have been called dictionaries, right?)

We’re not done yet. What is the difference between the following calls?

test(x)

test(*x)

test(**x)

This is what I learned:

test(x) # the function test must be defined with one argument. For example, def test(a):

test(*x) # x must be a sequence (list, tuple, etc.) with as many items as the arguments in the function definition

test(**x) # x must be a dictionary of key/value pairs, with as many pairs as the arguments in the function definition

Ok. Let’s try it out. Let’s define test:

>>> def test(a,b,c): print a,b,c

…

>>> test(1,2,3)

1 2 3

As expected, I called test with 3 arguments and it printed the three. What if x is a sequence (a list or a tuple)? Let’s see:

>>> x=(1,2,3)

>>> test(x)

Traceback (most recent call last):

File “”, line 1, in ?

TypeError: test() takes exactly 3 arguments (1 given)

Again, as expected. We fed a function that expects 3 arguments with only one argument (a tuple with 3 items), and we got an error. If we want to use the items in the sequence, we use the following form:

>>> test(*x)

1 2 3

There, x was split up, and its members used as the arguments to test. What about the third form?

>>> test(**x)

Traceback (most recent call last):

File “”, line 1, in ?

TypeError: test() argument after ** must be a dictionary

It returned an error, because ** always refers to key/value pairs, i.e., dictionaries. Let’s define a dictionary then:

>>> x={‘a’:1,’b':2,’c':3}

>>> test(**x)

1 2 3

>>> test(*x)

a c b

Ok. The first call passed on the values in the dictionary. The second call passed on the keys (but in wrong order!). Remember, ** is for dictionaries, * is for lists or tuples.

Enough.