Unpacking operator in code and function header

Unpacking

In Python you are allowed to do deconstruction similar to how you can do deconstruction assignment in JavaScript. You can take a list and then assign each of the elements individually to variables on the left hand. For example:

```
(a, b, c) = (1, 2, 3)
# a = 1
# b = 2
# c = 3
```

If you are using this simple unpacking assignment then the values on the right hand side will be automatically assigned to the variables to the left according to their position in the tuple.

When you unpack values into variables the number of variables present must be exactly equal to the number of elements in the tuple, else ValueError

You can use this tuple unpacking to unpack a String into characters, unpack list into variables, unpack a generator, and unpack a dictionary as well.

The left hand side can be a list as well but is rarely used.

* Unpacking operator

The * operator extends the unpacking tuple functionality to allow you to pack multiple values into a single variable as a list. For example:

```
*a, = [1, 2]
# a = [1, 2]
```

In this example, the variable a gets stuffed two values 1 and 2 into it as a list. Notice the usage of comma there, it is required otherwise it will be a normal list assignment statement and not unpacking.

You can have as many variables as you want on the left hand side, but there can only be one starred variable. The starred variable contains all the rest of the values that are not distributed to the mandatory variable in the list, even if it has no value. If there are no element left to distribute to the starred variable, then the starred variable will just be an empty list:

```
a, b, *c = [1, 2]
# a = 1
# b = 2
# c = []
```

Function header with *args and **kwargs

Sometimes in Python function header you will see functions that takes in variable number of position arguments with *, and variable number of keyword arguments with **. For example:

```
def func(required, *args, **kwargs)
```

`args` and `kwargs` are both optional, and are automatically default to `() and {}` respectively, if none are provided for the function.

Calling the function above would be like so:

```
func("hello", 1, 2, 3, site="hi", oh="lord")
# required = "hello"
# args = (1, 2, 3)
# kwargs = {"site": "hi", "oh": "lord"}
```

Side note on method headers

1. Variables without default value must be before those that have default values

   ```
   def test_func(a=10, b, *args, **kwargs)
   ```

   Is not valid Python function
2. `*args` must be fore `**kwargs`. They cannot switch, otherwise, invalid Python syntax
3. Follow the structure of `func`, don't try to put position arguments after `*args`.

Revision #4
Created 2023-04-30 16:53:28 UTC by Tamarine
Updated 2023-04-30 18:07:39 UTC by Tamarine