

# Variable Arguments & Argument Unpacking

# The basics

I'll assume you know how to define functions with default arguments.

```
>>> def foo(a, b, x=3, y=2):  
...     return (a+b)/(x+y)  
...  
>>> foo(5, 0)  
1.0  
>>> foo(10, 2, y=3)  
2.0  
>>> foo(b=4, x=3, a=1)  
1.0
```

# Accepting 0 or more arguments

Sometimes you want to define a function that can take any number of arguments. Python's syntax for doing that looks like this:

```
# Note the asterisk. That's the magic part
def takes_any_args(*args):
    print("Type of args: " + str(type(args)))
    print("Value of args: " + str(args))
```

```
>>> takes_any_args(1)
Type of args: <class 'tuple'>
Value of args: (1,)
>>> takes_any_args("x", "y", "z")
Type of args: <class 'tuple'>
Value of args: ('x', 'y', 'z')
>>> takes_any_args()
Type of args: <class 'tuple'>
Value of args: ()
```

# Stored in tuple

Within the function, args is a tuple:

```
def takes_any_args(*args):  
    print("Type of args: " + str(type(args)))  
    print("Value of args: " + str(args))
```

```
>>> takes_any_args(5, 4, 3, 2, 1)  
Type of args: <class 'tuple'>  
Value of args: (5, 4, 3, 2, 1)  
>>> takes_any_args(["first", "list"], ["another", "list"])  
Type of args: <class 'tuple'>  
Value of args: (['first', 'list'], ['another', 'list'])
```

# Single Argument vs. \*args

```
>>> def takes_any_args(*args):  
...     print("Type of args: " + str(type(args)))  
...     print("Value of args: " + str(args))  
...  
>>> def takes_a_list(items):  
...     print("Type of items: " + str(type(items)))  
...     print("Value of items: " + str(items))  
...  
>>> data = ["x", "y", "z"]  
>>> takes_a_list(data)  
Type of items: <class 'list'>  
Value of items: ['x', 'y', 'z']  
>>> takes_any_args(data)  
Type of args: <class 'tuple'>  
Value of args: (['x', 'y', 'z'],)
```

# \*args

By convention, the tuple argument's default name is `args`. But it doesn't have to be.

```
def read_files(*paths):  
    data = ""  
    for path in paths:  
        with open(path) as handle:  
            data += handle.read()  
    return data  
  
# ch1.txt has text of Chapter 1; ch2.txt for Ch. 2, etc.  
story = read_files("ch1.txt", "ch2.txt", "ch3.txt", "ch4.txt")
```



# Quick exercise

Open a file named `varargs1.py` and type in the following:

```
def print_args(*args):  
    for arg in args:  
        print(arg)  
  
print_args("red", "blue", "green")
```

You should see this output, one color per line:

```
red  
blue  
green
```

**Extra credit:** Can you find a way to call `print_args`, with different arguments, that triggers an error?

# Variable Keyword Arguments

Keyword arguments can't be captured by the `*args` idiom:

```
def print_args(*args):  
    for arg in args:  
        print(arg)
```

```
>>> print_args(a=4, b=7)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: print_args() got an unexpected keyword argument 'a'
```

What do we do?



# kwargs

For keyword arguments, use `**` syntax:

```
def print_kwargs(**kwargs):  
    for key, value in kwargs.items():  
        print("{} -> {}".format(key, value))
```

kwargs is a **dictionary**.

```
>>> print_kwargs(hero="Homer", antihero="Bart", genius="Lisa")  
hero -> Homer  
genius -> Lisa  
antihero -> Bart
```

Notice this is normal Python syntax for calling a function. Has nothing to do with `**kwargs`.

# Combine them

Combine them together:

```
def print_all(*args, **kwargs):  
    for arg in args:  
        print(arg)  
    for key, value in kwargs.items():  
        print("{} -> {}".format(key, value))
```

A defined function can use either `*args`, or `**kwargs`, or both.

# Some notes...

- `args` and `kwargs` are conventional names. Use them unless it's more readable to do something different.
- Always be clear on the types:
  - `args` is a tuple, in the same order as passed in
  - `kwargs` is a dictionary, unordered

# Quick exercise

Create a file named `varargs2.py` and type in the following (on the left). When run, it should print the output on the right.

varargs2.py source	program output
<pre>def print_all(*args, **kwargs):     for arg in args:         print(arg)     for key, value in kwargs.items():         print("{} -&gt; {}".format(key, value))  print_all("dog", "cat") print_all(x=7, z=3) print_all("red", "green", x=7, z=3)</pre>	<pre>dog cat z -&gt; 3 x -&gt; 7 red green z -&gt; 3 x -&gt; 7</pre>

EXTRA CREDIT: How can you write a function taking from 0 to 4 arguments (but no more)? How about 0 to 20? What are the different approaches and trade-offs?

# Positional + kwargs

```
def add_to_dict(stuff, **kwargs):  
    for key, value in kwargs.items():  
        # Do not overwrite existing values.  
        if key not in stuff:  
            stuff[key] = value  
    return stuff
```

```
>>> add_to_dict({})  
{}  
>>> add_to_dict({}, foo=42)  
{'foo': 42}  
>>> add_to_dict({"foo": 42}, bar=21)  
{'bar': 21, 'foo': 42}  
>>> add_to_dict({"foo": 42}, foo=21)  
{'foo': 42}
```

# Another problem

Suppose library A defines this function:

```
def order_book(title, author, isbn):  
    """  
    Place an order for a book.  
    """  
    print("Ordering '{}' by {} ({}).format(title, author, isbn))  
    # ...
```

... and library B defines this one:

```
def get_required_textbook(class_id):  
    """  
    Returns a tuple (title, author, ISBN)  
    """  
    # ...
```



# Incompatible types

How do you integrate these two?

You could do this, which is pretty horrible:

```
>>> book_info = get_required_textbook(4242)
>>> order_book(book_info[0], book_info[1], book_info[2])
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

Tedious and error-prone. What's a better way?

# Better: Arg Unpacking

Python provides a better way.

```
>>> def normal_function(a, b, c):  
...     print("a: {} b: {} c: {}".format(a,b,c))  
...  
>>> numbers = (7, 5, 3)  
>>> normal_function(*numbers)  
a: 7 b: 5 c: 3  
>>> # Exactly equivalent to:  
... normal_function(numbers[0], numbers[1], numbers[2])  
a: 7 b: 5 c: 3
```

Notice `normal_function` is just a regular function! This is called **argument unpacking**.

# Argument unpacking

Given these:

```
one_args = [ 42 ]  
two_args = (7, 10)  
three_args = [1, 2, 3]  
  
def f(n): return n / 2  
def g(a, b): return a + b  
def h(x, y, z): return x * y * z
```

The following pairs are all equivalent:

```
f(one_args[0])  
f(*one_args)  
  
g(two_args[0], two_args[1])  
g(*two_args)  
  
h(three_args[0], three_args[1], three_args[2])  
h(*three_args)
```

# Back to the books...

So instead of this....

```
>>> book_info = get_required_textbook(4242)
>>> order_book(book_info[0], book_info[1], book_info[2])
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

We can do this:

```
>>> book_info = get_required_textbook(4242)
>>> order_book(*book_info)
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

# Keyword Unpacking

Just like with `*args`, double-star works the other way too. We can take a regular function, and pass it a dictionary using two asterisks:

```
>>> def normal_function(a, b, c):  
...     print("a: {} b: {} c: {}".format(a,b,c))  
...  
>>> numbers = {"a": 7, "b": 5, "c": 3}  
>>> normal_function(**numbers)  
a: 7 b: 5 c: 3  
>>> # Exactly equivalent to:  
... normal_function(a=numbers["a"], b=numbers["b"], c=numbers["c"])  
a: 7 b: 5 c: 3
```

# Matching Keys

Keys of the dictionary *must* match up with how the function was declared:

```
>>> bad_numbers = {"a": 7, "b": 5, "z": 3}
>>> normal_function(**bad_numbers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: normal_function() got an unexpected keyword argument 'z'
```



# Calling both

You can call a function using both - and both will be unpacked:

```
>>> def addup(a, b, c=1, d=2, e=3):  
...     return a + b + c + d + e  
...  
>>> nums = (3, 4)  
>>> extras = {"d": 5, "e": 2}  
>>> addup(*nums, **extras)  
15
```

# Two different things

Python uses `*` and `**` for two *very different* things:

- Variable arguments (when *defining* a function), and
- Argument unpacking (when *calling* a function).

These look similar in code. **But they are completely different things.**

# Lab: Variable Arguments

Lab file: `functions/varargs.py`

- In labs/py3 for 3.x; labs/py2 for 2.7

When you finish:

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `functions/varargs_extra.py`

**When that's done:** Open and skim through **PythonTheNextLevel.pdf** - just notice what topics interest you.