## Let's have some fun this beat is sick

Functions are used extensively in Python.

The basic function definition is shown here:

You choose this name

you choose the name of the arguments

```
def function_name (arg):
    return function_value
```

mandatory def keyword

the function calculates and returns this value

you should always use return so the value your function returns is defined

the function body is indented.

After you define the function, you can call it like this:

function_name (somearg) $\Rightarrow$ function_value

This is a mandatory, positional argument because no default value is defined in the function definition

What is a positional argument?

It is an argument defined by its position.

For example with this function:

```
def f(a, b):
    return a+b
```

if we call it as f(1, 2)

this means a=1 and b=2

because 1 is first in the call and a is first in the definition.

In otherwords, the arguments are <u>assigned</u> by their position when the function is called.

When we call a function without specifying the argument names, the arguments are assigned in the order provided.

## Using argument names

here is one two other ways to call the function where we name the arguments. These are called keyword arguments. Since we name every argument, the order does not matter as it is unambiguous what the arguments are.

```
f(a=1, b=2)        or        f(b=2, a=1)
```

## Mixing positional and keyword arguments

It is possible to do both, but positional arguments must come first, and then the keyword arguments.

```
f(1, b=2)    is ok.
```

```
f(b=2, 1)    is not ok. A positional argument comes after
                 a keyword argument, which is not allowed.
keyword  positional
```

## What about arbitrary numbers of positional arguments?

It is an error to use more arguments than a function is defined for.

```
def f(a, b):
    return a+b
```

f(1, 2, 3) ⇒ ERROR!! 3 arguments used for function defined with two arguments.

There is a syntax to define a function that can take an arbitrary number of arguments:

you choose the name here, and prefix it with *

```
def f(*args):
    return sum(args)
```

In this function all the positional args you provide are stored as a tuple named args.

f(1,2) ⇒ args = (1,2) returns 3

f(1,2,3) ⇒ args = (1, 2,3) returns 6

An alternative to using *args is to use a tuple or list as the single argument.

```
def g(args):
    return sum(args)
```

f(1,2) vs g([1, 2])

f(1, 2, 3) vs g([1, 2, 3])

## What about optional arguments?

You can have optional arguments, if you define them as keyword arguments with a default value.

x: mandatory positional argument

a=3: optional keyword argument that defaults to 3

```
def f(x, a=3):
    return x**a
```

$f(3) = 27$ — a is not provided so it defaults to 3

$f(3, 1) = 3$ — a=1

$f(x=3, a=1) = 3$ — using keywords

## I want to allow arbitrary keyword arguments too!

You can do this with **kwargs.

mandatory positional arg
optional arg
additional position args in tuple
additional kwargs in dictionary

```
def f(a, b=2, *args, **kwargs):
    return something
```

you choose this name

The extra kwargs will be a dictionary of keyword: value pairs you can use in your function.

so here: f(1, 3, 5, reduce=True) leads to
a=1
b=3
args = (5,)
kwargs = {'reduce' : True}

you can use these variables inside your function.

This is commonly used to pass keyword arguments to other functions inside a function. Suppose you use solve_ivp in a function, but you want to pass arguments to it.

```
def f(mu):
    sol = solve_ivp(ode, tspan, y0, args=(mu,))
    return sol.y
```

we can't pass any keyword args to solve_ivp here without rewriting the function

Instead use this:

```
def f(mu, **kwargs):
    sol = solve_ivp(ode, tspan, y0, args=(mu,), **kwargs)
    return sol.y
```

this unpacks the dictionary into keyword arguments.

Now, you can call your function as:

f(0.2, max_step = 0.5) and max_step=0.5 will be passed into the solve_ivp call!