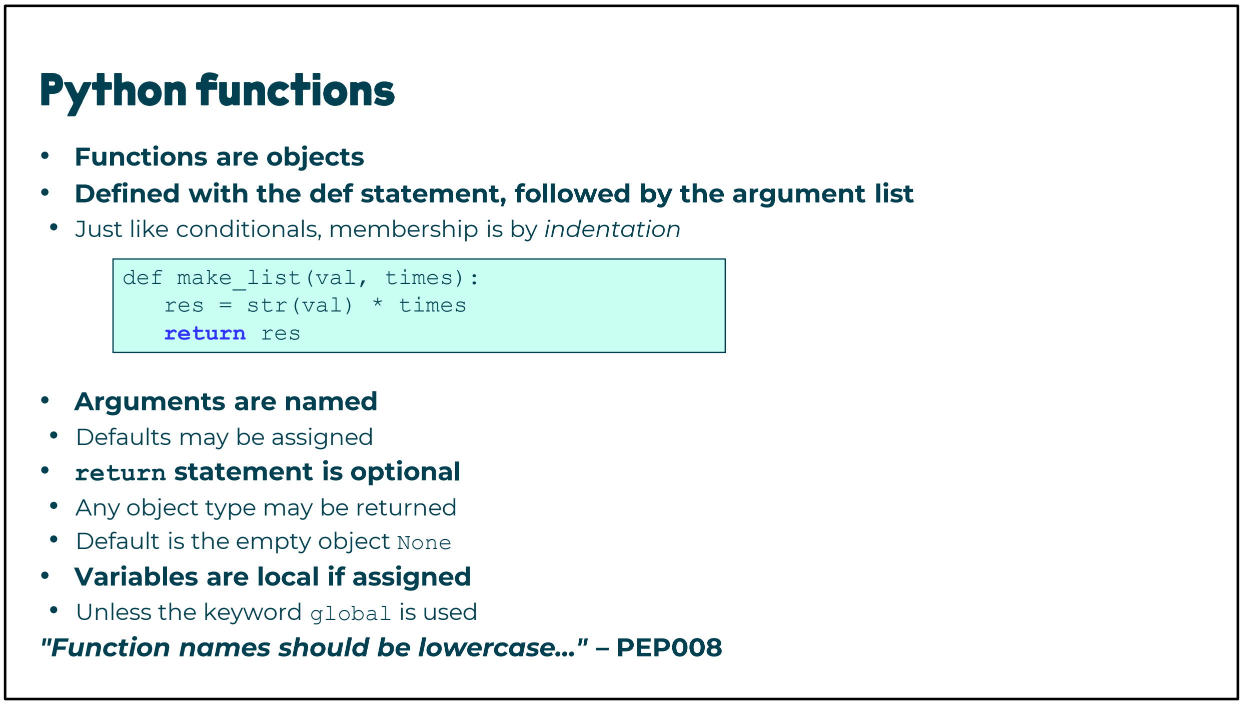
POWERING
POTENTIAL

Python 3 Programming

Functions

FUNCTIONS

Contents



Python functions

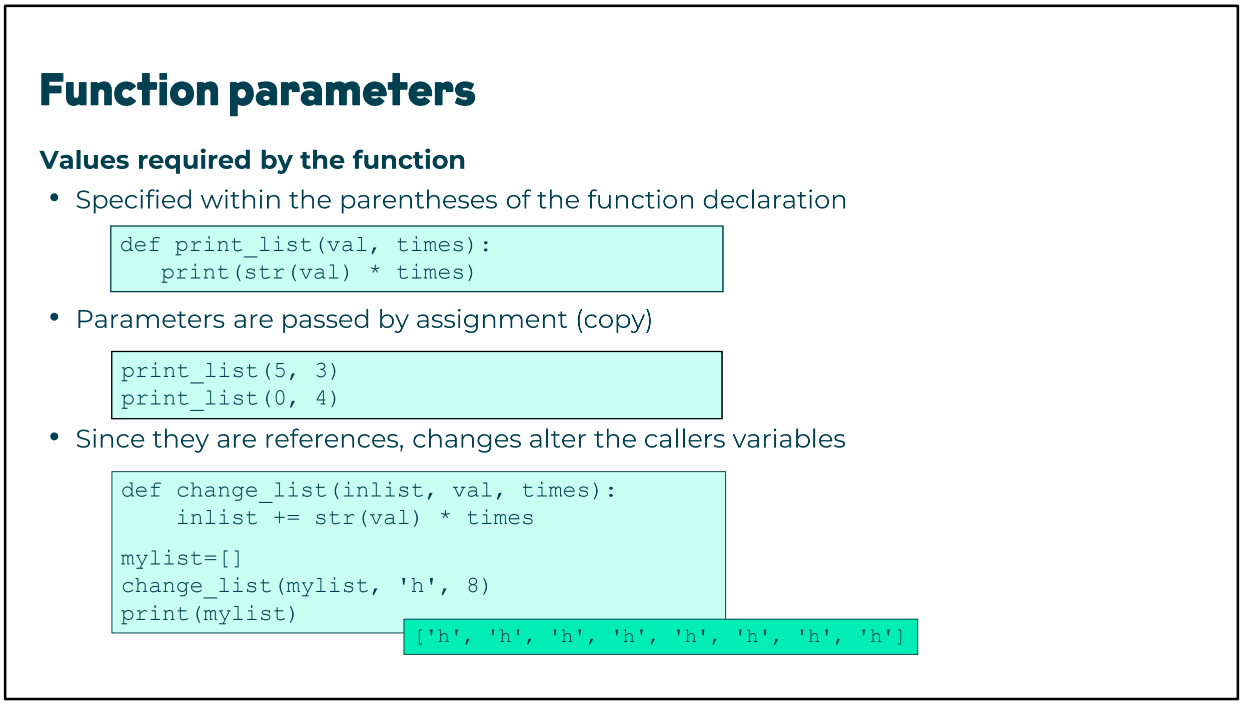
- **Functions are objects**
- **Defined with the def statement, followed by the argument list**
  - Just like conditionals, membership is by *indentation*

```
def make_list(val, times):
    res = str(val) * times
    return res
```

- **Arguments are named**
  - Defaults may be assigned
- **`return` statement is optional**
  - Any object type may be returned
  - Default is the empty object `None`
- **Variables are local if assigned**
  - Unless the keyword `global` is used

"Function names should be lowercase…" **– PEP008**

Python functions follow the same general syntax as conditional statements. The function is named, with an argument list terminated by a colon. Statements are part of the function by indentation. The argument list defines the variables passed with default values set. This a little like awk, PHP, and some mainframe scripting languages, and with similarities to aspects of C++. The arguments supplied by the caller must match in number those required, otherwise the program will not compile. Arguments may be of any type, however.



Function parameters

Values required by the function

- Specified within the parentheses of the function declaration

```
def print_list(val, times):
   print(str(val) * times)
```

- Parameters are passed by assignment (copy)

```
print_list(5, 3)
print_list(0, 4)
```

- Since they are references, changes alter the callers variables

```
def change_list(inlist, val, times):
    inlist += str(val) * times

mylist=[]
change_list(mylist, 'h', 8)
print(mylist)
```

```
['h', 'h', 'h', 'h', 'h', 'h', 'h', 'h']
```

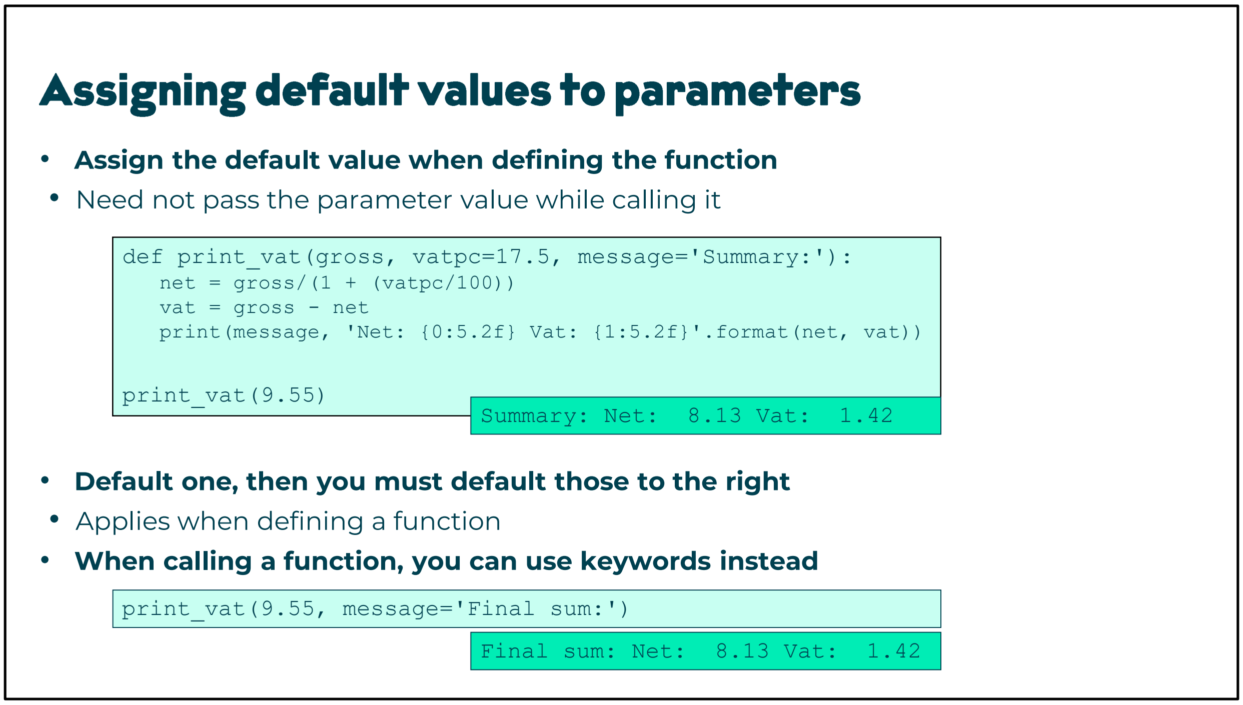
Passing parameters by assignment means that the function gets a local reference to the object passed. If that local reference is changed by the function it will have no effect outside the function. However, passing variables means we are assigning a reference, so changing a parameter within a function will alter the caller's object when a variable (reference) is passed. To prevent this for lists and dictionaries, pass a slice of the entire structure:

    print_thing(mylist[:])

Yes, it's a hack! Unfortunately, we get no indication that the function attempted to change the list and failed. Alternatively, we could use a tuple, which would produce a runtime error:

    print_thing(tuple(mylist))

Generally, we prefer to return values from functions rather than alter parameters, since it is not always clear if a function is going to clobber your variable.



Assigning default values to parameters

- **Assign the default value when defining the function**
  - Need not pass the parameter value while calling it

```
def print_vat(gross, vatpc=17.5, message='Summary:'):
   net = gross/(1 + (vatpc/100))
   vat = gross - net
   print(message, 'Net: {0:5.2f} Vat: {1:5.2f}'.format(net, vat))


print_vat(9.55)
```

```
Summary: Net:  8.13 Vat:  1.42
```

- **Default one, then you must default those to the right**
  - Applies when defining a function
- **When calling a function, you can use keywords instead**

```
print_vat(9.55, message='Final sum:')
```

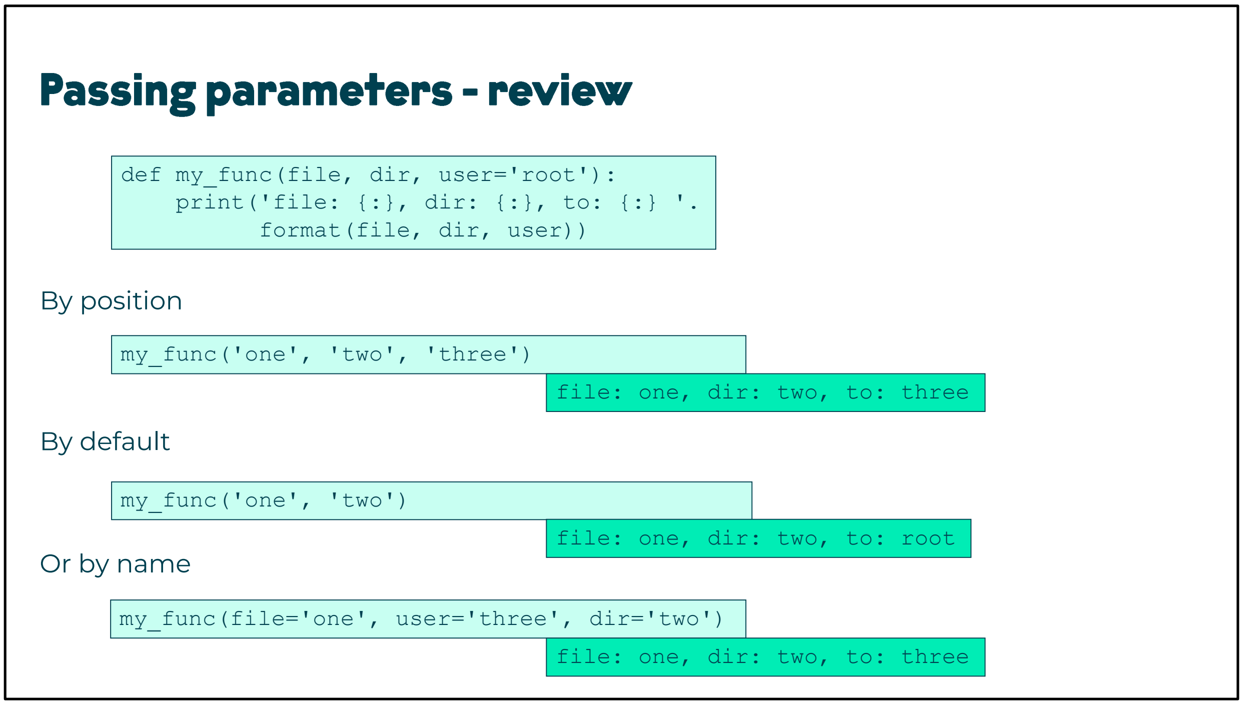
```
Final sum: Net:  8.13 Vat:  1.42
```

In Python, a function can have parameters with default values.
For example, suppose a function calculates the VAT on an item.
Most items in the UK were subject to a rate of 17.5%, which seemed reasonable as a default. However, a different rate might sometimes be used, so we do not want to hard-code it.
Parameters may only be assigned defaults on the right, if one is defaulted then all that follow must also be defaulted. So:

```
        def print_vat2 (vatpc=17.5, gross):
        ...
```

Gives:

```
File "functions.py", line 31
   def print_vat2 (vatpc=17.5, gross):
SyntaxError: non-default argument follows default argument
```



Passing parameters - review

```
def my_func(file, dir, user='root'):
    print('file: {:}, dir: {:}, to: {:} '.
          format(file, dir, user))
```

By position

```
my_func('one', 'two', 'three')
```

```
file: one, dir: two, to: three
```

By default

```
my_func('one', 'two')
```

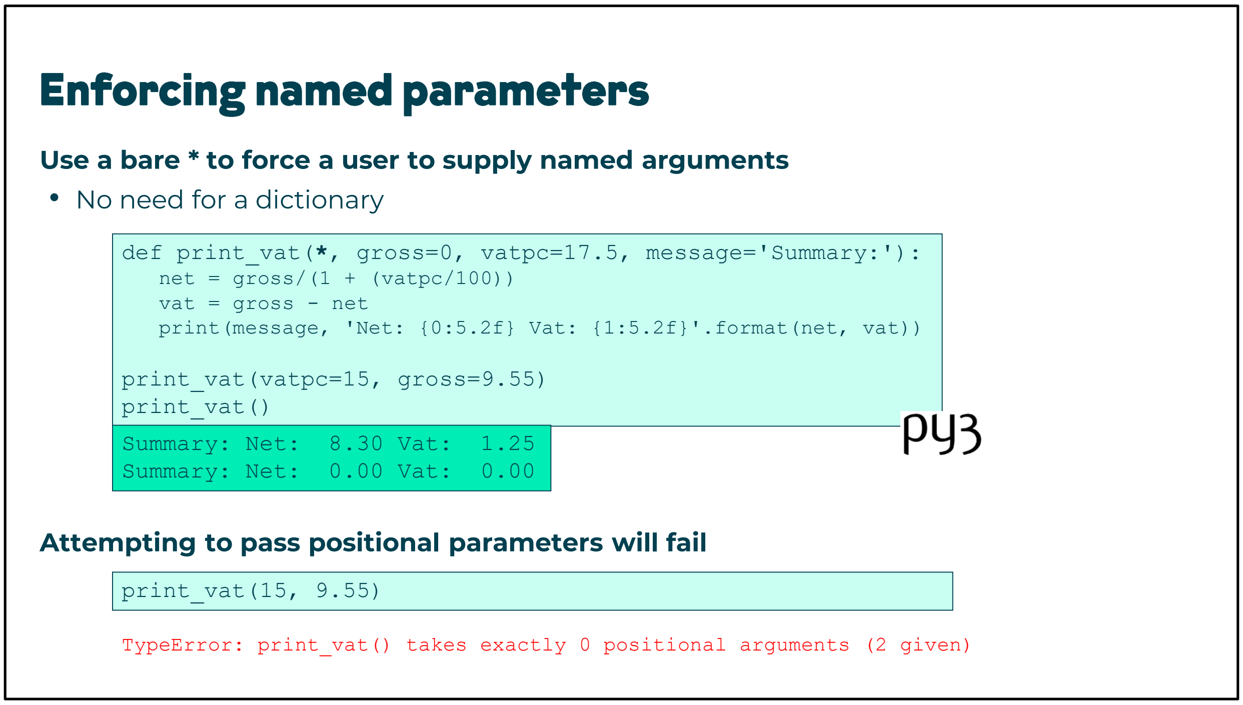
```
file: one, dir: two, to: root
```

Or by name

```
my_func(file='one', user='three', dir='two')
```

```
file: one, dir: two, to: three
```

Before we look further at parameter passing, we should review the mechanisms we have seen so far.



Enforcing named parameters

Use a bare * to force a user to supply named arguments

- No need for a dictionary

```
def print_vat(*, gross=0, vatpc=17.5, message='Summary:'):
    net = gross/(1 + (vatpc/100))
    vat = gross - net
    print(message, 'Net: {0:5.2f} Vat: {1:5.2f}'.format(net, vat))

print_vat(vatpc=15, gross=9.55)
print_vat()
```

```
Summary: Net:  8.30 Vat:  1.25
Summary: Net:  0.00 Vat:  0.00
```

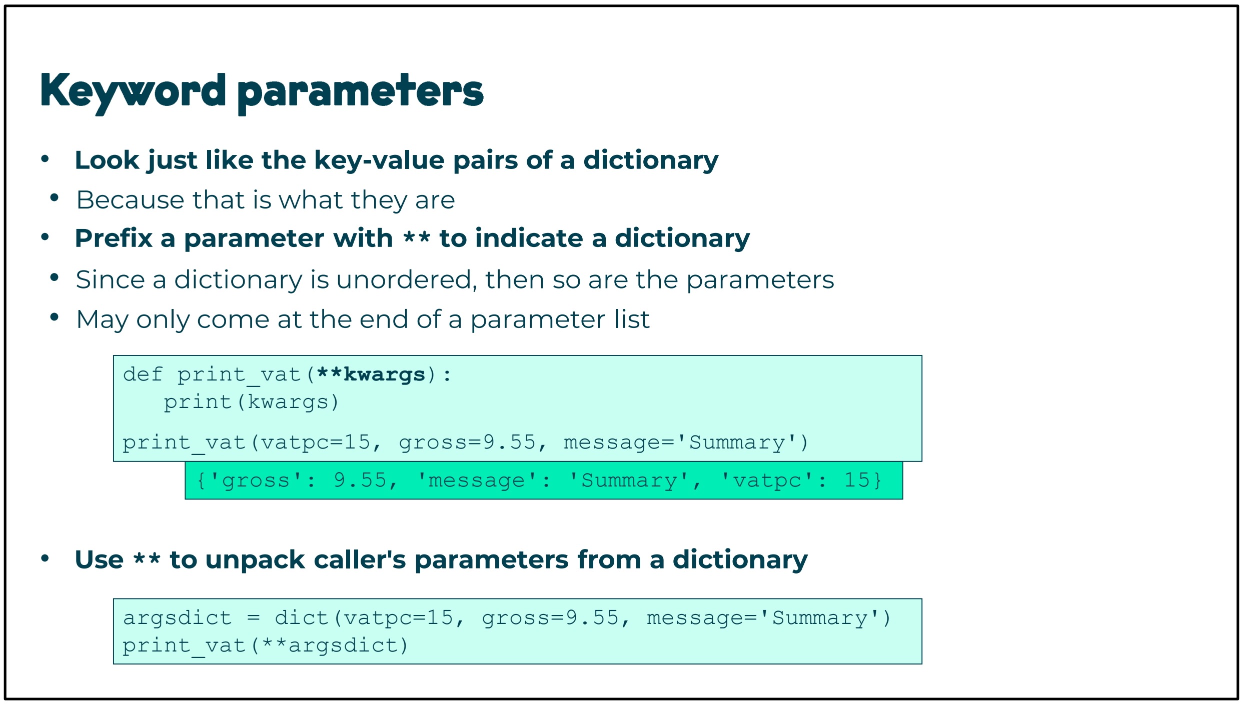
py3

Attempting to pass positional parameters will fail

```
print_vat(15, 9.55)
```

```
TypeError: print_vat() takes exactly 0 positional arguments (2 given)
```

The example shows a technique to force the user to specify parameters by name rather than position. However, calling such a function with *no* parameters is perfectly legal, since (in this case) they all have defaults.
All parameters defined after (to the right) of the ***** must be named parameters, but those to the left may be positional. This syntax is Python 3 specific.



Keyword parameters

- **Look just like the key-value pairs of a dictionary**
  - Because that is what they are
- **Prefix a parameter with `**` to indicate a dictionary**
  - Since a dictionary is unordered, then so are the parameters
  - May only come at the end of a parameter list

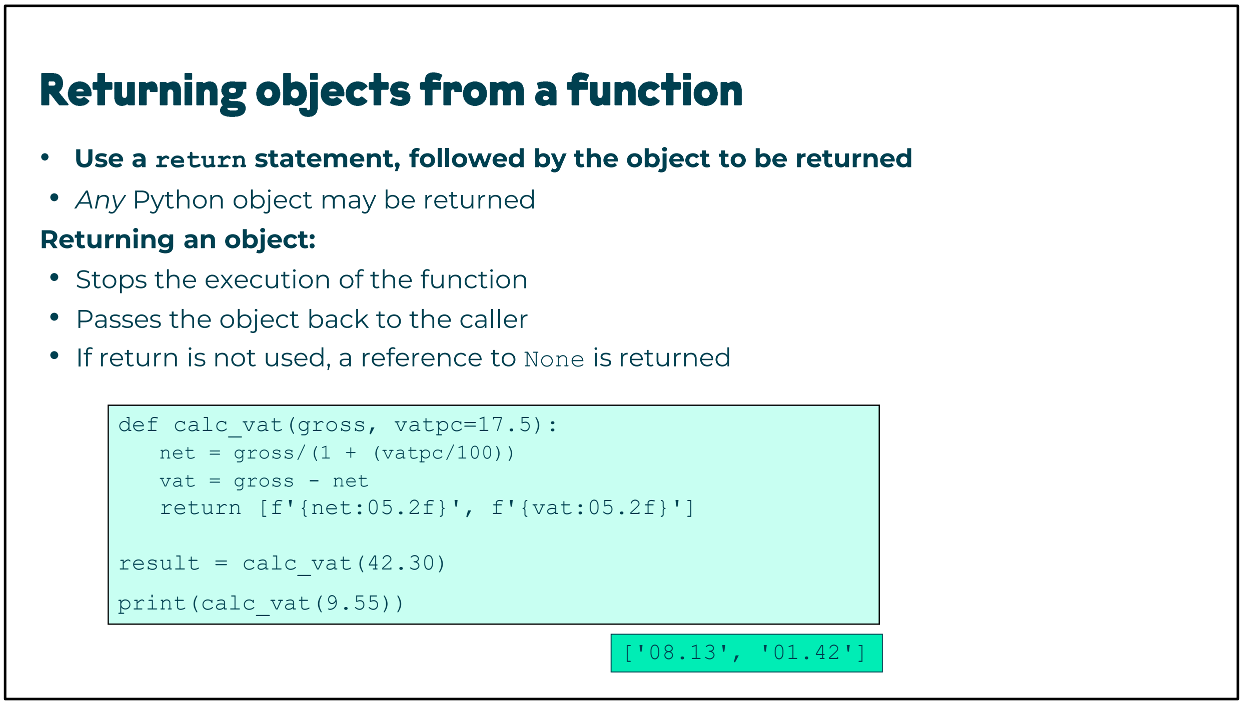
```
def print_vat(**kwargs):
   print(kwargs)

print_vat(vatpc=15, gross=9.55, message='Summary')
```

```
{'gross': 9.55, 'message': 'Summary', 'vatpc': 15}
```

- **Use `**` to unpack caller's parameters from a dictionary**

```
argsdict = dict(vatpc=15, gross=9.55, message='Summary')
print_vat(**argsdict)
```

Keyword parameters (by convention called **`kwargs`**) enable the parameters to be passed as a dictionary.
Unpacking named parameters requires two asterisks as a prefix. The parameters are placed into a dictionary which, of course, is unordered. This means that the user does not need to get the order correct. This is particularly useful with a long parameter list.
A parameter prefixed **`**`** may only be used at the end of a parameter list, any other position will give a syntax error.



Returning objects from a function

- **Use a `return` statement, followed by the object to be returned**
- *Any* Python object may be returned

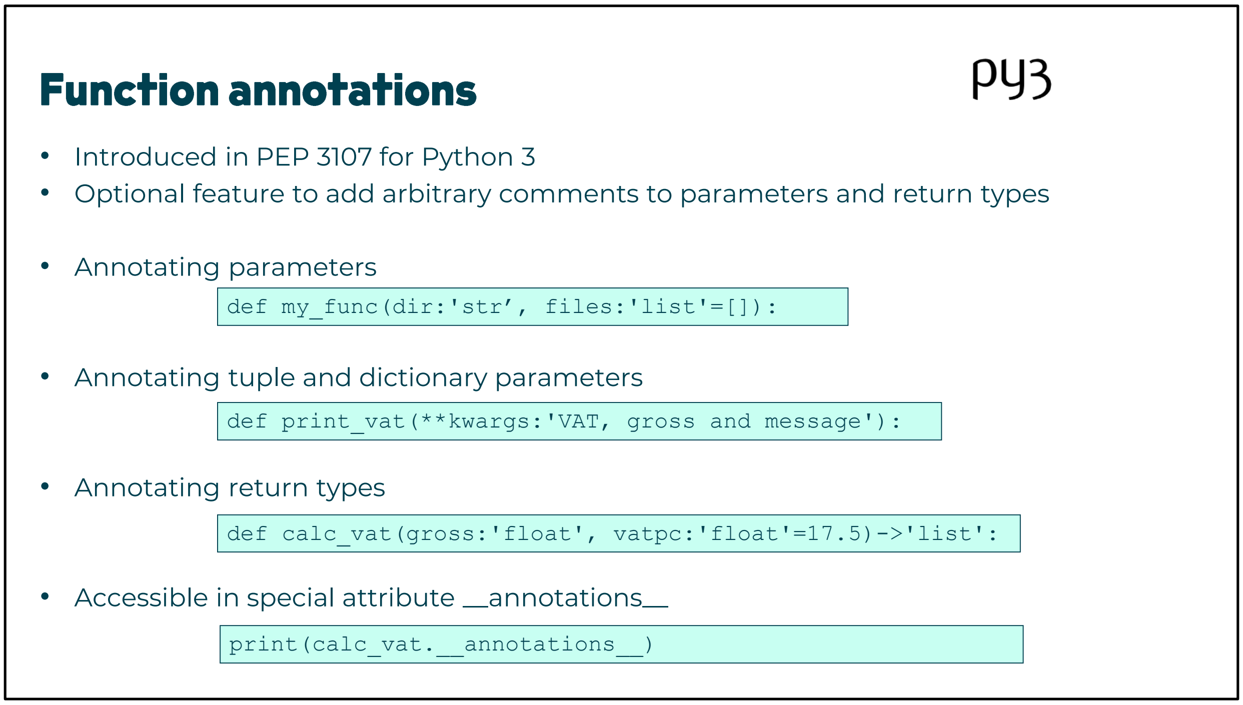
Returning an object:

- Stops the execution of the function
- Passes the object back to the caller
- If return is not used, a reference to `None` is returned

```
def calc_vat(gross, vatpc=17.5):
    net = gross/(1 + (vatpc/100))
    vat = gross - net
    return [f'{net:05.2f}', f'{vat:05.2f}']

result = calc_vat(42.30)
print(calc_vat(9.55))
```

```
['08.13', '01.42']
```

A function may return a reference to any type of object, including a list, a tuple, or a dictionary.
If a function does not return a value, and the calling code tries to use one, then None is passed. None is a "catch-all" object which means the lack of a value. It does not explicitly mean zero, but can have the same effect under some circumstances. For example, None evaluates to False in a Boolean context.



Function annotations

- Introduced in PEP 3107 for Python 3
- Optional feature to add arbitrary comments to parameters and return types

- Annotating parameters

```
def my_func(dir:'str’, files:'list'=[]):
```

- Annotating tuple and dictionary parameters

```
def print_vat(**kwargs:'VAT, gross and message'):
```

- Annotating return types

```
def calc_vat(gross:'float', vatpc:'float'=17.5)->'list':
```

- Accessible in special attribute __annotations__

```
print(calc_vat.__annotations__)
```

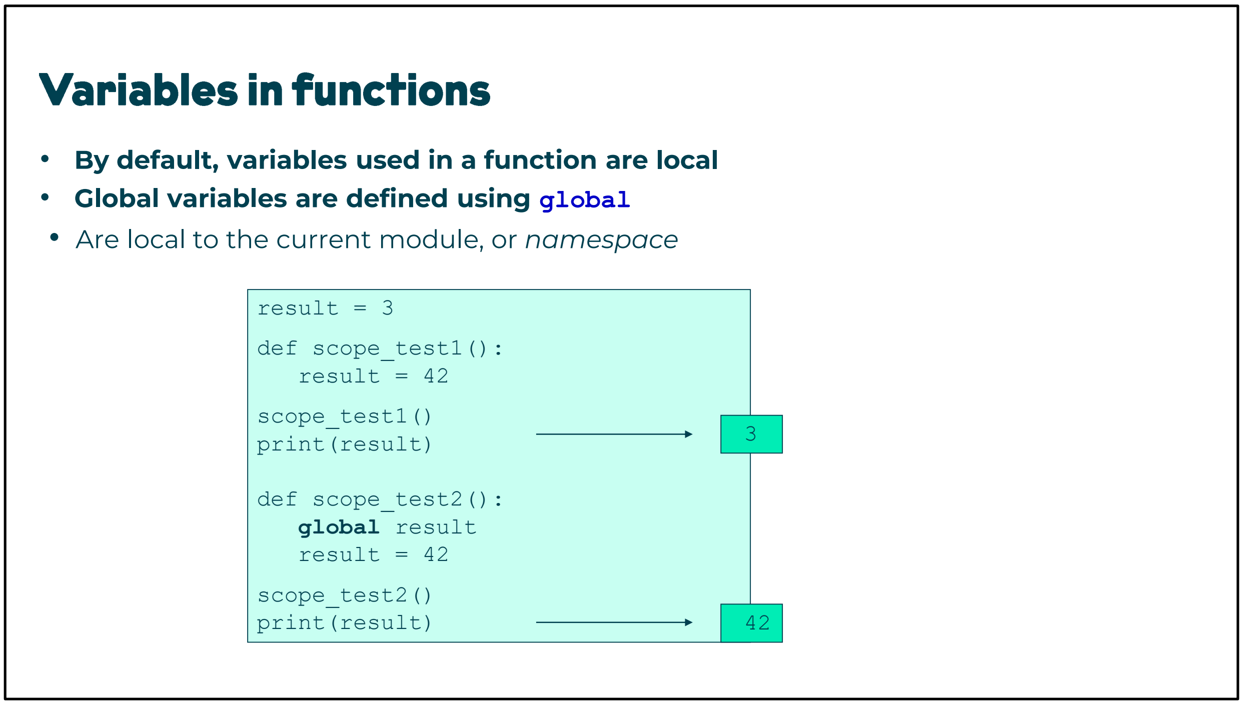
Function annotations were introduced in Python 3 to add arbitrary metadata to arguments and return values. They are completely optional but can be useful for explaining the use of a parameter and return type by using an expression, e.g. ‘str’ or ‘VAR, gross and message’. These expressions are evaluated at compile time and have no life or use in the runtime environment of the program. So, although similar in looks to type checking in other languages, they are strictly just explanations and cannot enforce type checking; remember Python is a dynamic language.

Python 2 did not support annotations but several tools and libraries filled this gap using decorators (see PEP 318) or parsing the functions docstring for annotations.

Even in Python 3, annotations are effectively ignored and only really useful with 3rd party libraries. For instance, one library might use annotations to perform type checking or another might use them for improved help messages.

The annotations can be accessed using the special attribute:

function_name.__annotations__.



Variables in functions

- **By default, variables used in a function are local**
- **Global variables are defined using `global`**
  - Are local to the current module, or *namespace*

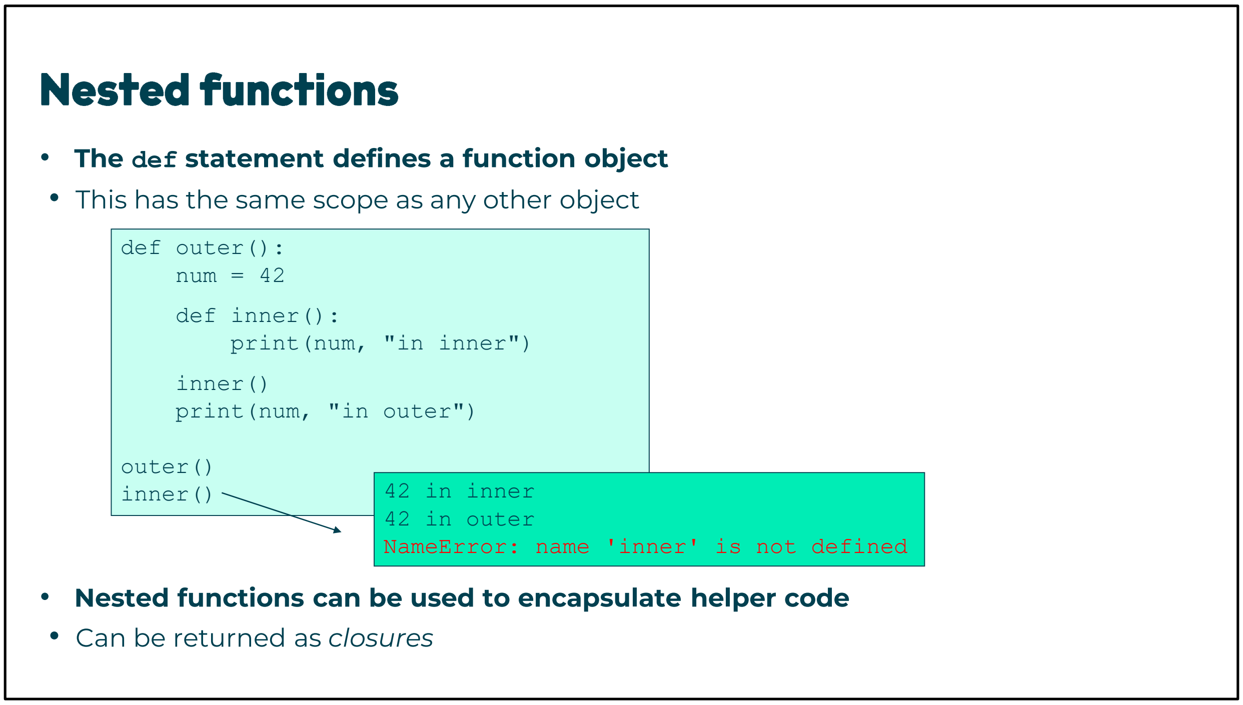
```
result = 3

def scope_test1():
    result = 42

scope_test1()
print(result)
```

```
3
```

```
def scope_test2():
    global result
    result = 42

scope_test2()
print(result)
```

```
42
```

The rules of scope in Python are that an undefined variable used in a function must be a global, but if a value is assigned in the function before it is used then it is a local (unless it is prefixed with global).
Variables first assigned in a function are local to the function body, they disappear on exit from the function. In the example, we have two variables called result (bad coding, but it can happen). The variable inside the function bears no relationship with the one outside it. Assigning a different object to the local version does not affect the "outer" version in any way.
Occasionally, we might want to alter a global variable, in which case we must declare it as being global. The global keyword should be used before the variable is used. You can actually get away with declaring it as global later in the function, but you will get syntax warnings.
Global variables are generally frowned upon, and are particularly problematic when an application is multi-threaded. Try to avoid them.



Nested functions

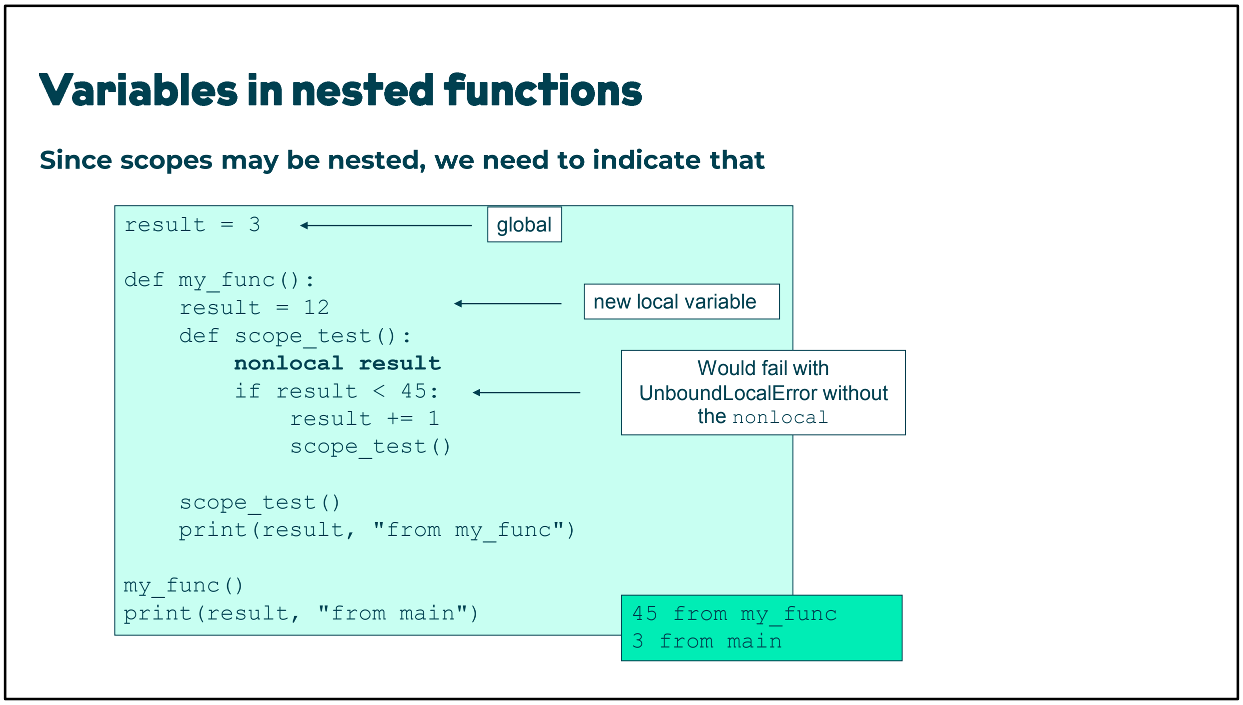
- **The `def` statement defines a function object**
  - This has the same scope as any other object

```
def outer():
    num = 42

    def inner():
        print(num, "in inner")

    inner()
    print(num, "in outer")

outer()
inner()
```

```
42 in inner
42 in outer
NameError: name 'inner' is not defined
```

- **Nested functions can be used to encapsulate helper code**
  - Can be returned as *closures*

Placing a function inside another is not something which all languages support. It can be useful for simple functions, for so-called "closures", and for function factories. Notice that, in the example, the function has not declared num as global, yet the inner() function has access to it because it is still in scope. If an assignment was made to num in inner(), then a new variable would be created which would only be visible to inner().



Variables in nested functions

Since scopes may be nested, we need to indicate that

```
result = 3

def my_func():
    result = 12
    def scope_test():
        nonlocal result
        if result < 45:
            result += 1
            scope_test()

    scope_test()
    print(result, "from my_func")

my_func()
print(result, "from main")
```

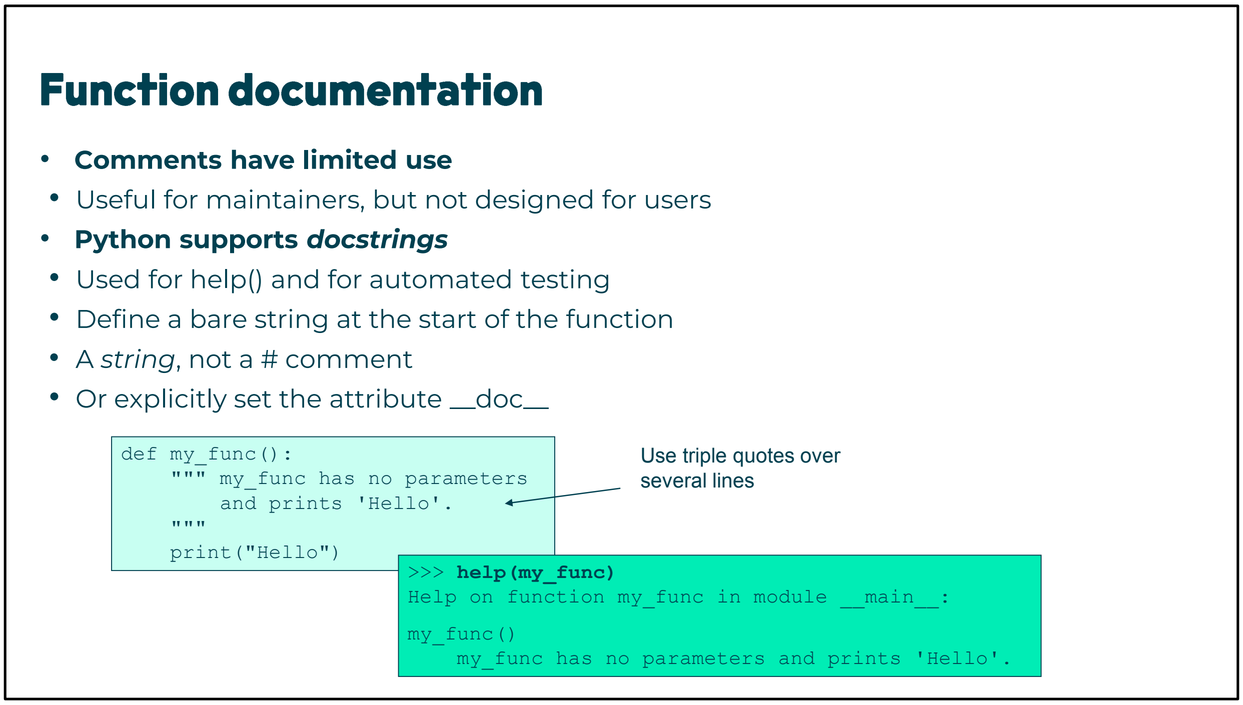
```
45 from my_func
3 from main
```

Python's clean syntax and dynamic typing means that there is no mechanism for being able to define a new variable. This means that, on occasion, there can be confusion concerning exactly when a new variable should be created, and when an existing one should be used. We have already seen **`global`** as a statement which identifies a variable as being in global scope.
What about other enclosing scopes? Python is a little unusual in being able to define local, or nested, functions. If we wish to define a variable as being within an enclosing scope, then **`nonlocal`** is used.
In the example on the slide, if **`nonlocal`** was omitted then the `if` statement would be testing a variable before it was declared, since we have not defined result as global (in fact we don't want to, because that is not one we wish to use).
Defining as **`nonlocal`** means the variable `result` declared in the outer function will be used, and the recursion works successfully. Of course in this case, we could have passed the variable as a parameter instead.
Note that if the inner function (`scope_test`) is returned then it may be called from the outside, still with the nonlocal variable in

scope:

```
    def my_func():
        ...
        return scope_test
    rfunc = my_func()
    rfunc()
```

This is known as a *closure*. In Python 2, it could only be achieved using globals.



Function documentation

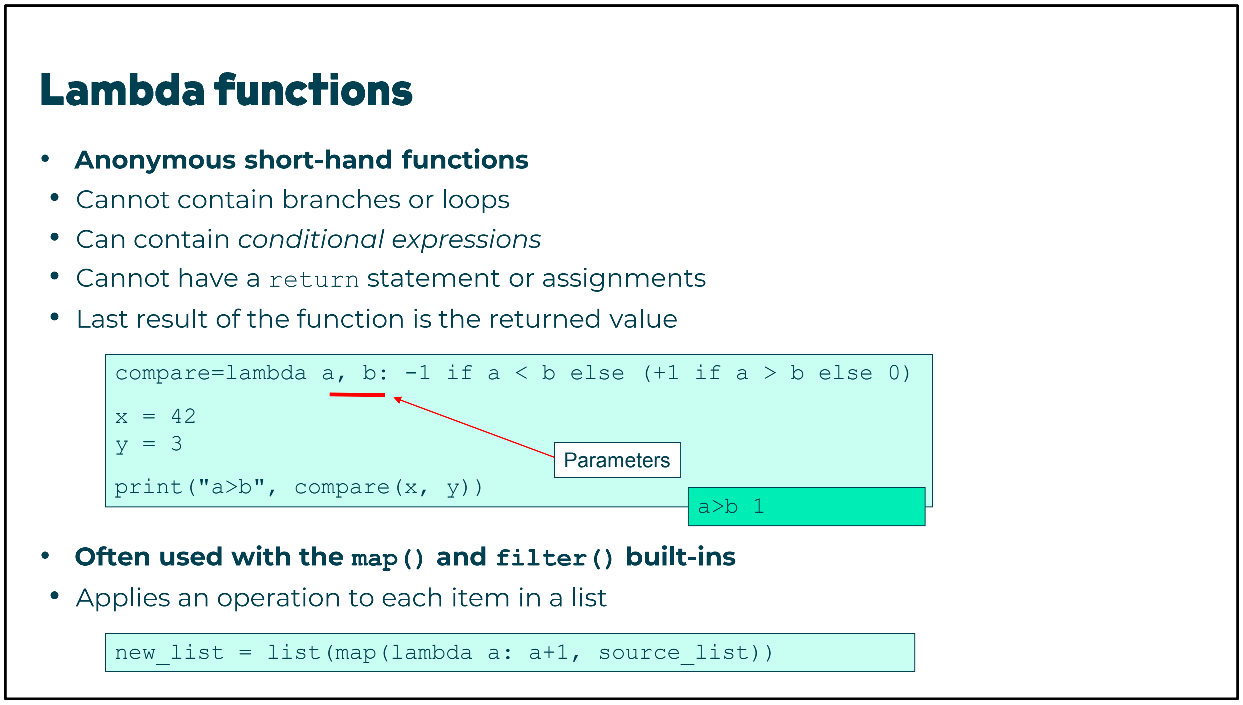
- **Comments have limited use**
  - Useful for maintainers, but not designed for users
- **Python supports *docstrings***
  - Used for help() and for automated testing
  - Define a bare string at the start of the function
  - A *string*, not a # comment
  - Or explicitly set the attribute __doc__

```
def my_func():
    """ my_func has no parameters
        and prints 'Hello'.
    """
    print("Hello")
```

Use triple quotes over several lines

```
>>> help(my_func)
Help on function my_func in module __main__:

my_func()
    my_func has no parameters and prints 'Hello'.
```

One reason why the online help in Python is so comprehensive is because it is built-in to the language - not added as an afterthought like some we could mention.
If we place a string at the start of a function (or a module), it is taken as documentation. Most people use a three-quoted multi-line string, but it can be any form of string.
By "start of the function" we mean that only white-space or comments are allowed between the function `def` statement and the docstring. Alternatively, we can assign the string to a special variable called **`__doc__`**.



Lambda functions

- **Anonymous short-hand functions**
  - Cannot contain branches or loops
  - Can contain *conditional expressions*
  - Cannot have a `return` statement or assignments
  - Last result of the function is the returned value

```
compare=lambda a, b: -1 if a < b else (+1 if a > b else 0)

x = 42
y = 3

print("a>b", compare(x, y))
```

Parameters

```
a>b 1
```

- **Often used with the `map()` and `filter()` built-ins**
  - Applies an operation to each item in a list

```
new_list = list(map(lambda a: a+1, source_list))
```

The term **`lambda`** comes from Lisp, and is a little off-putting. Being Greek and sounding mathematical, it has the aura of complexity, but Python **`lambda`** functions are really quite simple, for **`lambda`** read **inline**.
`Lambda` functions cannot really do anything that "normal" **`def`**'ed functions cannot do, they are just shorter and quicker to write. They are great for relatively small in-line functions that do not need a name - **`lambda`** returns a reference to the function, which may be stored in a variable. That variable is then used *as if* it was the name of the function.
Many Python built-ins take a "callable-object" as a parameter. This can be the name of a function, or the name of a reference to a function, or a **`lambda`** statement used without any names being involved. The example with **`map()`** adds one to each item in the list.

SUMMARY

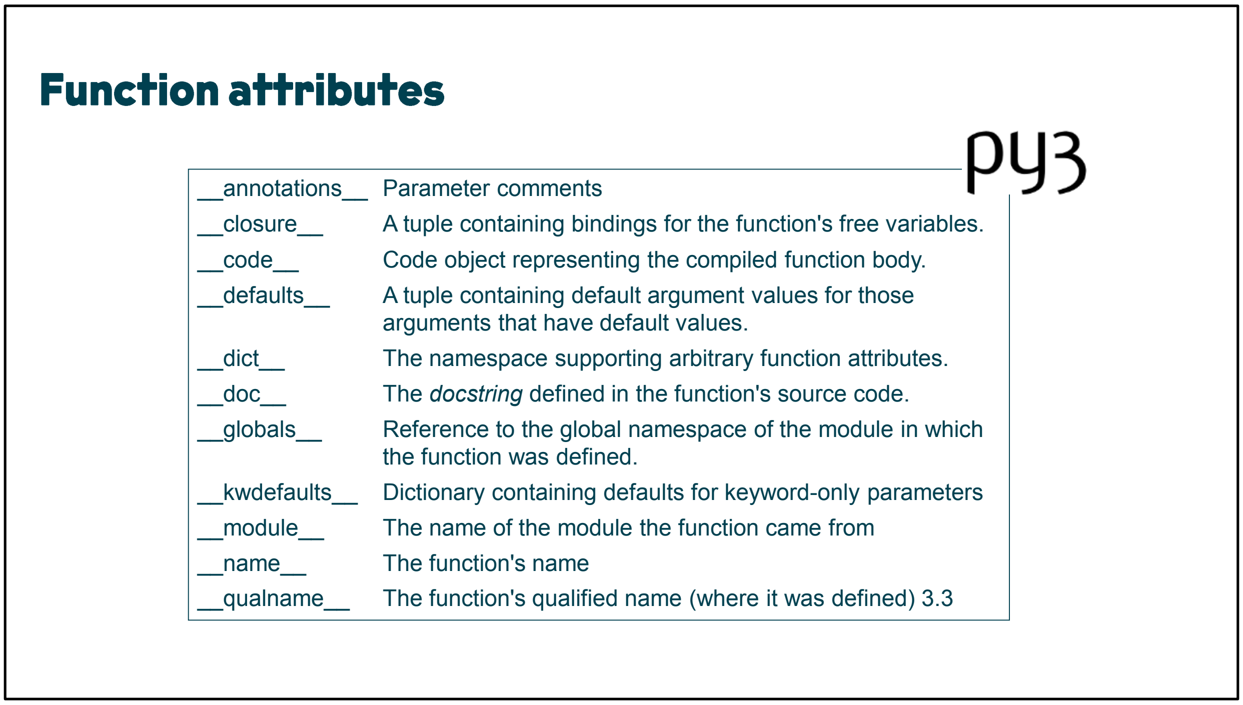
- **A function is a defined object**
  - Variables have local scope unless `global` is used
  - Other functions can be nested within
- **Parameters are declared local variables**
  - May be assigned defaults, from the right
  - `*`*arg* means unpack to a tuple
  - `**`*arg* means unpack to a dictionary
  - `*` forces the caller to use named parameters
- **Can return any object**
  - Including lists and dictionaries
- **Can include a *docstring* - a bare string at the start**
- **Short, inline, anonymous functions can be defined using lambda**

More on keyword parameters

```
import sys

def my_func(**user_args):
    args = {'country':'England', 'town':'London',
            'currency':'Pound', 'language':'English'}

    diff = set(user_args.keys()) - set(args.keys())
    if diff:
        print('Invalid args:', tuple(diff), file=sys.stderr)
        return

    args.update(user_args)
    print(args)


my_dict = dict(town = 'Glasgow', country = 'Scotland’)
my_func(**my_dict)

my_func(twn = 'Glasgow', county = 'Scotland')
```

```
{'town': 'Glasgow', 'currency': 'Pound',
                    'language': 'English', 'country': 'Scotland'}
Invalid args: ('county', 'twn')
```

The first line of the output has been wrapped around to fit.



Function attributes

py3

__annotations__	Parameter comments
__closure__	A tuple containing bindings for the function's free variables.
__code__	Code object representing the compiled function body.
__defaults__	A tuple containing default argument values for those arguments that have default values.
__dict__	The namespace supporting arbitrary function attributes.
__doc__	The *docstring* defined in the function's source code.
__globals__	Reference to the global namespace of the module in which the function was defined.
__kwdefaults__	Dictionary containing defaults for keyword-only parameters
__module__	The name of the module the function came from
__name__	The function's name
__qualname__	The function's qualified name (where it was defined) 3.3

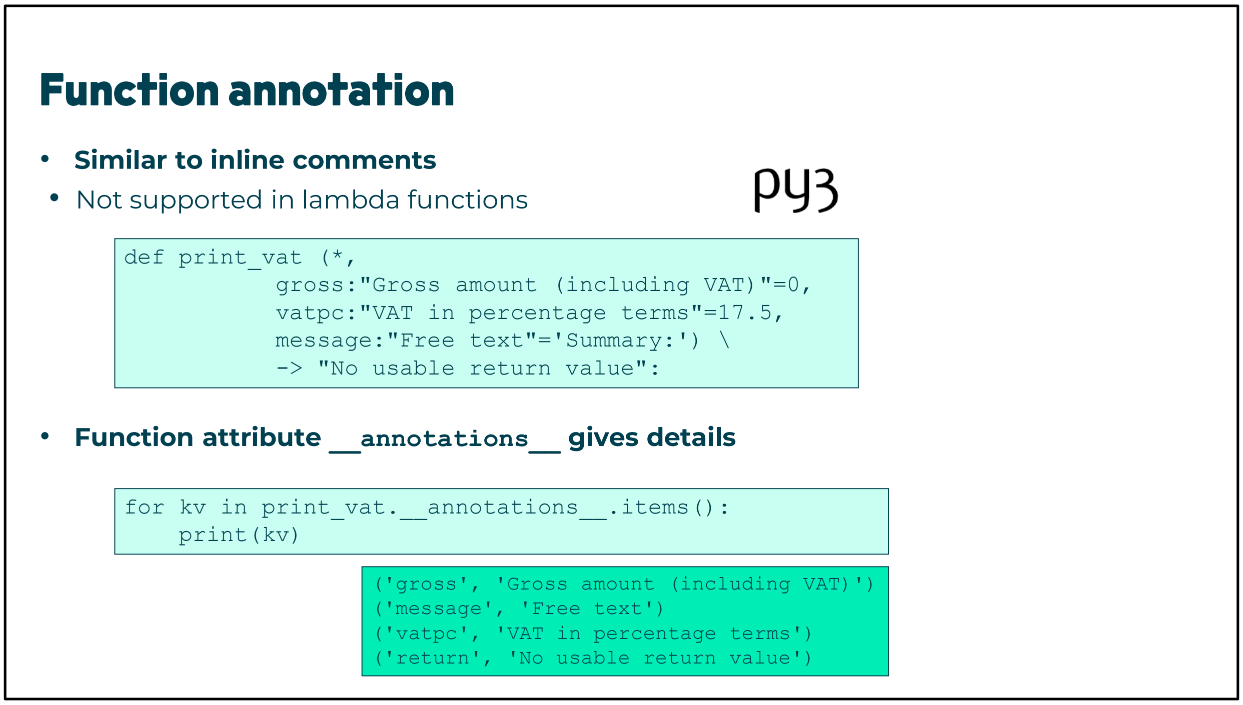
In Python 2, most of these were prefixed `func_`, but `__annotations__`, `__kwdefaults__`, and `__qualname__` are Python 3 specific.
Use them like this:

```
def myfunc():
    print (myfunc.__name__)
```

You might think there is little point in getting the name if you already need it, but it is used more with references to functions. To give a simple example:

```
fref = myfunc
...
print (fref.__name__)
```

The attribute `__qualname__` (introduced at 3.3) will give the original name and where it was defined.



Function annotation

- **Similar to inline comments**
- Not supported in lambda functions

py3

```
def print_vat (*,
            gross:"Gross amount (including VAT)"=0,
            vatpc:"VAT in percentage terms"=17.5,
            message:"Free text"='Summary:') \
            -> "No usable return value":
```

- **Function attribute `__annotations__` gives details**

```
for kv in print_vat.__annotations__.items():
    print(kv)
```

```
('gross', 'Gross amount (including VAT)')
('message', 'Free text')
('vatpc', 'VAT in percentage terms')
('return', 'No usable return value')
```

Annotations allow meta-data to be attached to parameters and return values. Python itself ignores them, as it would comments, but they are available for use by third-party libraries. Aside from the basic syntax, there are no standard data types or formats - these are left flexible for third-party libraries to exploit.