Course 5

POO

revision PROCEDURAL PROGRAMMING

Chap. 3 Functions
Contents

- Function calls
- Type conversion functions
- Maths functions
- Importing with from
- Composition — **Course 4 (previous)**

- Adding new functions — **Course 5 (today)**
- Flow of execution
- Parameters and arguments
- Variables and parameters are local
- Stack diagram
- Fruitful functions and void functions
- WHY FUNCTIONS?
- Debugging, Glossary, Exercises

Adding new user-functions

- So far, we have only been using the functions that come with Python, but it is also possible to add new functions.
- A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.

```
>>> def print_lyrics():
        print “Gaudeamus igitur"
        print “Juvenes dum sumus“

>>> print_lyrics()
Gaudeamus igitur
Juvenes dum sumus
```

header

body

!!!
layout rule

call

Adding new user-functions

- Defining a function creates an object with the same name → a function object which has type function.
- Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called repeat_lyrics:

```
>>> def repeat_lyrics():
        print_lyrics()
        print_lyrics()
```

- You have to create a function before you can execute it. It means that the function definition has to be done before the first time the function is called.

Flow of execution

- In order to ensure that a function is defined before its first use, you have to know **the order in which statements are executed**, which is called the flow of execution.
- Execution always begins at the first statement of the program. Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- A function call is like **a detour in the flow of execution**. Instead of going to the next statement, **the flow jumps** to the body of the function, executes all the statements there, and then comes back to pick up where it left off.
- That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!
- Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.
- What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution instead of reading the program statement by statement.

Parameters and Arguments

- Some of the built-in functions we have seen require arguments. For example, when you call math.sin you pass a number as an argument. Some functions take more than one argument: math.pow takes two, the base and the exponent.

- Here is an example of a user-defined function that takes an argument:

parameter

```
>>> def print_twice(bruce):
        print bruce
        print bruce
```

- **Inside the function, the arguments are assigned to variables which correspond to parameters**.

- The print_twice function assigns different arguments to the bruce parameter. When this function is called, it prints the value of the argument (whatever it is) twice.

Parameters and arguments

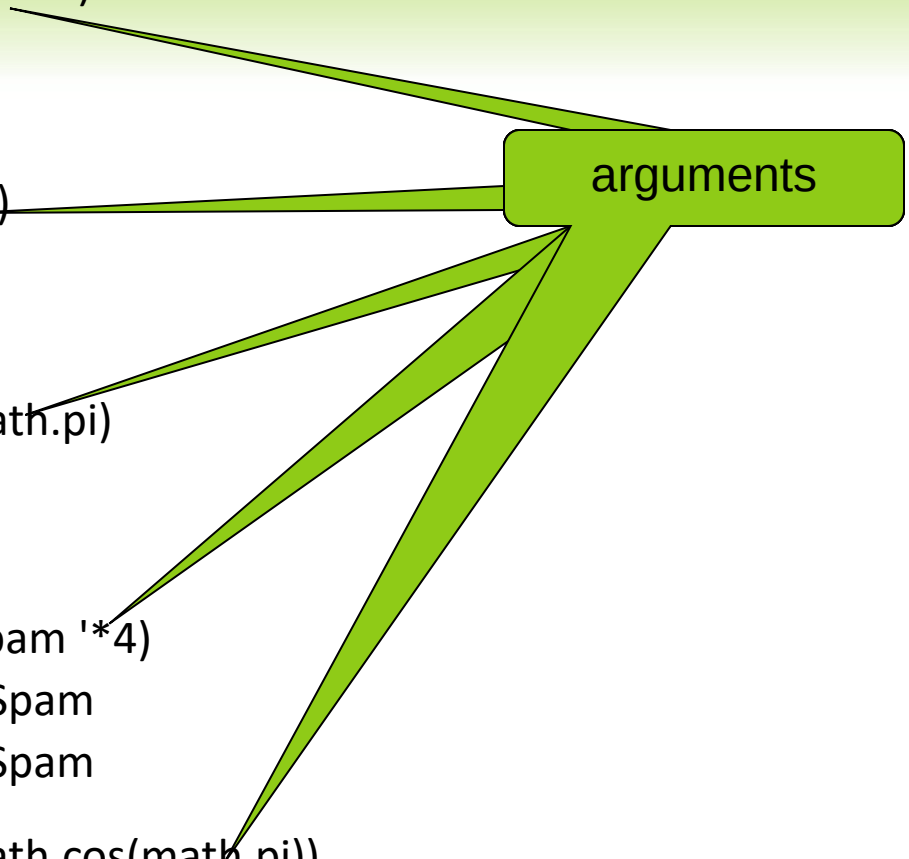
```
>>> print_twice('Spam')
Spam
Spam

>>> print_twice(17)
17
17

>>> print_twice(math.pi)
3.14159265359
3.14159265359

>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam

>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```



Parameters and arguments

- You can also use a variable as an argument:

```
>>> michael = ‘My best friend!'
>>> print_twice(michael)
My best friend!
My best friend!
```

- The name of the variable michael we pass as an argument has nothing to do with **the name** of the parameter bruce.

Variables and parameters are local

- When you create a variable inside a function, that variable is **local**, which means that it only exists inside the function.

```
>>> def cat_twice(part1, part2):
        cat = part1 + part2
        print_twice(cat)
```

- This function takes two arguments, concatenates them, and prints the result twice.

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

- When cat_twice terminates, the variable cat is destroyed. If we try to print it, we get an exception:

```
>>> print cat
NameError: name 'cat' is not defined
```

Variables and parameters are local

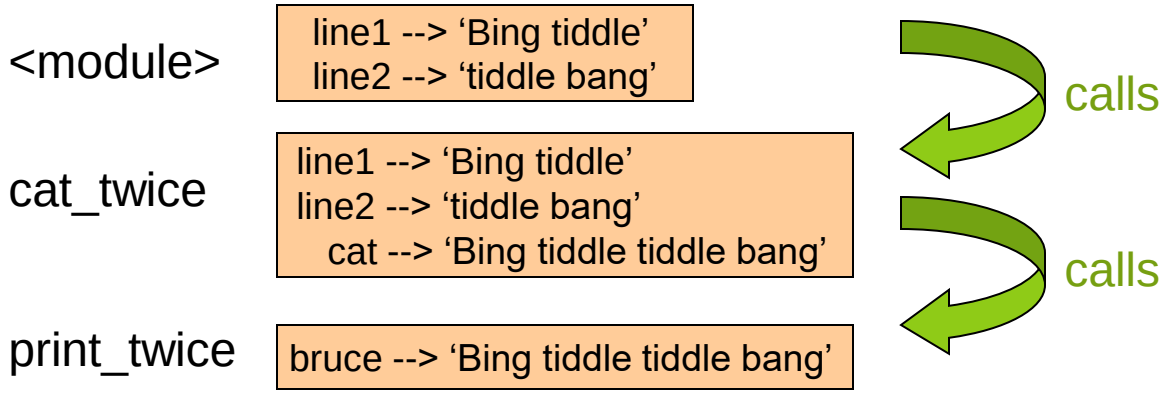
- Parameters are also local. Outside print-twice, there is no such thing as bruce.

```
>>> def print_twice(bruce):
        print bruce
        print bruce
```

Stack diagrams

- To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.

- Each function is represented by a **frame**. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is



- The frames are arranged in a stack that indicates which function called which, and so on. In this example, print_twice was called by cat_twice, and cat_twice was called by __main__, which is **a special name for the topmost frame**. When you create a variable **outside of any function**, it belongs to __main__ and it becomes a global variable.

Fruitful functions and void functions

- Some of the functions we are using, such as the math functions, yield results; we call them **fruitful functions**.
- Other functions, like print_twice, perform an action but don't return a value. They are called **void functions**.

```
>>> x = math.cos(radians)
>>> golden = (math.sqrt(5) + 1) / 2
```

- When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.2360679774997898
```

- But **in a script**, if you call a fruitful function all by itself, the return value is lost forever! This script computes the square root of 5, but since it doesn't store or display the result, it is not very useful.

```
math.sqrt(5)
```

Fruitful functions and void functions

- **Void functions** might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called None.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result
None
```

- The value None is not the same as the string 'None'. It is a special value that has its own type:

```
>>> print type(None)
<type 'NoneType'>
```

WHY FUNCTIONS?

- Creating a new function gives you an opportunity **to name a group of statements**, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have **to make that change in one place**.
- Dividing a long program into functions allows you to **debug** the parts one at a time and then **assemble** them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can **reuse** it.

Debugging, Glossary, Exercises