Chapter_3_Functions

March 6, 2024

0.1 3.1 Function calls

[1]:
```
int('32')
```

[1]:
```
32
```

[2]:
```
float('3.14159')
```

[2]:
```
3.14159
```

[3]:
```
int(-2.3)
```

[3]:
```
-2
```

[4]:
```
int(-2.7)
```

[4]:
```
-2
```

[5]:
```
int('Hello')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-6765ce49acfe> in <module>
----> 1 int('Hello')

ValueError: invalid literal for int() with base 10: 'Hello'
```

[7]:
```
int(3.99999)
```

[7]:
```
3
```

[8]:
```
float(32)
```

[8]:
```
32.0
```

[9]:
```
str(32)
```

[9]:
```
'32'
```

[10]:
```
str(3.14159)
```

[10]:
```
'3.14159'
```

0.2 3.2 Math functions

Python has a math module that provides most of the familiar mathematical functions. A module is a file that contains a collection of related functions.

Before we can use the functions in a module, we have to import it with an import statement:

[11]:
```
import math
```

This statement creates a module object named math. If you display the module object, you get some information about it:

[12]:
```
math
```

[12]:
```
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called dot notation.

[13]:
```
log_10 = math.log10(100)
log_10
```

[13]:
```
2.0
```

[14]:
```
radians = 0.7
height = math.sin(radians)
height
```

[14]:
```
0.644217687237691
```

[15]:
```
degrees = 45
radians = degrees / 180.0 * math.pi
math.sin(radians)
```

[15]:
```
0.7071067811865476
```

The expression math.pi gets the variable pi from the math module. Its value is a floating point approximation of pi, accurate to about 15 digits.

0.3 3.3 Composition

One of the most useful features of programming languages is their ability to take small building blocks and compose them

[17]:
```
x=4
math.exp(math.log(x+1))
```

[17]:
```
4.999999999999999
```

0.4 3.4 Adding new functions

A function definition specifies the name of a new function and the sequence of statements that run when the function is called

[18]:
```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

def is a keyword that indicates that this is a function definition. The name of the function is print_lyrics. The rules for function names are the same as for variable names: letters, numbers and underscore are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name

The empty parentheses after the name indicate that this function doesn't take any arguments. The first line of the function definition is called the header; the rest is called the body. The header has to end with a colon and the body has to be indented. By convention, indentation is always four spaces. The body can contain any number of statements

The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string. All quotation marks (single and double) must be "straight quotes", usually located next to Enter on the keyboard. "Curly quotes", like the ones in this sentence, are not legal in Python

If you type a function definition in interactive mode, the interpreter prints dots (...) to let you know that the definition isn't complete:

[21]:
```
def print_lyrics():
... print("I'm a lumberjack, and I'm okay.")
... print("I sleep all night and I work all day.")
```

```
  File "<ipython-input-21-7df705a4ef51>", line 2
    print("I'm a lumberjack, and I'm okay.")
        ^
IndentationError: expected an indented block
```

Defining a function creates a function object, which has type function:

[22]:
```
print(print_lyrics)
```

```
<function print_lyrics at 0x00000235D379CAE8>
```

[23]:
```
type(print_lyrics)
```

[23]:
```
function
```

[24]:
```
print_lyrics()
```

```
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

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:

[25]:
```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

[26]:
```
repeat_lyrics()
```

```
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

0.5 3.8 Variables and parameters are local

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

[39]:
```
def print_twice(x):
    print(x)
    print(x)
```

[40]:
```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

[41]:
```
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:

[42]:
```
print(cat)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-42-34599fba884e> in <module>
----> 1 print(cat)
```

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

Parameters are also local

When you create a variable outside of any function, it belongs to **main**.

This list of functions is called a traceback. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error. The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

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

[43]:
```
result = print_twice('Bing')
```

```
Bing
Bing
```

[44]:
```
print(result)
```

```
None
```

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

[45]:
```
type(None)
```

[45]: NoneType

[1]:
```
'None'
```

[1]: 'None'

[ ]:
```

```