

Introduction to Python

Topic 5: Functions

Aim: How can we write expressions as functions and call on them?

Functions

Function calls

In the context of programming, a *function* is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a *function call*:

```
>>> type(32)
<class 'int'>
```

The name of the function is `type`. The expression in parentheses is called the *argument* of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the `type` function, is the type of the argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the *return value*.

Turn and Talk

Identify the name of the function. State the type of the input shown and predict the outcome

```
>>> max('Hello world')
```

```
>>> min('Hello world')
```

Built-in functions

Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

The `max` and `min` functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

The `max` function tells us the "largest character" in the string (which turns out to be the letter "w") and the `min` function shows us the smallest character (which turns out to be a space).

Answers and explanations to #3

Another very common built-in function is the `len` function which tells us how many items are in its argument. If the argument to `len` is a string, it returns the number of characters in the string.

```
>>> len('Hello world')  
11  
>>>
```

These functions are not limited to looking at strings. They can operate on any set of values, as we will see in later chapters.

You should treat the names of built-in functions as reserved words (i.e., avoid using "max" as a variable name).

Practice (group work):

Predict the outcomes of the following. Then run them to confirm the answers.

`int('32')` `int('Hello')` `int(3.99999)` `int(-2.3)` `float(32)` `float('3.14159')` `str(32)` `str(3.14159)`

Type conversion functions

Answer to #6

Python also provides built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

`int` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finally, `str` converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Random numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be *deterministic*. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use *algorithms* that generate *pseudorandom* numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

You Try it!
Post a screenshot for the
outcome

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Exercise 1: Run the program on your system and see what numbers you get. Run the program more than once and see what numbers you get.

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

Answer to #9

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

The `random` function is only one of many functions that handle random numbers. The function `randint` takes the parameters `low` and `high`, and returns an integer between `low` and `high` (including both).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The `random` module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

Adding new 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. Once we define a function, we can reuse the function over and over throughout our program.

Here is an example:

```
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 some punctuation marks 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. Later we will build functions that take arguments as their inputs.

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, the indentation is always four spaces. The body can contain any number of statements.

The strings in the print statements are enclosed in 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.

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

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

To end the function, you have to enter an empty line (this is not necessary in a script).

Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)  
<function print_lyrics at 0xb7e99e9c>  
>>> print(type(print_lyrics))  
<class 'function'>
```

The value of `print_lyrics` is a *function object*, which has type "function".

The syntax for calling the new function is the same as for built-in functions:

```
>>> 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` :

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

And then call `repeat_lyrics` :

```
>>> 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.
```

But that's not really how the song goes.

You Try it
(post a screenshot of your program from Python.org)

Exercises

Exercise 4: What is the purpose of the "def" keyword in Python?

- a) It is slang that means "the following code is really cool"
- b) It indicates the start of a function
- c) It indicates that the following indented section of code is to be stored for later
- d) b and c are both true
- e) None of the above

You Try it

(post a screenshot of your program from Python.org)

Exercise 5: What will the following Python program print out?

```
def fred():  
    print("Zap")  
  
def jane():  
    print("ABC")  
  
jane()  
fred()  
jane()
```

- a) Zap ABC jane fred jane
- b) Zap ABC Zap
- c) ABC Zap jane
- d) ABC Zap ABC
- e) Zap Zap Zap

You Try it

(post a screenshot of your program from Python.org)

Exercise 7: Write a grade program using a function called `compute_grade` that takes a score from 0 to 100 as its parameter and returns a grade as a string.

Score	Grade
> 90	A
> 80	B
> 70	C
> 60	D
<= 60	F

Program Execution:

```
Enter score: 95
A
```

```
Enter score: perfect
Bad score
```

```
Enter score: 1000.0
Bad score
```

```
Enter score: 75
C
```

```
Enter score: 5
F
```

Hint:
try :
except: