Functions

+6
Published May 6, 2021•**Updated Feb 13, 2023**

Contribute to Docs

Some tasks need to be performed multiple times within a program. Rather than rewrite the same code in multiple places, a function may be defined using the `def` keyword. Function definitions may include parameters, providing data input to the function.

Syntax

```
def my_function(value):
  return value + 1

print(my_function(2))
print(my_function(3 + 5))
```

Functions may return a value using the `return` keyword followed by a `value`. They can then be called, or invoked, elsewhere in the program. The output from the snippet above would look like this:

```
3
9
```

Note: Function names in Python are written in snake_case.

Return Values

The `return` keyword is used to return a value from a Python function. The value returned from a function can be assigned to a variable which can then be used in the program.

In the example below, the `check_leap_year()` function returns a string that indicates if the passed parameter is a leap year or not.

```
def check_leap_year(year):
 if year % 4 == 0:
  return str(year) + " is a leap year."
 else:
  return str(year) + " is not a leap year."

year_to_check = 2018

returned_value = check_leap_year(year_to_check)

print(returned_value)
```

The resulting output will look like this:

```
2018 is not a leap year.
```

Returning Values With `yield`

A function can also return values with the `yield` keyword.
Like `return`, `yield` suspends the function's execution and returns the value specified. Unlike `return`, the `yield` statement retains the state of the function and will resume where it left off on the next function call (i.e. execution resumes after the last `yield` statement). This way, the function can produce a number of values over time.

Functions using `yield` rather than `return` are known as generator functions. Such a function can be used as an iterator.

The example below will automatically generate successive Fibonacci numbers.

```
# Function to produce infinite Fibonacci numbers
def fibonacci():
 # Generate first number
```

```
    a = 1
    yield a

    # Generate second number
    b = 1
    yield b

    # Infinite loop
    while True:
      # Return sum of a + b
      c = a + b
      yield c
      # Function resumes loop here on next call
      a = b
      b = c

# Iterate through the Fibonacci sequence until a limit is reached
for num in fibonacci():
    if num > 50:
      break
    print(num)
```

This will output the following:

```
1
1
2
3
5
8
13
21
34
```

Higher-Order Functions

In Python, functions are treated as first-class objects. This means that they can be assigned to variables, stored in data structures, and passed to or returned from other functions.

Functions are considered to be “higher-order” values because they can be used as parameters or return values for other functions. One example is the built-in `filter()` function:

```
# Returns True if n is a perfect square, and False otherwise

def is_perfect_square(n):
  return (n ** 0.5).is_integer()

numbers = [3, 4, 37, 9, 7, 32, 25, 81, 79, 100]

perfect_squares = filter(is_perfect_square, numbers)

print(list(perfect_squares))
```

`filter()` takes a predicate (a function that returns a boolean value) and an iterable, and returns a new iterable containing all elements of the first one that makes the predicate true.

Functions

Anonymous Functions

Defines a function without a name using the lambda keyword.

Arguments/Parameters

Supplies data to a defined function when it is called in a program.