CS101 - Functions with Parameters and Return Values
Lecture 4

School of Computing
KAIST
Roadmap

Last week we learned

- Objects
- Types
- Variables
- Methods
- Tuples
Roadmap

Last week we learned

- Objects
- Types
- Variables
- Methods
- Tuples

This week we will learn

- Functions with parameters and return values
The name **function** comes from mathematics. A function is a mapping from one set to another set:

\[ f : \mathbb{R} \rightarrow \mathbb{R} \]

\[ x \rightarrow \pi \times \frac{x}{180.0} \]
Functions

The name function comes from mathematics. A function is a mapping from one set to another set:

\[ f : \mathbb{R} \rightarrow \mathbb{R} \]
\[ x \rightarrow \pi \times \frac{x}{180.0} \]

Here, \( x \) is the argument of the function, \( f(x) \) is the result of the function.
Functions

The name **function** comes from mathematics. A function is a mapping from one set to another set:

\[ f : \mathbb{R} \rightarrow \mathbb{R} \]

\[ x \rightarrow \pi \times \frac{x}{180.0} \]

Here, \( x \) is the **argument** of the function, \( f(x) \) is the **result** of the function.

In Python, functions also take **arguments** and return a **result**:

```python
def to_radians(deg):
    return (deg / 180.0) * math.pi
```
Functions

The name **function** comes from mathematics. A function is a mapping from one set to another set:

\[ f : \mathbb{R} \rightarrow \mathbb{R} \]

\[ x \rightarrow \pi \times \frac{x}{180.0} \]

Here, \( x \) is the **argument** of the function, \( f(x) \) is the **result** of the function.

In Python, functions also take **arguments** and return a **result**:

```python
def to_radians(deg):
    return (deg / 180.0) * math.pi

>>> a = to_radians(90)
>>> print(a)
1.5707963267948966
```
Useful functions

Python comes with many built-in functions.
Useful functions

Python comes with many built-in functions.

**Type conversion functions** convert from one type to another type:

```python
>>> int("32")
32

>>> int(17.3)
17

>>> float(17)
17.0

>>> float("3.1415")
3.1415

>>> str(17) + " " + str(3.1415)
'17 3.1415'

>>> complex(17)
(17 + 0j)
```
Math functions

To use math functions, we need to tell Python that we want to use the `math` module:

```python
import math
degrees = 45
radians = degrees / 360.0 * 2 * math.pi
print(math.sin(radians))
print(math.sqrt(2) / 2)
```
Math functions

To use math functions, we need to tell Python that we want to use the `math` module:

```python
import math
degrees = 45
radians = degrees / 360.0 * 2 * math.pi
print(math.sin(radians))
print(math.sqrt(2) / 2)
```

When using math functions often, we can use shorter names:

```python
import math
sin = math.sin
pi = math.pi
radians = degrees / 360.0 * 2 * pi
print(sin(radians))
```
Defining functions with parameters

The function definition uses **names** for the arguments of the function. These names are called **parameters**:

```python
def compute_interest(amount, rate, years):
```
Defining functions with parameters

The function definition uses names for the arguments of the function. These names are called parameters:

```python
def compute_interest(amount, rate, years):
    value = amount * (1 + rate/100.0) ** years
```

Inside the function, the parameter is just a name:
Defining functions with parameters

The function definition uses **names** for the arguments of the function. These names are called **parameters**:

```python
def compute_interest(amount, rate, years):
```

Inside the function, the parameter is just a name:

```python
value = amount * (1 + rate/100.0) ** years
```

When we have computed the result of the function, we **return** it from the function. The function ends at this point, and the result object is given back:

```python
return value
```
Defining functions with parameters

The function definition uses **names** for the arguments of the function. These names are called **parameters**:

```python
def compute_interest(amount, rate, years):
```

Inside the function, the parameter is just a name:

```python
    value = amount * (1 + rate/100.0) ** years
```

When we have computed the result of the function, we **return** it from the function. The function ends at this point, and the result object is given back:

```python
    return value
```

We can now call the function with different argument values:

```python
>>> s1 = compute_interest(200, 7, 1)
>>> s2 = compute_interest(500, 1, 20)
```
Converting to black-and-white

What is the light intensity (luma) of pixel \((r,g,b)\)?
Converting to black-and-white

What is the light intensity (\textit{luma}) of pixel (r,g,b)?

<table>
<thead>
<tr>
<th>r</th>
<th>g</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
</tbody>
</table>
Converting to black-and-white

What is the light intensity (luma) of pixel \((r,g,b)\)?

\[
\begin{align*}
(255, 0, 0) & \quad (0, 255, 0) & \quad (0, 0, 255) \\
\text{Red} & \quad \text{Green} & \quad \text{Blue}
\end{align*}
\]

A good formula is:

```python
def luma(p):
    r, g, b = p
    return int(0.213 * r + 0.715 * g + 0.072 * b)
```
More than one return in a function

Compute the absolute value (like builtin function **abs**):

```python
def absolute(x):
    if x < 0:
        return -x
    else:
        return x
```
More than one return in a function

Compute the absolute value (like builtin function \texttt{abs}):

\begin{verbatim}
def absolute(x):
    if x < 0:
        return -x
    return x
\end{verbatim}

The same function can be written like this:

\begin{verbatim}
def absolute(x):
    if x < 0:
        return -x
    return x
\end{verbatim}
More than one return in a function

Compute the absolute value (like builtin function $\texttt{abs}$):

```python
def absolute(x):
    if x < 0:
        return -x
    else:
        return x
```

The same function can be written like this:

```python
def absolute(x):
    if x < 0:
        return -x
    return x
```

But not like this:

```python
def absolute(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```
Returning a boolean

A function that tests a condition and returns either True or False is often called a predicate:

```python
# is integer a divisible by b?
def is_divisible(a, b):
    if a % b == 0:
        return True
    else:
        return False
```
Returning a boolean

A function that tests a condition and returns either True or False is often called a predicate:

```python
# is integer a divisible by b?

def is_divisible(a, b):
    if a % b == 0:
        return True
    else:
        return False
```

A predicate (function) can be used directly in an if or while statement:

```python
if is_divisible(x, y):
    print('x is divisible by y')
```
Returning a boolean

A function that tests a condition and returns either True or False is often called a predicate:

```python
# is integer a divisible by b?
def is_divisible(a, b):
    if a % b == 0:
        return True
    else:
        return False
```

A predicate (function) can be used directly in an if or while statement:

```python
if is_divisible(x, y):
    print('x is divisible by y')
```

Easier:

```python
def is_divisible(a, b):
    return a % b == 0
```
Functions without results

We have seen many functions that do not use `return`:

```python
def turn_right():
    for i in range(3):
        hubo.turn_left()
```
We have seen many functions that do not use `return`:

```python
def turn_right():
    for i in range(3):
        hubo.turn_left()
```

In fact, a function that does not call `return` automatically returns `None`:

```python
>>> s = turn_right()
>>> print(s)
None
```
Calling functions

When a function is called, the arguments of the function call are assigned to the parameters:

```python
def print_twice(text):
    print(text)
    print(text)
```
Calling functions

When a function is called, the **arguments** of the function call are assigned to the **parameters**:

```python
def print_twice(text):
    print(text)
    print(text)
```

The number of arguments in the function call must be the same as the number of parameters.

```python
>>> print_twice("I love CS101")
I love CS101
I love CS101
```

```python
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```
Hubo’s family

We can now write a `turn_right` function that will work for any robot, not just for Hubo:

```python
def turn_right(robot):
    for i in range(3):
        robot.turn_left()

ami = Robot("yellow")
hubo = Robot("blue")
turn_right(ami)
turn_right(hubo)
```
Hubo’s family

We can now write a `turn_right` function that will work for any robot, not just for Hubo:

```python
def turn_right(robot):
    for i in range(3):
        robot.turn_left()

ami = Robot("yellow")
hubo = Robot("blue")
turn_right(ami)
turn_right(hubo)
```

Remember: A **parameter** is a **name** for an object. The name can only be used **inside** the function.
Harvesting again
Harvesting again

\[ \text{stairs(hubo, 5)} \]
Harvesting again

stairs(hubo, 3)

stairs(hubo, 5)
Harvesting again

stairs(hubo, 1)
stairs(hubo, 3)
stairs(hubo, 5)
Harvesting again

stairs(hubo, 5)  stairs(hubo, 5)
stairs(hubo, 5)  stairs(hubo, 5)
Harvesting again

```python
def stairs(robot, n):
    for i in range(n):
        robot.pick_beeper()
        robot.move()
        turn_right(robot)
        robot.move()
        robot.turn_left()

def diamond(robot, n):
    for i in range(4):
        stairs(robot, n)
        robot.turn_left()

def harvest_all(robot):
    for i in range(3):
        n = 5 - 2 * i
        diamond(robot, n)
        robot.move()
        robot.move()
```

14 / 17
Converting to black and white, again

white = (255, 255, 255)
black = (0, 0, 0)

def blackwhite(img, threshold):
    w, h = img.size()
    for y in range(h):
        for x in range(w):
            v = luma(img.get(x, y))
            if v > threshold:
                img.set(x, y, white)
            else:
                img.set(x, y, black)

pict = load_picture("../photos/yunal1.jpg")
blackwhite(pict, 100)
pict.show()
Returning more than one value

A function can only return one value.
Returning more than one value

A function can only return one value.

But this value can be a tuple, and functions can return arbitrarily many values by returning them as a tuple:

def student():
    name = "Hong, Gildong"
    id = 20101234
    return name, id
Returning more than one value

A function can only return one value.

But this value can be a tuple, and functions can return arbitrarily many values by returning them as a tuple:

```python
def student():
    name = "Hong, Gildong"
    id = 20101234
    return name, id
```

Often function results are unpacked immediately:

```python
name, id = student()
```
Keyboard input

The `input` function prints a message and waits for the user to enter a string on the keyboard. When the user presses the Enter key, the whole string is returned:

```python
name = input("What is your name? ")
print("Welcome to CS101, " + name)
```
Keyboard input

The `input` function prints a message and waits for the user to enter a string on the keyboard. When the user presses the Enter key, the whole string is returned:

```python
name = input("What is your name? ")
print("Welcome to CS101, " + name)
```

If we need a number, we should convert the string:

```python
raw_n = input("Enter a positive integer> ")
n = int(raw_n)
for i in range(n):
    print("*" * i)
```