01-24-2022

CS051A

INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

2: Functions

David Kauchak

he/him/his

Lectures

Alexandra Papoutsaki

she/her/hers

Lectures

Zilong Ye

he/him/his

Labs

Lecture 2: Functions

- Administrative
- Syntax
- Functions

This week

- First (online) lab today or tomorrow.
  - Zoom link in Sakai announcement.
  - Read handout beforehand.
    - Installation of PyCharm.
    - Practice with running programs and using the Python shell (console).
- First assignment due this coming Sunday.
- Mentor sessions begin today.
  - Check schedule on course website. TAs' Zoom links on Slack.

Lecture 2: Functions

- Administrative
- Syntax
- Functions I
- Strings
- Functions II

Syntax in English and in programming languages

- In English: arrangement of words and phrases to create well-formed sentences.
  - "I like dogs" is syntactically correct
  - "I dog like" is not syntactically correct
- Programming languages also have their own syntax, that is their own rules of what code valid in that language.
- In contrast to English, programming languages are less forgiving to syntax errors: the computer won't get the gist of our program

Syntax errors

- In our brief Python adventures, have we seen any examples of syntax?
  - All of the math operations have implicit syntax, e.g., we can't just write `4 +` and leave it as is.
- Similar issues can arise if we flip an assignment.

```
>>> 2 = jasmine
Traceback (most recent call last):
  File "/usr/local/Cellar/python@3.9/3.9.1_1/Frameworks/Python.framework/Versions/3.9/lib/python3.9/code.py"
    code = self.compile(source, filename, symbol)
  File "/usr/local/Cellar/python@3.9/3.9.1_1/Frameworks/Python.framework/Versions/3.9/lib/python3.9/codeop.p
    return _maybe_compile(self.compiler, source, filename, symbol)
  File "/usr/local/Cellar/python@3.9/3.9.1_1/Frameworks/Python.framework/Versions/3.9/lib/python3.9/codeop.p
    raise err1
  File "/usr/local/Cellar/python@3.9/3.9.1_1/Frameworks/Python.framework/Versions/3.9/lib/python3.9/codeop.p
    code1 = compiler(source + "\n", filename, symbol)
  File "/usr/local/Cellar/python@3.9/3.9.1_1/Frameworks/Python.framework/Versions/3.9/lib/python3.9/codeop.p
    codeob = compile(source, filename, symbol, self.flags, True)
  File "<input>", line 1
    2 = jasmine
    ^
SyntaxError: cannot assign to literal
```

Syntax errors in PyCharm

- Both the Python shell (console) and program mode will recognize syntax errors.
- Luckily, *most of the times,* Python will indicate the correct line that the syntax error occurred.
- Sometimes though, an error might be elsewhere, probably before the line that is highlighted.

Lecture 2: Functions

- Administrative
- Syntax
- Functions

Beyond a basic math calculator

- What other mathematical operations might we want from our calculator?
  - `abs`, `round`, `min`, `max` etc.
- These operations are supported by functions.

Functions

- A function in Python has:
  - A name
  - Zero or more parameters (i.e. inputs)
  - Generally, does something
  - Gives us back a value (not all functions do this)

Built-in Python functions

- `abs(number)`: returns the absolute value of the specified number
  - e.g., `abs(-2)` will return 2.
- `round(number)`: returns a number that is the rounded version of the specified number.
  - e.g., `round(2.8)` will return 3.
- `int(number)`: returns the integer part of a specified number by throwing away decimals.
  - e.g., `int(2.8)` will return 2.

Function type

- type(expression): returns type of the specific expression. E.g.,

```
>>> type(10)
<class 'int'>
>>> type(10/2)
<class 'float'>
>>> type(10//2)
<class 'int'>
```

- Note that all of these functions take a single parameter and give us back (return) a value,
- We'll talk more later about what it means to "give back" a value, but some functions will simply "do something" and then not return a value.

Defining your own functions

- Allows you to bundle together code and reuse it.
- Remember, the important components for a function are:
  - the name of the function,
  - how many parameters the function takes,
  - what the function does,
  - what value (if any) it returns/gives you back when it's done.

Syntax for defining your own functions

```
def function_name(parameter1, parameter2, ...) :
        statement1
        statement2
        ...
        return expression # this is optional
```

- `function_name` is the name of the function (i.e. what you want it to be called)
- `parameter1`, `parameter2`, ... are the list of parameters that are expected
    - you can use the parameters in the body of the function like variables.
    - when you call the function, the number of parameters specifies the number you must supply in the function call
- the spacing (tab) indicates which statements are within a function (called the "body" of the function).
- The `return` statement is the value that we want to return to whoever called the function. It's not necessary to have a return statement.

Anything new here?

simple-functions.py

```
def dog_years(human_years):
    return human_years * 7


def dog_name():
    return "Fido"


def interest_calculator(money, rate_percent):
    rate = rate_percent * .01
    return money * rate


def dog_stats(years):
    name = dog_name()
    age = dog_years(years)
    return name + " is " + str(age) + " years old"


def say_my_name():
    name = "Dave"
    return name

```

Strings

- `string`: a “string” of characters.
  - Represent text.
  - Denoted by quotes
    - You can either use double quotes, e.g., `"this is a string"` or single quotes `'this is also a string'`
    - But you can't mix the two for any given string `'this is not a valid string"`
- A new type (we have seen `int`, and `float`)

Writing code with strings

- If you want to concatenate (i.e. combine) two strings, you can use the plus sign. E.g.,

```
>>> "this is one string " + "plus another one"
'this is one string plus another one'
```

- If you want to use an `int` or a `float` as a `string`, you need to convert it to a `string` using `str` function.

```
>>> x=4
>>> "The value of x is " + str(x)
'The value of x is 4'
>>> "The value of x is " + x
Traceback (most recent call last):
  File "/usr/local/Cellar/python@3.9/3.9.1_1/Frameworks/Python.framework/Versions/3.9/lib/python3.9/code
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

What do the first three functions do?

- `dog_years`: calculates the number of dog years, given human years.
- `dog_name`: returns the name of the dog (in this case "Fido"), as a string.
- `interest_calculator`: calculates the amount of interest earned for a given amount of money at a particular rate.

Calling functions

- If we “Run file in Python console", what do you think will happen?
  - nothing gets printed out!
- but we've now defined new functions that we can use:

```
>>> dog_years(7)
49
>>> dog_name()
'Fido'
>>> interest_calculator(1000, 3)
30.0
```

Parameters and arguments

- parameter: variable listed inside the parentheses in the function definition.
- argument: the value passed to the function when calling it.
- Notice that the number of parameters defines how many arguments we must specify
- When a function is called:
  - we evaluate each of the arguments
  - then we execute the function line by line
  - if there is a `return` statement, where the original function call was made is replaced by the value returned.

Number of parameters and arguments have to agree

- If we try to call one of our functions with the wrong number of arguments, we get an error.

```
>>> interest_calculator(1000)
Traceback (most recent call last):
  File "/usr/local/Cellar/python@3.9/3.9.1_1/Frameworks/Python.framework/Versions/3.9/lib/python3.9/code.py"
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: interest_calculator() missing 1 required positional argument: 'rate_percent'
```

- the last line is the most important and tells us what the error was, i.e. that the function takes 2 arguments, but we only gave it 1.
- If we knew that we wanted to call some of these functions, we could also add this code to the end of the file and then that would get executed when we run it.

`dog_stats` function

- Say we call the function as `dog_stats(2+5)`. The following will happen:
  - First, we'll evaluate the argument to the function (2+5) and get 7
    - 7 will then get associated with the parameter `years`
  - The first statement in the function calls our other function, `dog_name()`
    - the interpreter will go to the `dog_name` function and execute its code as defined in its body.
      - `dog_name` will return `"Fido"`, which will then get stored into the variable `name`.
- The second statement is a call to the `dog_years` function
  - We evaluate its argument (`years`), which gives us 7
  - 7 is then passed to `dog_years`
  - 7 is associated with the parameter `human_years`
  - `7*7` is calculated, giving us 49, which is returned
  - 49 is then stored in the variable `age`.
- Finally, we return the string `"Fido" + " is " + "49" + " years old"` -> `"Fido is 49 years old"`

Advanced BBQing

- If we look back at our bbq code, we notice that we only need information from some of the people to calculate the number of hot dogs.
- only `angie` and `jasmine` affect the number of hotdogs required.

hotdogs method in bbq-functions.py

```
def hotdogs(angie, jasmine):
    chris = 2 * jasmine
    brenda = chris - 1
    wenting = (brenda + 1) // 2 + 1  # add 1 to brenda to round up

    total_hotdogs = angie + jasmine + chris + brenda + wenting
    return total_hotdogs
```

- Look at the hotdogs method in bbq-functions.py. What does it do?
  - same thing as our bbq program, just now we've encapsulated it as a program where we can pass it parameters.

hotdogs method in bbq-functions.py

```
def hotdogs(angie, jasmine):
    chris = 2 * jasmine
    brenda = chris - 1
    wenting = (brenda + 1) // 2 + 1  # add 1 to brenda to round up

    total_hotdogs = angie + jasmine + chris + brenda + wenting
    return total_hotdogs
```

- Look at the hotdogs method in bbq-functions.py. It does as our bbq program, just now we've encapsulated it as a program where we can pass it parameters.

```
>>> hotdogs(1,2)
13
>>> hotdogs(1,3)
19
```

Rest of the methods in `bbq-functions.py`?

- Look at the other functions in `bbq-functions.py` code: what do they do?
  - The rest of these functions don't really have any new features from the ones we've previously seen.
- Notice that we can build up more complicated functions by using the simpler functions.
- Don't forget that if you want to combine a `string` and an `int/float`, you need to convert the `int/float` to a `string` using the `str` method.

Look at `bbq-functions-bad-style.py`

- What does this code do?
  - These have the same functionality as the first three functions in `bbq-functions.py` code.
    - But... they're much harder to read and understand.
- Use good variable names, good function names and whitespace to help make the code more readable (this is called using good style)!

Resources

- Textbook: Continue reading Chapter 1 and 2
- simple-functions.txt
- bbq-functions.txt
- bbq-functions-bad-style.txt

Homework

- (Work in progress) - Assignment 1