PROGRAMMING IN PYTHON I

Unit 03: Functions, Print, Modules

Michael Widrich & Sebastian Lehner
Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

PYTHON VARIABLES: IMPLICATIONS

Python variables: Recap

- We already learned: Python variables are actually references to an object in the background
  - Object keeps information on its datatype, stored bits, and reference count
  - Allows for dynamic typing and other shenanigans
  - Assigning a value to a variable changes which object the variable references
- Lists in Python can group multiple variables
  - Elements correspond to “variables” without names
  - Lists elements are accessed (indexed) using square brackets `[]`
  - In-place operations, such as `.append()`, modify a list in-place (often only returning `None`)

    ```
    # Changes my_list and returns None:
    my_list.append(4)
    ```

Python variables: Implications (1)

- Assume a list

```
my_list = ['a', 'b', 'c']
```

Q: What are the values of `var` and `my_list` after the next line?

```
var = my_list[0]
```

Python variables: Implications (1)

- Assume a list

  `my_list = ['a', 'b', 'c']`

Q: What are the values of `var` and `my_list` after the next line?

`var = my_list[0]`

A: `var` references object `'a'`, `my_list` is `['a', 'b', 'c']`

Python variables: Implications (1)

- Assume a list

  `my_list = ['a', 'b', 'c']`

Q: What are the values of `var` and `my_list` after the next line?

  `var = my_list[0]`

A: `var` references object `'a'`, `my_list` is `['a', 'b', 'c']`

Q: What are the values of `var` and `my_list` after the next line?

  `var = 5`

Python variables: Implications (1)

- Assume a list

  ```
  my_list = ['a', 'b', 'c']
  ```

Q: What are the values of `var` and `my_list` after the next line?

```
var = my_list[0]
```

A: `var` references object `'a'`, `my_list` is `['a', 'b', 'c']`

Q: What are the values of `var` and `my_list` after the next line?

```
var = 5
```

A: `var` references object `5`, `my_list` is `['a', 'b', 'c']`

Python variables: Implications (2)

- Assume a list

```
my_list = ['a', 'b', 'c']
```

Q: What are the values of `var` and `my_list` after the next lines?

```
var = my_list[0]
my_list[0] = 'd'
```

Python variables: Implications (2)

- Assume a list

```
my_list = ['a', 'b', 'c']
```

Q: What are the values of `var` and `my_list` after the next lines?

```
var = my_list[0]
```

```
my_list[0] = 'd'
```

A: `var` references object `'a'`, `my_list` is `['d', 'b', 'c']` (the first element now references object `'d'`)

Python variables: Implications (3)

- Assume a list

  ```
  my_list = ['a', 'b', 'c']
  ```

Q: What are the values of `var` and `my_list` after the next line?

```
var = my_list
```

Python variables: Implications (3)

- Assume a list

  `my_list = ['a', 'b', 'c']`

Q: What are the values of `var` and `my_list` after the next line?

  `var = my_list`

A: `var` and `my_list` both reference the same list object `['a', 'b', 'c']` (which references objects `'a'`, `'b'`, and `'c'`)

Python variables: Implications (3)

- Assume a list

  `my_list = ['a', 'b', 'c']`

Q: What are the values of `var` and `my_list` after the next line?

`var = my_list`

A: `var` and `my_list` both reference the same list object `['a', 'b', 'c']` (which references objects `'a'`, `'b'`, and `'c'`)

Q: What are the values of `var` and `my_list` after the next line?

`var[0] = 'e'`

Python variables: Implications (3)

- Assume a list

  `my_list = ['a', 'b', 'c']`

Q: What are the values of `var` and `my_list` after the next line?

  `var = my_list`

A: `var` and `my_list` both reference the same list object `['a', 'b', 'c']` (which references objects `'a'`, `'b'`, and `'c'`)

Q: What are the values of `var` and `my_list` after the next line?

  `var[0] = 'e'`

A: `var` and `my_list` still both reference the same list object `['e', 'b', 'c']` (which now references objects `'e'`, `'b'`, and `'c'`)

Python variables: Implications (4)

- Assume a list

```
my_list = ['a', 'b', 'c']
```

Q: What are the values of `var` and `my_list` after the next line?

```
var = my_list.copy()
```

Python variables: Implications (4)

- Assume a list

  `my_list = ['a', 'b', 'c']`

Q: What are the values of `var` and `my_list` after the next line?

   `var = my_list.copy()`

A: `var` and `my_list` reference different list objects `['a', 'b', 'c']` (both lists reference objects `'a'`, `'b'`, and `'c'`)

JKU

Python variables: Implications (4)

- Assume a list

  ```
  my_list = ['a', 'b', 'c']
  ```

Q: What are the values of `var` and `my_list` after the next line?

```
var = my_list.copy()
```

A: `var` and `my_list` reference different list objects `['a', 'b', 'c']` (both lists reference objects `'a'`, `'b'`, and `'c'`)

Q: What are the values of `var` and `my_list` after the next line?

```
var[0] = 'e'
```

Python variables: Implications (4)

- Assume a list

  `my_list = ['a', 'b', 'c']`

Q: What are the values of `var` and `my_list` after the next line?

  `var = my_list.copy()`

A: `var` and `my_list` reference different list objects `['a', 'b', 'c']` (both lists reference objects `'a'`, `'b'`, and `'c'`)

Q: What are the values of `var` and `my_list` after the next line?

  `var[0] = 'e'`

A: `var` is `['e', 'b', 'c']` and `my_list` is `['a', 'b', 'c']`

Python variables: Implications (5)

- We need to be careful when multiple variables reference the same list object
  - Assigning the same list object to multiple variables will not duplicate the list object
  - Changing elements of a list object will affect all variables referencing this list object
- We can create actual copies of the list object via `.copy()`
  - For nested lists, this is only a shallow copy
  - If we want to copy a list and all lists it includes as elements, we need a deep copy
  - See `https://docs.python.org/3.7/library/copy.html` for shallow and deep copies
- The same behavior holds for dictionaries and other mutable Python objects

FUNCTIONS

Motivation

- Often we encounter similar problems multiple times

→ We need to perform the same sequence of operations again and again

- However, we do not want to repeat code! Why are redundancies bad?
    - Prone to errors
    - Make program long, which means more to read
    - More difficult to maintain (need to change all redundant code-parts for updates)

Motivation

- Often we encounter similar problems multiple times

→ We need to perform the same sequence of operations again and again

- However, we do not want to repeat code! Why are redundancies bad?
  - Prone to errors
  - Make program long, which means more to read
  - More difficult to maintain (need to change all redundant code-parts for updates)

```
# Calculate the distance between (2,1)and (1,1)
dist1 = math.sqrt((2-1)**2+(1-1)**2)
# Calculate the distance between (4,0)and (0,4)
dist2 = math.sqrt((4-0)**2+(0-4)**2)
# Many more repetitions...
```

Motivation

- Preferred solution: Function with input and output
  - Parameters: Takes the coordinates of two points as input
  - Outputs: Function returns the distance

```
def getdist(x1,y1,x2,y2):
  result = math.sqrt((x1-x2)**2+(y1-y2)**2)
  return result
# Calculate the distance between (2,1)and (1,1)
dist1 = getdist(2,1,1,1)
# Calculate the distance between (4,0)and (0,4)
dist2 = getdist(4,0,0,4)
# Many more repetitions...
```

- Finding bugs much easier - only one formula to check!

Functions and Variables (1)

- Functions in Python:
  - The following defines a function with name `fun` that takes one argument `b` and returns the value of `c`
    ```
    def fun(b):
      c = b*2
      return c
    ```
  - We can then call the function like this:
    ```
    fun(1)# returns 2
    ```
  - The variables defined within the function only exists within the function
  - The assignment `b=1` takes place as `fun(1)` is called (call-by-object)

Functions and Variables (2)

- Example:

```
def fun(b):
  b = b*2
  return b
a = 5
c = fun(a)
```

- Argument `b` is a local variable in the function
- Effectively, the assignment `b=a` takes place as `fun(a)` is called (call-by-object)
- What is the value of `a` and `c`?

Functions and Variables (2)

- Example:

```
def fun(b):
  b = b*2
  return b
a = 5
c = fun(a)
```

- Argument `b` is a local variable in the function
- Effectively, the assignment `b=a` takes place as `fun(a)` is called (call-by-object)
- What is the value of `a` and `c`?
- `a` is still referencing object `5`, `c` is referencing object `10`

Functions and Variables (3)

- In the following we discuss several important rules concerning the interplay between variables and functions
- Example:

```
a = 5
def fun(b):
    result = a-b
    return result
fun(1)
```

- Would this work? Is `a` visible inside the function definition?

Functions and Variables (3)

- In the following we discuss several important rules concerning the interplay between variables and functions
- Example:

```
a = 5
def fun(b):
  result = a-b
  return result
fun(1)
```

- Would this work? Is `a` visible inside the function definition?
- Yes! `fun(1)` would return 4
- Variables outside the function definition are visible inside the function scope

Functions and Variables (4)

- Another example:

```
a = 2
def fun(b):
    a = 10
    result = a-b
    return result
```

- Would this work?

Functions and Variables (4)

- Another example:

```
a = 2
def fun(b):
    a = 10
    result = a-b
    return result
```

- Would this work?
- Yes! `fun(1)` would return 9
- What is the value of `a` now?

Functions and Variables (4)

- Another example:

```
a = 2
def fun(b):
    a = 10
    result = a-b
    return result
```

- Would this work?
- Yes! `fun(1)` would return 9
- What is the value of `a` now?
- Still 2, variables defined in function definitions are local!
  - The function has its own variable scope (it shadows variable `a`)

Functions and Variables (5)

- Let’s exchange the first two lines in the function definition:

```
a = 2
def fun(b):
  result = a-b
  a = 10
  return result
```

- Would this work?

Functions and Variables (5)

- Let's exchange the first two lines in the function definition:

```
a = 2
def fun(b):
  result = a-b
  a = 10
  return result
```

- Would this work?
- No. (`UnboundLocalError`)
- Using a variable from outside the function scope and then redefining it is not allowed
- Recommendation: Provide all relevant variables as function parameters

[More information: `https://docs.python.org/3.7/faq/programming.html#why-am-i-getting-an-unboundlocalerror-when-the-variable-has-a-value`]

Functions and Variables (6)

- For mutable arguments, we have to pay attention:

```
def fun(b):
    b[0]=10
    return b
a = [5,6]
c = fun(a)
```

- What is the value of `a` and `b` now?

Functions and Variables (6)

- For mutable arguments, we have to pay attention:

  ```
  def fun(b):
    b[0]=10
    return b
  a = [5,6]
  c = fun(a)
  ```

- What is the value of `a` and `b` now?
- `a` and `b` reference the same list object `[10,6]`

[More information: `https://docs.python.org/3.7/tutorial/controlflow.html#defining-functions`]

MODULES

Modules

- Often we want to re-use code, e.g. a function, in different programs and projects
- In Python we can do so by putting the function into a separate file (module)
- We can then load (import) this function definition from the file into our code file
- There are many modules with lots of functionalities available
  - You will write your own modules
  - We will learn about some important modules

[More information: `https://docs.python.org/3.7/tutorial/modules.html`]