Python Programming for Data Science

Week 37, Friday

Debugging code

Writing functions

Recap: loops: a popquiz

- What does a `while` loop do?
- What does a `for` loop do?
- What does this code do?:

```
while True:
    print("hello")
```

- What is the difference between a tuple and a list?
- What is the difference between a list and a dictionary?

A few comments regarding handin1 (1)

It looks like almost all of you got past the 50% grade mark. Congrats!
One tricky thing that we also wrote about in an announcement: the `print()` function adds a newline when its printing. If the string you are printing already has a newline, it will print an empty line:

```
print('hello\n')   # this will print an empty line
```

You can use `end=''` to stop `print()` from adding a new line

```
print('hello\n', end='')   # this will not print an empty line
```

A few comments regarding handin1 (2)

Q: But how do you know which invisible characters there are in a string?
A: you can print them out using `repr()`:

```
print(repr(line))
```

```
output
```

```
'  2000      0.367       0.047          0.482          0.041          0.317       0.050          0.432          0.043\n'
```

For a complete walk-through of Handin1, come to the Handin Help session this afternoon at 13:15

Debugging code

How do you find bugs in your program?

Simplest tricks in the book:

- Use print() to explore variable values
- Comment lines out

Simple debugging: printing

```
number_list = [1,2,3,4]
i = 0
while i < len(number_list):
    if i < 2:
        del number_list[i]
    i += 1

print(number_list)
```

Output:

```
[2, 4]
```

This is unexpected. What do we do?
An easy strategy is to use `print` to inspect the values of variables.

Simple debugging: printing

```
number_list = [1,2,3,4]
i = 0
while i < len(number_list):
    if i < 2:
        print (i, number_list)        # Inserted
        del number_list[i]
    i += 1
```

```
output

0 [1,2,3,4]
1 [2,3,4]
```

Ahhhhh... The indices shift once we start removing elements

Q: How can we solve this?

A: Iterate backwards (starting from the end of the list)

Debugging – commenting out code

This code does not run:

```
d = {'Mon':'monday','Tue':'tuesday'}
print(d['Wed'])
```

```
output
Traceback (most recent call last):
  File "days.py", line 1, in <module>
KeyError: 'Wed'
```

We can temporarily disable the faulty code by *commenting-out* the problematic line:

```
d = {'Mon':'monday','Tue':'tuesday'}
# print(d['Wed'])
```

Debugging – commenting out code (2)

We can now investigate the error

```
d={'Mon':'monday','Tue':'tuesday'}
# print(d['Wed'])
print(d.keys())
```

...and once it's fixed we can *comment-in* our line again.

```
d={'Mon':'monday','Tue':'tuesday','Wed':'wednesday'}
print(d['Wed'])
```

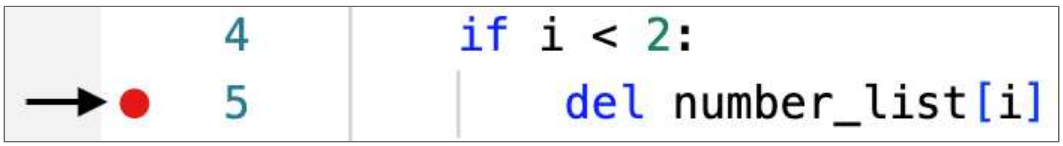
This is an efficient technique to quickly disable some code in your program.
Note that there is built-in support for commenting blocks of code in most editors. In VS Code, it's:
`Edit → Toggle Line Comment`

Debugging in VS Code

Using a *debugger*, you can inspect values without inserting print() lines
There is a debugger built into VS Code.
The following slides are a walk-through exercise. **Please follow along!**

VS Code debugger - breakpoints

You can tell program to temporarily pause its execution by inserting a *breakpoint*
In VS Code, you can set a breakpoint by clicking in the left margin of the editor window



In the code from the "Simple debugging: printing" slide, set a breakpoint in the same line as in this image. Then instead of the play button, we will now start the debugger...

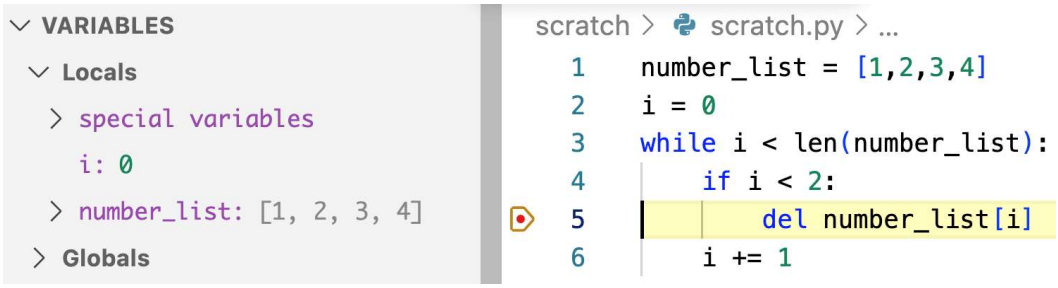
VS Code debugger - starting the debugger

Click on the arrow button next to the play button

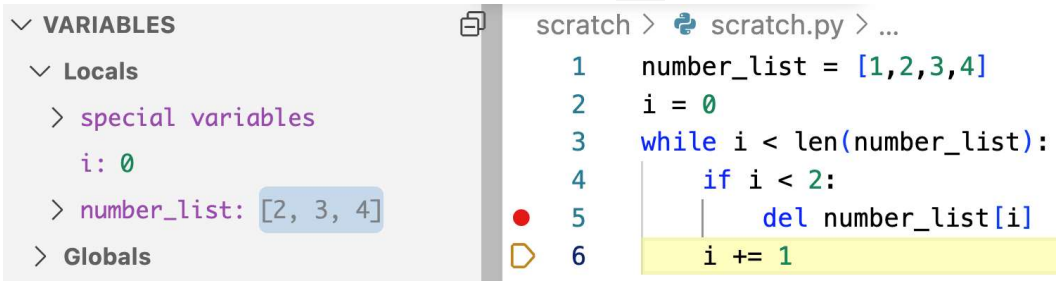
The debug window should now appear, and it will run your code until the breakpoint.

VS Code debugger - breakpoints (2)

The program stops *before* executing the line at the breakpoint



You can make the program move forward one line by clicking on the Step Over button ( ):



You can see that `number_list` is now one element shorter.

VS Code debugger - breakpoints (3)

Click "step over" one more time, we now see that the `i` index becomes 1, while the `number_list` is `[2,3,4]`, which means we have iterated past element 2.

VS Code debugger - exercise

Consider the following piece of code:

```
necessary_var = "hello"
if (len(necessary_var) > 4):
    neccessary_var = necessary_var[:4]  # truncate if length
larger than 4
print(necessary_var)
```

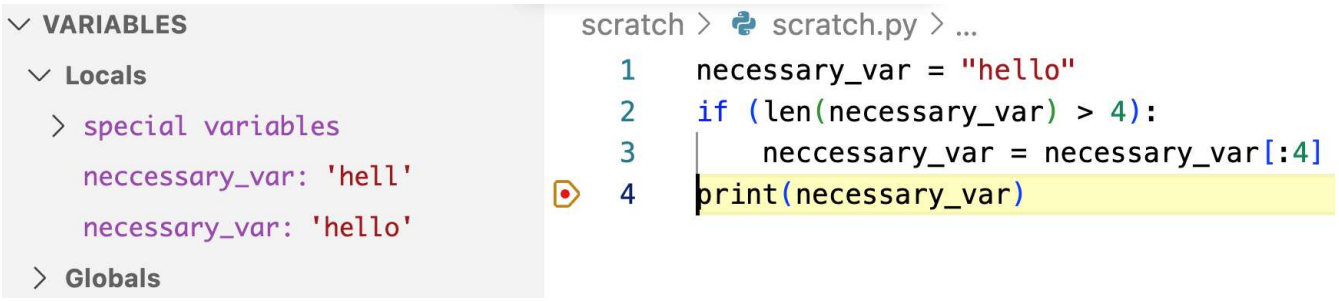
1. Discuss with your neighbor what the expected behavior of this program is, considering the comment in the next-to-last line.
2. Copy&paste the code into VS Code and run the code (without debugger). Does it behave as it should?
3. Use the VS Code debugger to find the error in the code

VS Code debugger - exercise - solution

1. *Discuss with your neighbor what the expected behavior of this program is, considering the comment in the next-to-last line.*
   We expect the output to be "hell"
2. *Copy&paste the code into VS Code and run the code (without debugger). Does it behave as it should?*
   No, it prints "hello"

VS Code debugger - exercise - solution (2)

3. *Use the VS Code debugger to find the error in the code*



Ahh...we misspelled "necessary"

Functions

Functions

A function is a way of bundling a piece of code and assigning it a name. It can hereafter be *called* multiple times with different input values.

Why functions?

1. Splitting code into small (named) pieces helps readability
2. Code reuse - avoid writing almost the same code many times

Functions - calling

Functions take *arguments* as input, and send back *return values*.

```
length_of_hello =
len('hello')
print(length_of_hello)
```

```
output

5
```

Here `'hello'` is an argument, and `5` is the return value

Functions - Can I define my own?

Functions are defined using the `def` keyword.

```
def function_name(argument1, argument2, ...):
    code block
```

(*argument1*, *argument2*, ...) is a tuple of variable names that are used to receive the values that you *pass* to the function when *calling* it.

```
def a_simple_function():          # defining a function with
no argument
    print("hello")

a_simple_function()               # calling the function
```

```
def function_with_argument(my_argument): # defining a
function with argument
    print(my_argument)

function_with_argument("bla bla")        # calling the
function
```

Functions - defining - example

```
def repeat_text(text, copies):
    for i in range(copies):
        print(text)

repeat_text("hello", 3)
```

```
output
hello
hello
hello
```

Functions - Exercise 1

1. Last week, we had an exercise where we created a die simulator, that would print numbers 1 to 6 at random. Take this code and turn it into a function called `die`, so that you can just call the function to print the result for a random throw of a die.

Functions - Exercise 1 - Solution

1. Turn the die simulator into a function called `die`, so that you can just call the function to print the result for a random throw of a die.

die.py

```
import random

def die():
    x = random.random()
    print(1+int(x*6))          # or simply
random.randint(1,6)

die()
die()
```

output

```
1
3
```

Functions - return values

So far, our functions only *print* their results to screen. They don't *return* anything (actually, they return None)

```
def add_two_numbers(x, y):
    print(x+y)

result =
add_two_numbers(2,3)
print(result)
```

```
output


5
None
```

We can change this by using the `return` statement inside our function.

```
def add_two_numbers(x, y):
    return x+y

result =
add_two_numbers(2,3)
print(result)
```

```
output


5
```

Functions with return values - Exercise

1. Change your die function so that it returns its result, instead of printing it
2. Create another function called `dice` that simulates throwing two dice and calculating the sum. This function should use the `die` function. Note how essential it is that the `die` function *returns* it's value instead of merely printing it.

Functions with return values - Exercise - solution

1. Change your die function so that it returns its result, instead of printing it

```
die_function.py
import random

def die():                          # define function
    x = random.random()
    return 1+int(x*6)               # or return
random.randint(1,6)

result = die()                      # call die function
print(result)
```

output

```
5
```

Functions with return values - Exercise - solution(2)

2. Create another function called `dice` that simulates throwing two dice. This function should use the `die` function.

```
dice_function.py
def dice():                     # define function
    d1 = die()                  # call die function
    d2 = die()                  # call die function again
    return d1+d2                # return result

result = dice()                 # call dice function
print(result)
```

output

```
7
```

Functions - several return values

Tuples are a natural way to return multiple values from a function:

```
def division(x, y):
    result = x//y
    remainder = x%y
    return result, remainder        # Returning a tuple of
two values

# print returned tuple
print(division(20, 3))

# Or, alternatively, use tuple assignment
res_division, res_remainder = division(20, 3)
print(res_division)
```

```
output
(6, 2)
6
```

Functions - Named arguments

We saw before that arguments to a function are just passed one by one in the order that they are specified. In Python, you can also name the arguments explicitly:

```
def division(x, y):
    result = x//y
    remainder = x%y
    return result, remainder

division(20, 3)       # Call with argument in correct
order
division(y=3, x=20)   # Call by naming arguments
```

The order of the arguments now no longer matters.

Functions - Optional arguments

You can specify optional arguments by giving them a default value.

repeat.py

```
def repeat(text, copies, new_lines=False):  # default value for new_lines
    new_text = ""
    for i in range(copies):
        new_text+=text
        if new_lines:
            new_text += "\n"
    return new_text
```

```
# Without specifying optional argument
print(repeat("hello", 3))
```

output

```
hellohellohello
```

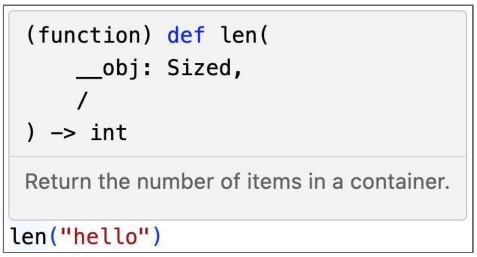
```
# Setting new_lines argument
print(repeat("hello", 3, True))
```

output

```
hello
hello
hello
```

Functions - Documenting

In VS Code, you can see a help message that explains what a function does, just by hovering over it



If you want others to be able to find help on the functions you write, specify a *doc-string* as the first line of your function definition.

```
def repeat_text(text, copies):
    '''Simple function that repeats a piece of text'''
    for i in range(copies):
        print(text)
```

In this course, we will expect to see doc-strings in all the functions that you hand in

Named/optional arguments - Exercise

1. Create a coin_toss function that simulates tossing a coin (i.e. returning either "heads" or "tails").
2. Change it so that it takes an argument specifying what the chance is of getting heads. Calling it like:

```
coin_toss(heads_prob=0.6)
```

   should create heads 60% of the time.
3. Change it again, so that this extra parameter is optional. If no argument is given the coin should be fair.

Named/optional arguments - Exercise - solution

1. Create a coin_toss function that simulates tossing a coin (i.e. returning either "heads" or "tails").

coin_toss_function.py

```
import random

def coin_toss():                        # define function
    x = random.random()                 # draw random
number
    if x < 0.5:                         # check whether it
is heads or tails
        return "heads"
    else:
        return "tails"

print(coin_toss())                       # call function
(no arguments)
```

output

```
tails
```

Named/optional arguments - Exercise - solution (2)

2. Change it so that it takes an argument specifying what the chance is of getting heads...

coin_toss_function2.py

```
import random

def coin_toss(heads_prob):        # define function with argument
    x = random.random()
    if x < heads_prob:            # determine whether it is heads or tails
        return "heads"
    else:
        return "tails"

print(coin_toss(0.9))             # call with different heads_prob
print(coin_toss(heads_prob=0.9))  # the same - but referring to argument by name
```

output

```
heads
heads
```

Named/optional arguments - Exercise - solution (3)

3. Change it again, so that this extra parameter is optional. If no argument is given the coin should be fair.

coin_toss_function3.py

```
import random

def coin_toss(heads_prob=0.5):    # define function
with optional argument
    x = random.random()
    if x < heads_prob:            # determine whether
it is heads or tails
        return "heads"
    else:
        return "tails"

print(coin_toss(heads_prob=0.9))  # this is still
possible
print(coin_toss())                # but now we can
also call without argument
```

output

```
tail
heads
```

Intermezzo...
...about variable names

Naming variables/functions/classes

1. Pick a meaningful name:
   `column_avg` is better than `x`
2. Use the Python naming convention:
   - Normal variables should be all lowercase
   - Long name should be separated with _

```
example_of_long_variabel_name
```

   - Function names follow the same convention

```
my_function()
```

   - Class names use CamelCase:

```
class MyClass:  # we'll see this later in this course
```

A bit more on VS Code

Code example

In VS Code, create a file called `dice_function.py`, and copy and paste the following into it

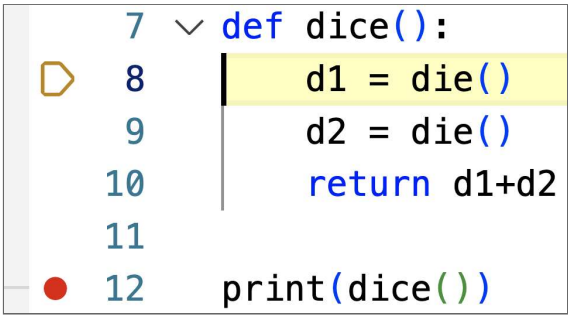
```
dice_function.py
import random

def die():
    x = random.random
    return 1+int(x*6)

def dice():
    d1 = die()
    d2 = die()
    return d1+d2

print(dice())
```

VS Code: Jump to definition

By right-clicking on a function name, you can jump to function's definition
You can jump back to the original position using `Go → Back`.
Try jumping to the definition of the `die` function and back again.
Note: get used to using keyboard shortcuts for this.

Debugging in VS Code: Stepping into

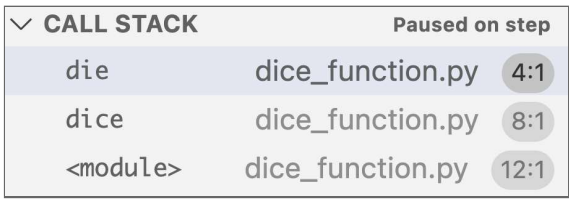
Place a breakpoint on the `print(dice())` line. Start the debugger.
Instead of using the `Step Over` button(), try pressing the `Step Into` (button). This should follow the function call up into the `dice()` function.



Press `Step Into` again to jump all the way up to the `die` function.

Debugging in VS Code: the Stacktrace

The left column of the debug window contains the *stack trace* (aka *call stack*)



Note that it shows which functions were called to reach your current position in the code.
You can click on the other lines to jump to where the function was called from.

VS Code debugging - Exercise

Press the continue button () to resume the program and see what it outputs.
What's going wrong? How would you fix this?

VS Code debugging - Exercise - solution

By following the execution of the program you will notice that this line produces a strange value:

```
x = random.random
```

This is because we are not calling `random` as a function. Instead, we should write:

```
x = random.random()
```

Speaker notes