Python Programming for Data Science

Week 37, Friday

- Debugging code
- Functions

Recap: loops: a popquiz

- What does a `while` loop do?
- What does a `for` loop do?
- What does `list(range(4))` do?
- What does this code do?:

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

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]
for i in range(len(number_list)):
    if i < 2:
        del number_list[i]

print(number_list)
```

Output:

```
[2, 4]
```

This is unexpected. What do we do?

Simple debugging: printing

```python
number_list = [1,2,3,4]
for i in range(len(number_list)):
    if i < 2:
        del number_list[i]

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]
for i in range(len(number_list)):
    if i < 2:
        print (i, number_list)    # Inserted
        del number_list[i]
```

```
output

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

Simple debugging: printing

```
number_list = [1,2,3,4]
for i in range(len(number_list)):
    if i < 2:
        print (i, number_list)    # Inserted
        del number_list[i]
```

```
                                                            output

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

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

Simple debugging: printing

```
number_list = [1,2,3,4]
for i in range(len(number_list)):
    if i < 2:
        print (i, number_list)    # Inserted
        del number_list[i]
```

```
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?

Simple debugging: printing

```
number_list = [1,2,3,4]
for i in range(len(number_list)):
    if i < 2:
        print (i, number_list)    # Inserted
        del number_list[i]
```

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

Debugging – commenting out code

This code does not run:

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

output

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

Debugging – commenting out code

This code does not run:

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

output

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

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

```python
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())
```

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'])
```

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.

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 PyCharm, it's:

`Code → Comment with Line Comment`

Debugging in PyCharm

Using a *debugger*, you can inspect values without inserting print() lines

There is a debugger built into PyCharm.

Debugging in PyCharm

Using a *debugger*, you can inspect values without inserting print() lines

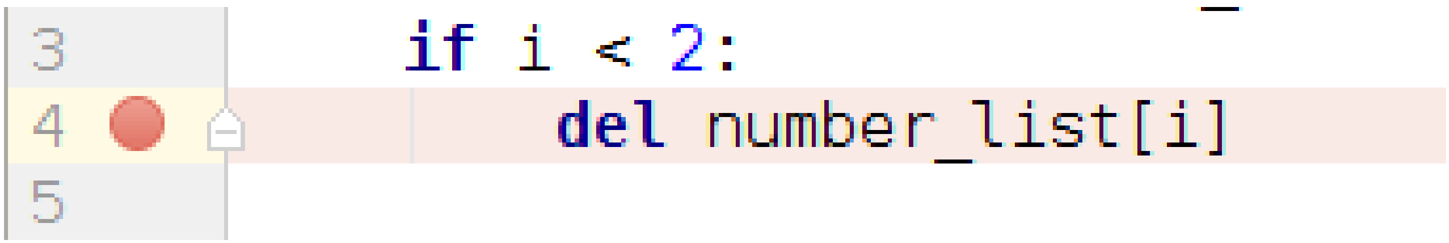
There is a debugger built into PyCharm.

The following slides are a walk-through exercise. **Please follow along!**

PyCharm debugger - breakpoints

You can tell program to temporarily pause its execution by inserting a *breakpoint*

In PyCharm, you can set a breakpoint by clicking in the left margin of the editor window



```
3       if i < 2:
4           del number_list[i]
5
```

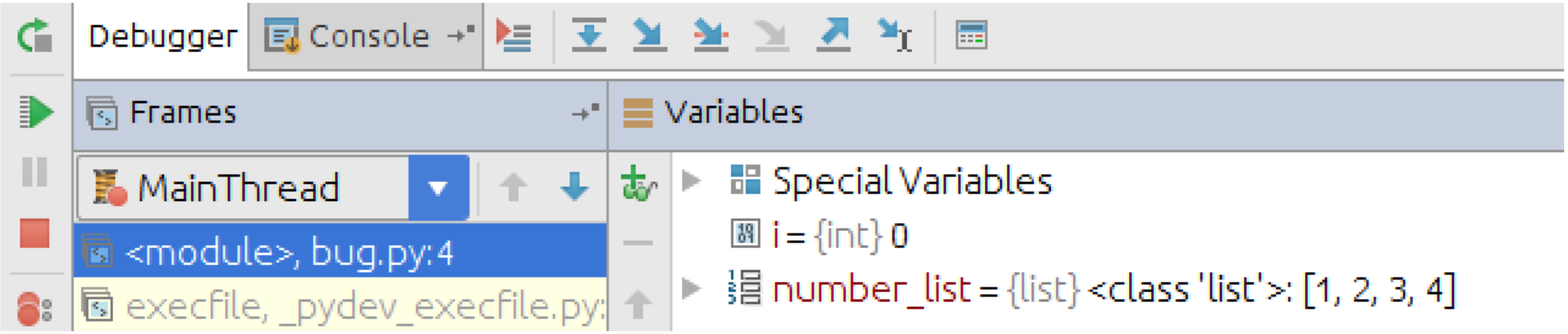
In the code from the last exercise, set a breakpoint in the same line as in this image. Then instead of the play button, we will now start the debugger...

PyCharm debugger - starting the debugger

Click on the bug button next to the green play button (🐞). The debug window should now appear, and it will run your code until the breakpoint.

PyCharm debugger - breakpoints (2)

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

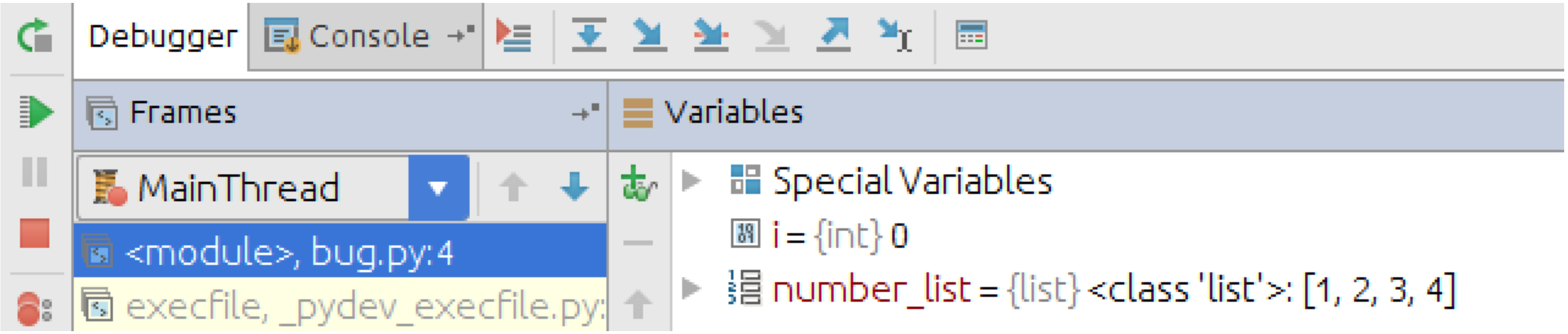


<module>, bug.py:4
execfile, _pydev_execfile.py:

Special Variables
i = {int} 0
number_list = {list} <class 'list'>: [1, 2, 3, 4]

PyCharm debugger - breakpoints (2)

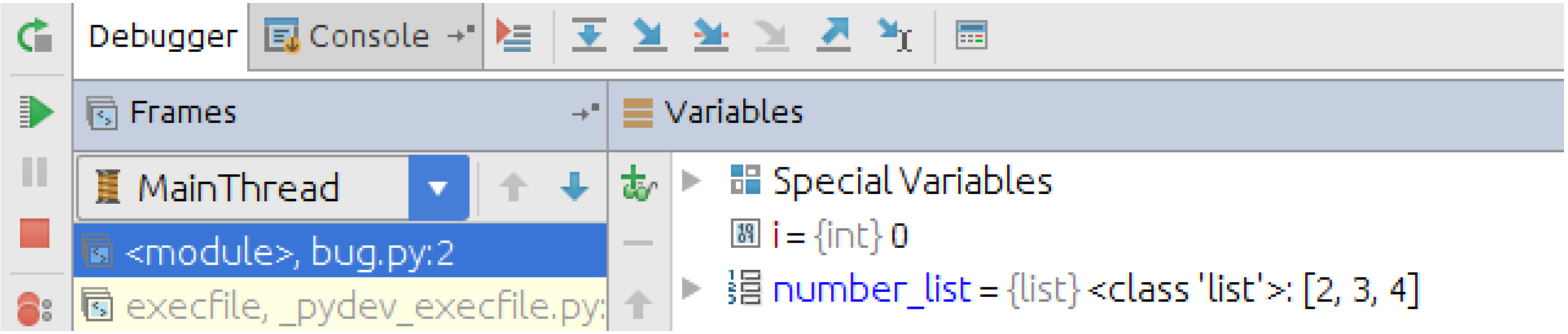
The program stops *before* executing the line at the breakpoint



```
<module>, bug.py:4
execfile, _pydev_execfile.py:
```

```
Special Variables
i = {int} 0
number_list = {list} <class 'list'>: [1, 2, 3, 4]
```

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



```
<module>, bug.py:2
execfile, _pydev_execfile.py:
```

```
Special Variables
i = {int} 0
number_list = {list} <class 'list'>: [2, 3, 4]
```

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

PyCharm 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 Pycharm and run the code (without debugger). Does it behave as it should?
3. Use the PyCharm debugger to find the error in the code

PyCharm 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.*

2. *Copy&paste the code into Pycharm and run the code (without debugger). Does it behave as it should?*

PyCharm 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 Pycharm and run the code (without debugger). Does it behave as it should?*

PyCharm 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 Pycharm and run the code (without debugger). Does it behave as it should?*

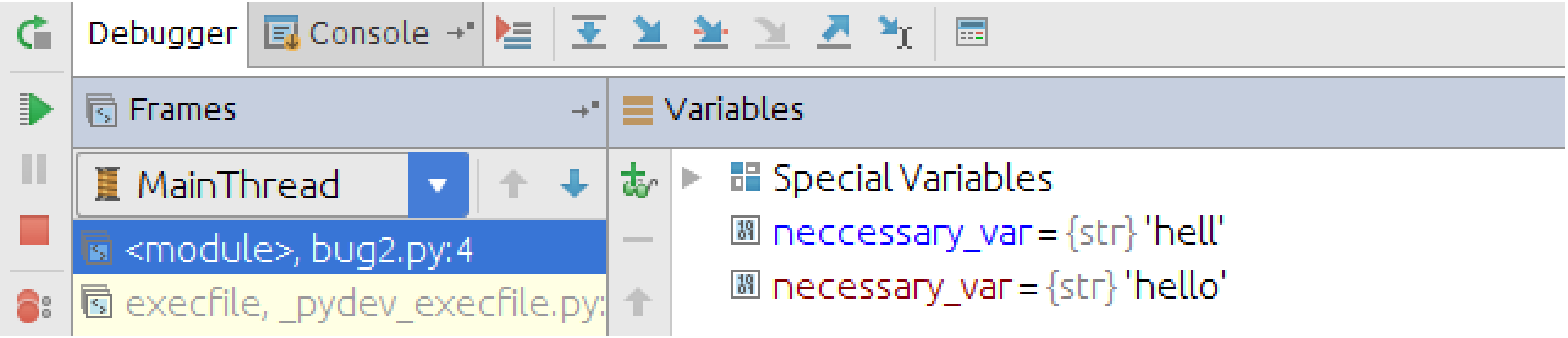
   No, it prints "hello"

PyCharm debugger - exercise - solution (2)

3. *Use the PyCharm debugger to find the error in the code*

PyCharm debugger - exercise - solution (2)

3. *Use the PyCharm debugger to find the error in the code*

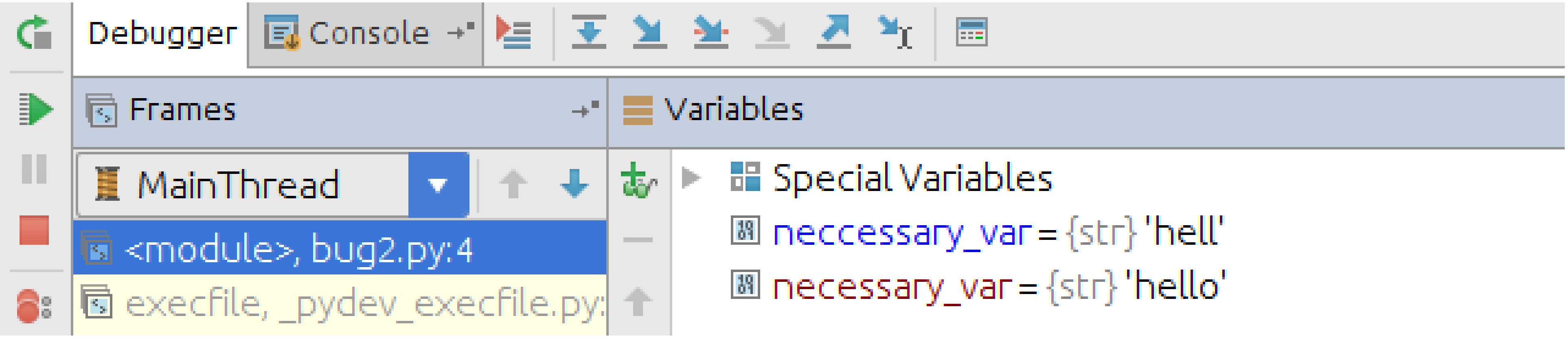


neccessary_var = {str} 'hell'

necessary_var = {str} 'hello'

PyCharm debugger - exercise - solution (2)

3. *Use the PyCharm debugger to find the error in the code*



```
neccessary_var = {str} 'hell'
necessary_var = {str} 'hello'
```

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.

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*.

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

```
output
5
```

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. Turn the die simulator from Monday's exercise 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 from Monday's exercise 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 from Monday's exercise 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

```python
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
```

Functions - return values

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

```python
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.

```python
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.

Functions with return values - Exercise - solution

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

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
```

Confusing behavior in interactive sessions

When we use our die function from the interactive prompt, it still *prints* the result:

```
>>> def die():                       # define function
...     x = random.random()
...     return 1+int(x*6)            # or simply random.randint(1,6)
...
>>> die()
5
```

Q: Why is this?

Confusing behavior in interactive sessions

When we use our die function from the interactive prompt, it still *prints* the result:

```
>>> def die():                      # define function
...     x = random.random()
...     return 1+int(x*6)           # or simply random.randint(1,6)
...
>>> die()
5
```

Q: Why is this?

A: Remember: the interactive session always automatically prints the return values of a function call.

Confusing behavior in interactive sessions

When we use our die function from the interactive prompt, it still *prints* the result:

```
>>> def die():                        # define function
...     x = random.random()
...     return 1+int(x*6)             # or simply random.randint(1,6)
...
>>> die()
5
```

Q: Why is this?

A: Remember: the interactive session always automatically prints the return values of a function call.

In a normal program (i.e. in PyCharm), it will not do this

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
```

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:

```python
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
```

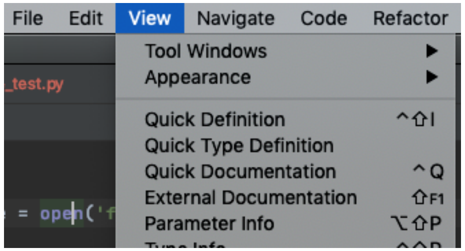
```
# Setting new_lines argument
print(repeat("hello", 3, True))
```

output

```
hello
hello
hello
```

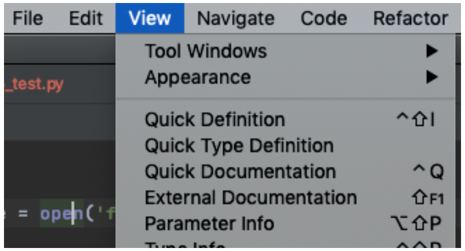
Functions - Documenting

In PyCharm, you can see a help message that explains what a function does



Functions - Documenting

In PyCharm, you can see a help message that explains what a function does

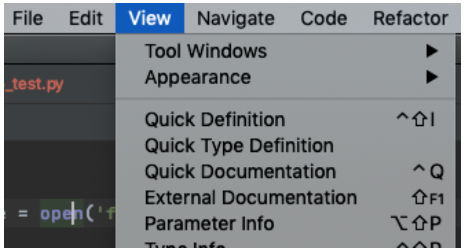


If you want others to be able to find help on the functions you write, specify a *doc-string* as the first line of your function definition.

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

Functions - Documenting

In PyCharm, you can see a help message that explains what a function does



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.

```python
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").

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...

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.p

```
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 nam
```

outpu

```
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.

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 argumen
```

output

```
tail
heads
```

Intermezzo...
...about variable names

Naming variables/functions/classes

Naming variables/functions/classes

1. Pick a meaningful name:
   `column_avg` is better than `x`

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

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
```

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()
     ```

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 PyCharm

Code example

In Pycharm, 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())
```

PyCharm: Jump to declaration

Using `Navigate` menu in PyCharm, you can jump around in your code.

PyCharm: Jump to declaration

Using `Navigate` menu in PyCharm, you can jump around in your code.

Handy feature: `Navigate` → `Declaration or Usages`.

If the cursor is currently on a function (or module or class) name, it will jump to the corresponding definition.

PyCharm: Jump to declaration

Using `Navigate` menu in PyCharm, you can jump around in your code.
Handy feature: `Navigate` → `Declaration or Usages`.
If the cursor is currently on a function (or module or class) name, it will jump to the corresponding definition.

You can jump back to the original position using `Navigate` → `Back`.

PyCharm: Jump to declaration

Using `Navigate` menu in PyCharm, you can jump around in your code.
Handy feature: `Navigate` → `Declaration or Usages`.
If the cursor is currently on a function (or module or class) name, it will jump to the corresponding definition.

You can jump back to the original position using `Navigate` → `Back`.

Try jumping to the definition of the `die` function and back again.

PyCharm: Jump to declaration

Using `Navigate` menu in PyCharm, you can jump around in your code.
Handy feature: `Navigate` → `Declaration or Usages`.
If the cursor is currently on a function (or module or class) name, it will jump to the corresponding definition.

You can jump back to the original position using `Navigate` → `Back`.

Try jumping to the definition of the `die` function and back again.

Note: the keyboard shortcuts might collide with those used by the window manager. They can be changed.

Debugging in PyCharm: Stepping into (2)

Place a breakpoint on the `print(dice())` line

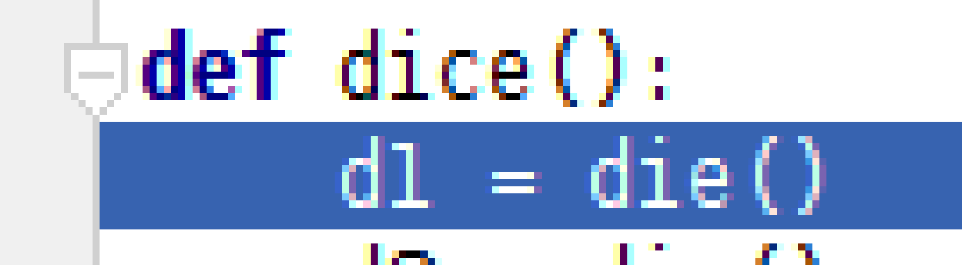
Start the debugger

Debugging in PyCharm: Stepping into (2)

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.



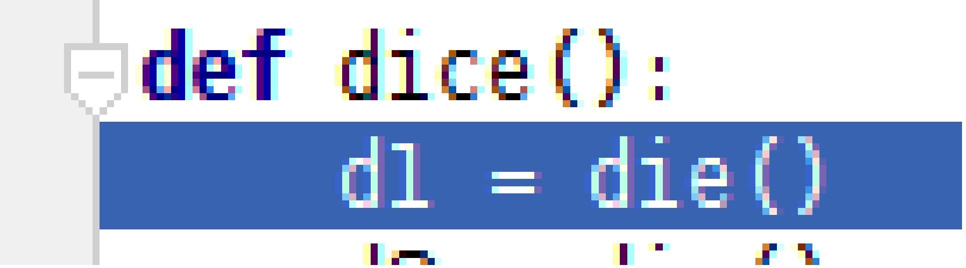
```
def dice():
    d1 = die()
```

Debugging in PyCharm: Stepping into (2)

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.

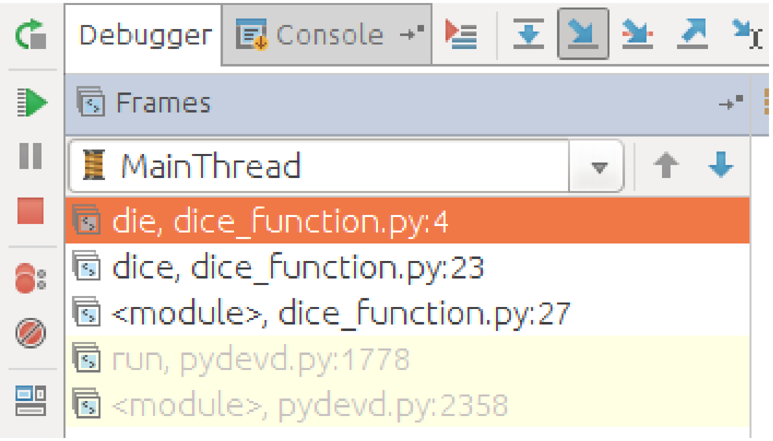


```
def dice():
    d1 = die()
```

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

Debugging in PyCharm: the Stacktrace

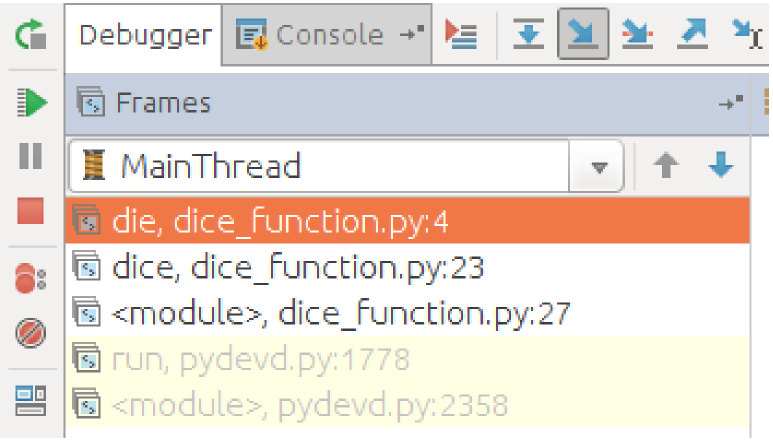
The left column of the debug window contains the *stack trace*



```
die, dice_function.py:4
dice, dice_function.py:23
<module>, dice_function.py:27
run, pydevd.py:1778
<module>, pydevd.py:2358
```

Debugging in PyCharm: the Stacktrace

The left column of the debug window contains the *stack trace*

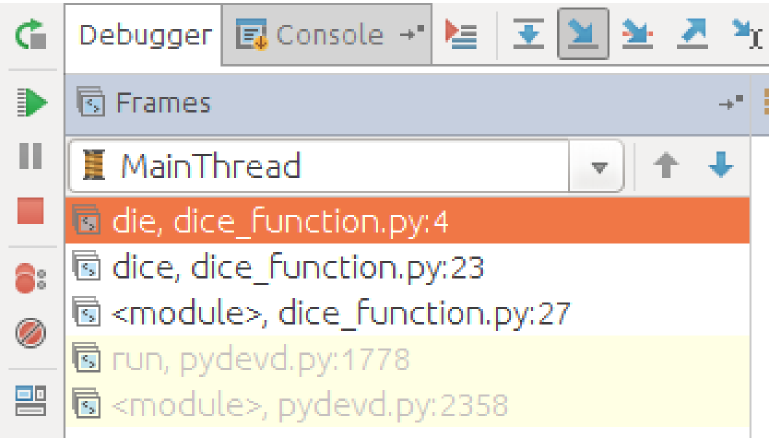


```
die, dice_function.py:4
dice, dice_function.py:23
<module>, dice_function.py:27
run, pydevd.py:1778
<module>, pydevd.py:2358
```

Note that it shows which functions were called to reach your current position

Debugging in PyCharm: the Stacktrace

The left column of the debug window contains the *stack trace*



die, dice_function.py:4
dice, dice_function.py:23
<module>, dice_function.py:27
run, pydevd.py:1778
<module>, pydevd.py:2358

Note that it shows which functions were called to reach your current position

You can click on the other lines to jump to where the function was called from.

PyCharm debugging - Exercise

Press the continue button (▶) to resume the program and see what it outputs.

What's going wrong? How would you fix this?

PyCharm debugging - Exercise - solution

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

```
x = random.random
```

`x = {builtin function...}`

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

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