Introduction to Application Development in Python

Lecture 2

Fatih Han Çağlayan

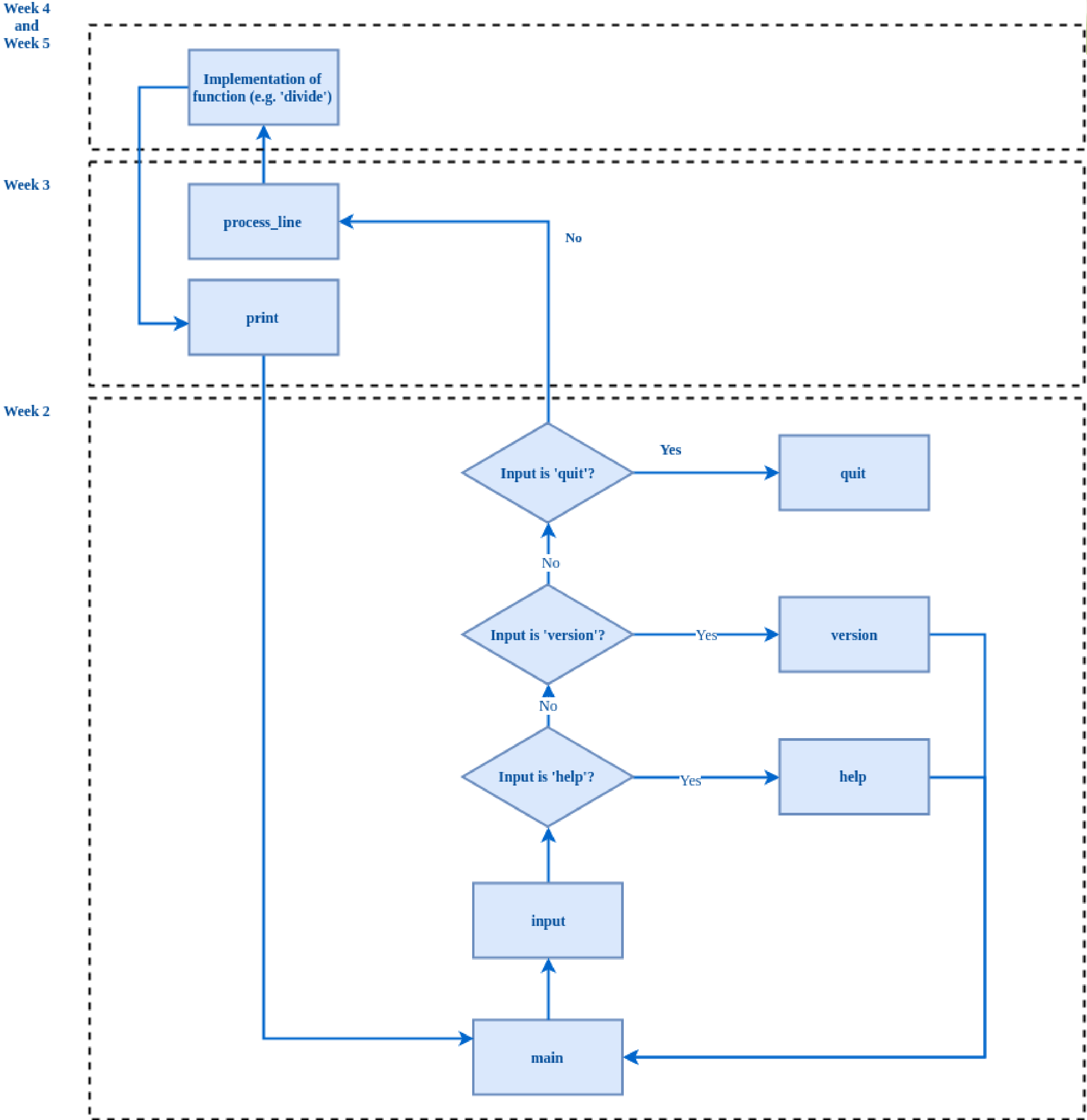
Lecture topics

- Overview and structure of final assignment
- What to expect when learning to program?
- Install the python environment and IDE
  - Latest Python 3
  - IDE Preference: PyCharm
- Modules
- Integrated Development Environment (IDE)
- Functions
- Conditions
- Loops and main loop of program
- Output and input

- Warm-up assignment with debugger
- Implement main loop, version command, help command and quit command

Functionality of calculator

- Addition
- Subtraction
- Multiplication
- Division
- Power
- Square root
- Greatest common divisor
- Least common multiple
- Modulo
- Etc…

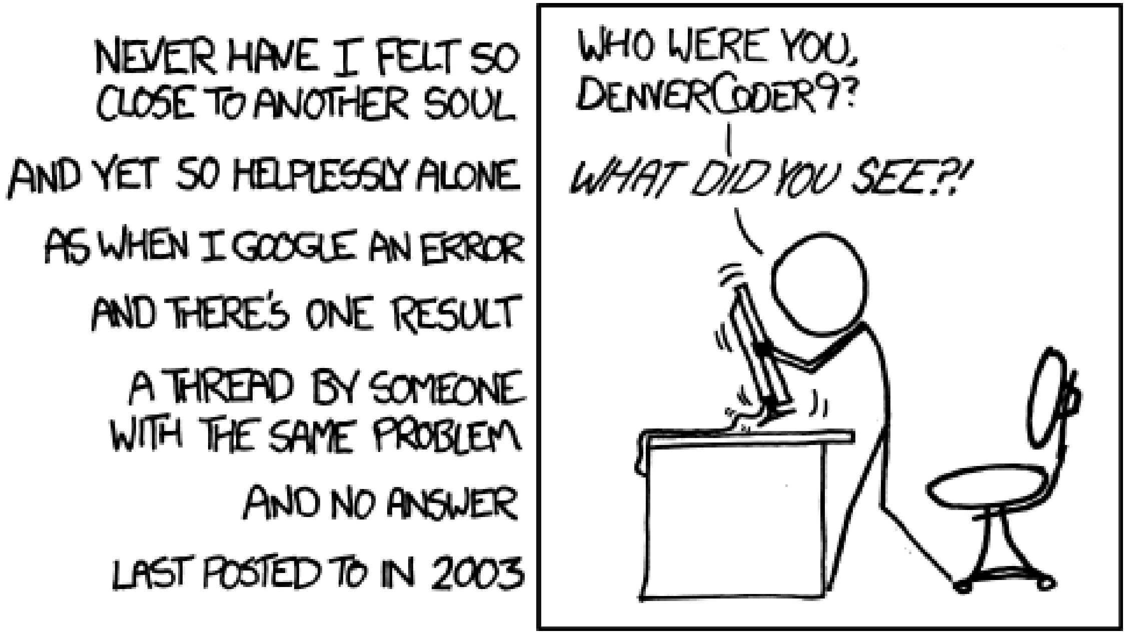


Existing code

- In the **code/** directory, you can find a starting point, i.e. existing code, for the implementation of your calculator
  - **MAKE USE OF THE EXISTING CODE AND FUNCTIONS**
- The code is divided into multiple modules:
  - **arith_tools**
  - **arith**
  - **functions**
  - **main**
- **debugging_exercises.py** is a separate program that will only be used for assignment 1

What to expect when learning to program?

- Be patient
- Learning a language takes time
- Understanding how to solve problems also takes time
- It is normal to be frustrated
  - It is the moment to be patient and ask questions
- Questions? => Teacher /student-assistant


NEVER HAVE I FELT SO CLOSE TO ANOTHER SOUL
AND YET SO HELPLESSLY ALONE
AS WHEN I GOOGLE AN ERROR
AND THERE'S ONE RESULT
A THREAD BY SOMEONE WITH THE SAME PROBLEM
AND NO ANSWER
LAST POSTED TO IN 2003
WHO WERE YOU, DENVERCODER9?
WHAT DID YOU SEE?!

Python environment and IDE installation

- Download the latest Python 3 environment from https://www.python.org/downloads/windows/
- Download PyCharm from: https://www.jetbrains.com/pycharm-edu/
- Questions? => Teacher /student-assistant

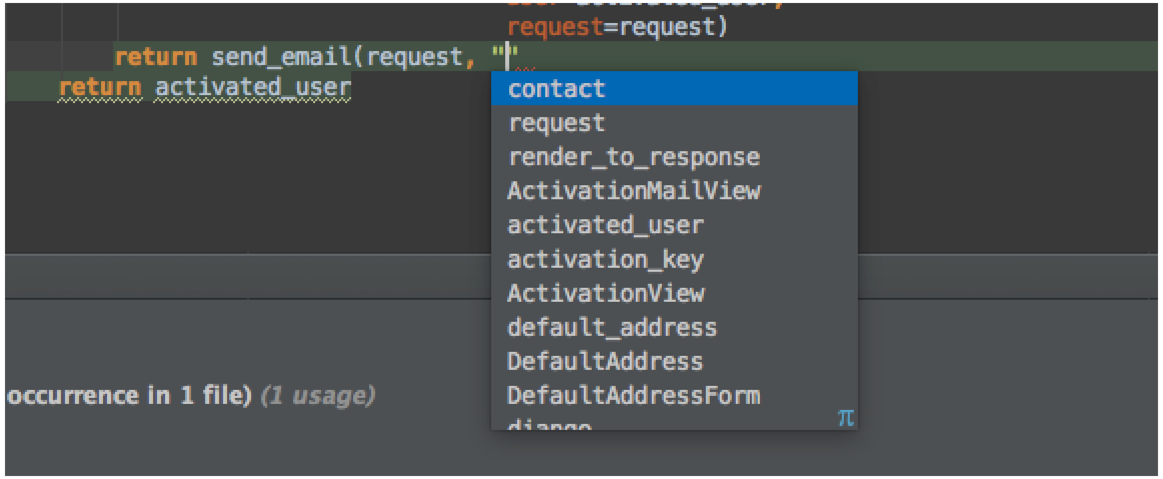
Integrated Development Environment (IDE)

An IDE combines multiple tools into one

- Text editor or actually source code editor
- Debugger
- Linter
- Compiler or interpreter
- Version control
- Etc.

Integrated Development Environment
Source code editor

- Auto completion
- Linting, a hint in case of an error
- Colored keywords
- Other visual aids

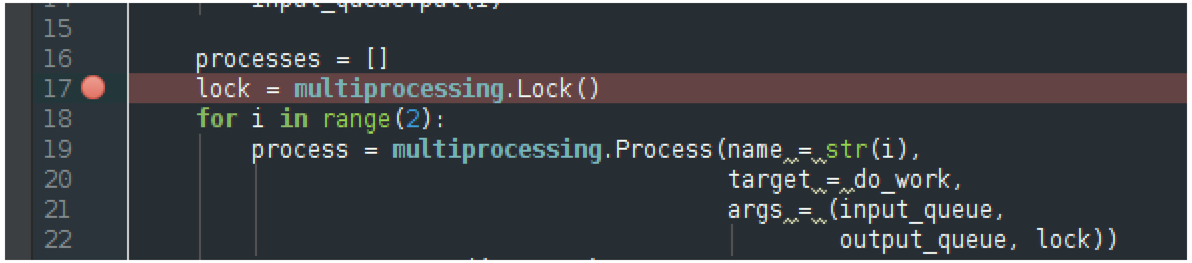


```
                                   request=request)
        return send_email(request, ""
    return activated_user
```

occurrence in 1 file) *(1 usage)*

Integrated Development Environment Debugger

- **Debug** software
- **Step** through code
- Set **breakpoints**
    - Points where the debugger pauses
- Peek in **memory**
    - See variables and their states
- Watch the function call order
    - The **call stack**

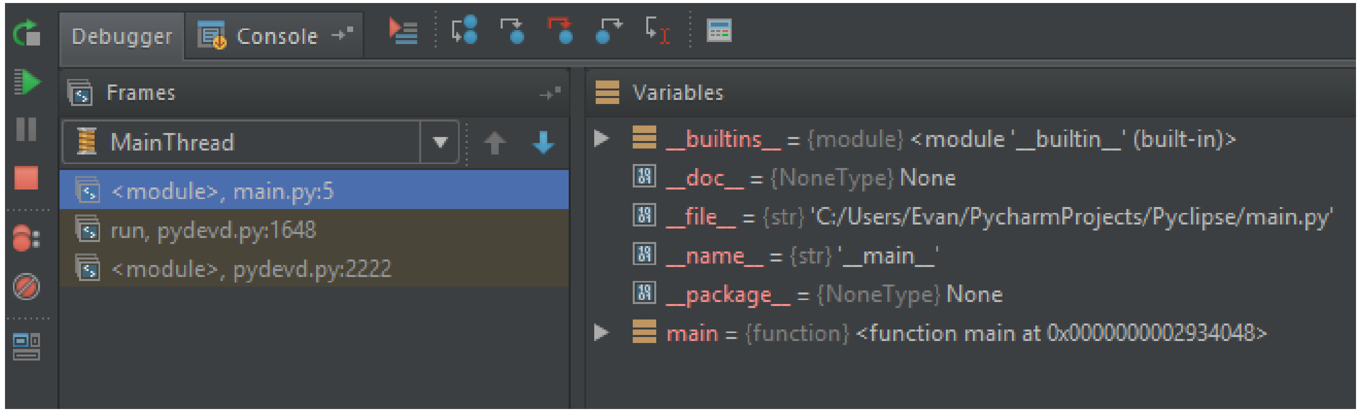


```
    input_queue.put(i)

    processes = []
    lock = multiprocessing.Lock()
    for i in range(2):
        process = multiprocessing.Process(name = str(i),
                                          target = do_work,
                                          args = (input_queue,
                                                  output_queue, lock))
```

Integrated Development Environment Debugger controls

- **Start:**
  - Start debugging the application
  - i.e. run your code
- **Step over:**
  - Move to the next line of code
- **Step in/out:**
  - Step into the function at current line
  - Step out of the current function
- **Restart:**
  - Restart debugging the application
  - i.e. re-run your code
- **Stop:**
  - Stop debugging the application
  - i.e. stop your code



```
<module>, main.py:5
run, pydevd.py:1648
<module>, pydevd.py:2222
```

```
__builtins__ = {module} <module '__builtin__' (built-in)>
__doc__ = {NoneType} None
__file__ = {str} 'C:/Users/Evan/PycharmProjects/Pyclipse/main.py'
__name__ = {str} '__main__'
__package__ = {NoneType} None
main = {function} <function main at 0x0000000002934048>
```

Integrated Development Environment Compiler & Interpreter

- You rely on a compiler to make your code executable
- You rely on an interpreter to execute your code
- For Python, we use the Python interpreter
- Visual Studio Code can make use of the Python interpreter to run your code
- You will require the Python extension for MSVSC

Integrated Development Environment Version control

- We strongly suggest you use **version control** for your code
- Version control keeps track of the changes to your code and thus acts as a backup for versions of your codebase
- A well-known version control system is **Git**

- **A very good interactive introduction to git:** https://www.katacoda.com/courses/git

Modules in python

- In Python we can import functionality from a **module**

pre-implemented module:

```
import math
print(math.sqrt(25))
```

or self created

testmodule.py

```
def test_function():
	print('test’)
```

main.py

```
import testmodule
testmodule.test_function()
```

Modules in python

- In Python we can import functionality from a **module**

pre-implemented module:

```
import math
print(math.sqrt(25))
```

or self created

testmodule.py

```
def test_function():
        print('test’)
```

main.py

```
import testmodule
testmodule.test_function()
```

- Remember to place the module name with a dot (.) in front of the function.

Functions – What & Why?

- Calculate area of circles **WITHOUT** functions

```
radius_one = 5
radius_two = 11
radius_three = 14

circle_one_area = radius_one * radius_one * 3.14
circle_two_area = radius_two * radius_two * 3.14
circle_three_area = radius_three * radius_three * 3.14
```

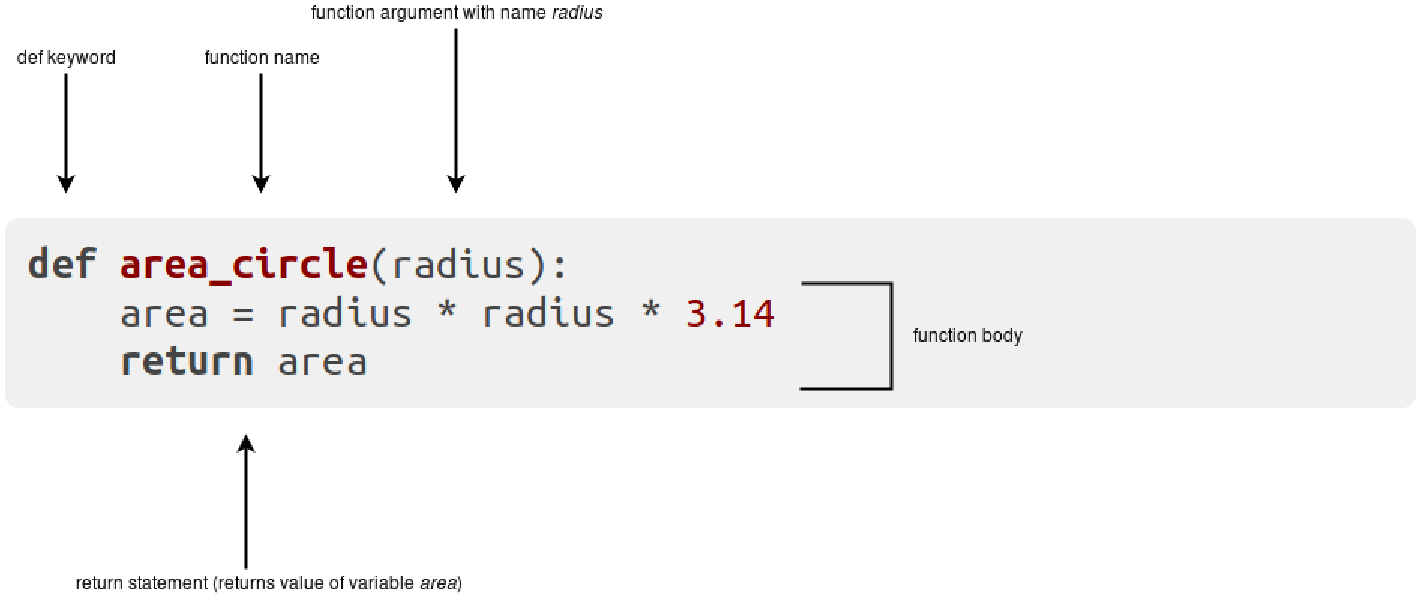
- Calculate area of circles **WITH** functions

```
def area_circle(radius):
        return radius * radius * 3.14

circle_one_area = area_circle(5)
circle_two_area = area_circle(11)
circle_three_area = area_circle(14)
```

Functions – What & Why?

- Functions are units that perform a specific **task**
- Functions can be used in your program wherever such a task should be performed
- Functions **group** instructions such that the instructions combined perform a task
- Executing a function is typically called a function **call**
- The instructions that make up the function are typically refer to as the **body** of a function
- A function can have multiple **parameters** (comma-separated)
- A function can **return** values



```python
def area_circle(radius):
    area = radius * radius * 3.14
    return area
```

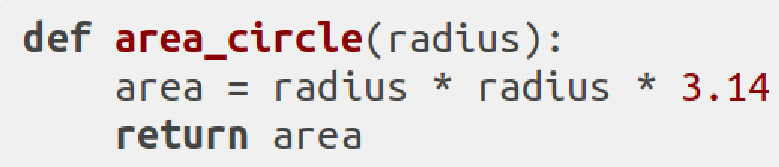
function argument with name *radius*
def keyword
function name
function body
return statement (returns value of variable *area*)

Functions – How to use?

- Assume that we have created the function area_circle(radius):

  area = area_circle(5)

  Passes 5 to function area_circle and assigns return value to area



```
def area_circle(radius):
    area = radius * radius * 3.14
    return area
```

- A function with multiple parameters:

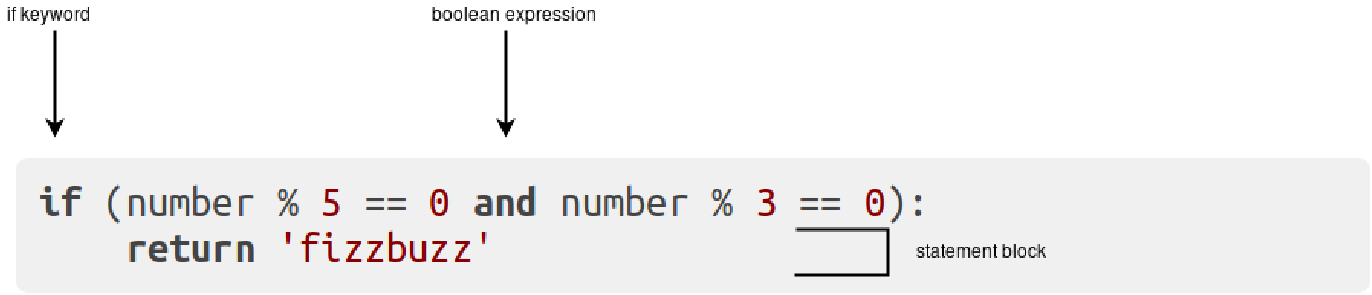
```
def area_square(width, height):
        return width * height
```

```
area = area_square(5, 10)
```

  Passes 5 and 10 to function area_square and assigns return value to area

Conditions

Conditional branching allows to execute different instructions depending on whether or not some condition, a boolean expression is **True.**



```
if (number % 5 == 0 and number % 3 == 0):
    return 'fizzbuzz'
```

```
a = 15
b = 14

def fizzbuzz(number):
    if (number % 5 == 0 and number % 3 == 0):
        return 'fizzbuzz'
    elif (number % 5 == 0):
        return 'fizz'
    elif (number % 3 == 0):
        return 'buzz'
    else:
        return str(number)

c = fizzbuzz(a)
d = fizzbuzz(b)
```

What are the values of **c** and **d**?

Main loop

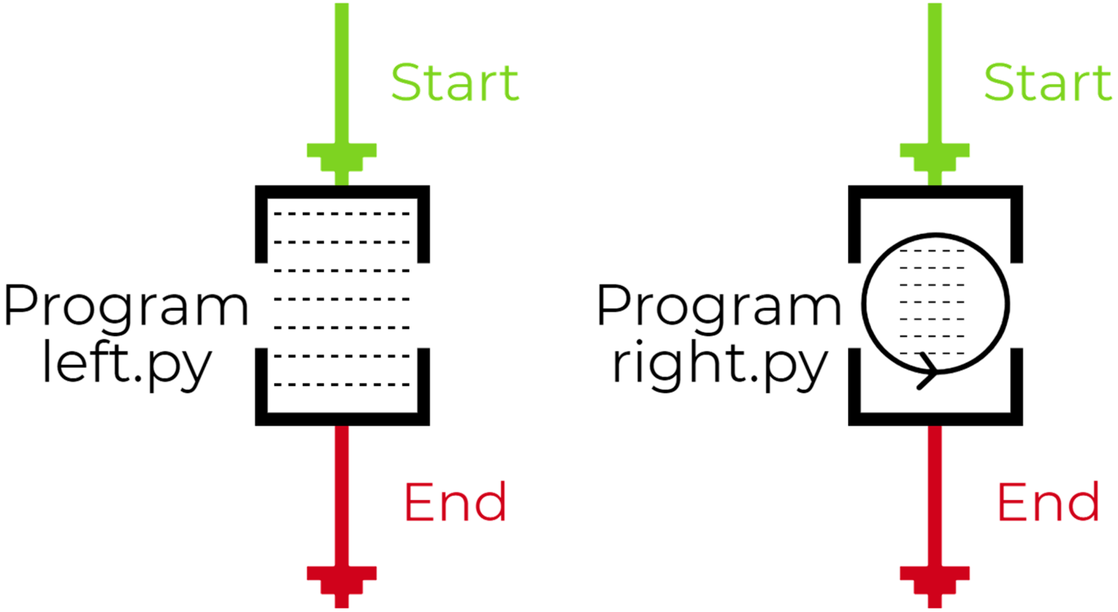
- **left.py**

```
do_something()
do_something_else()
do_something()
do_something_else()
```

- **right.py**

```
exit = False
while (not exit):
        do_something()
        do_something_else()
        do_something()
        do_something_else()
        exit = should_exit()
```

- The left example simply performs some tasks sequentially
- The right example runs until it should stop (denoted by the exit variable).



Output

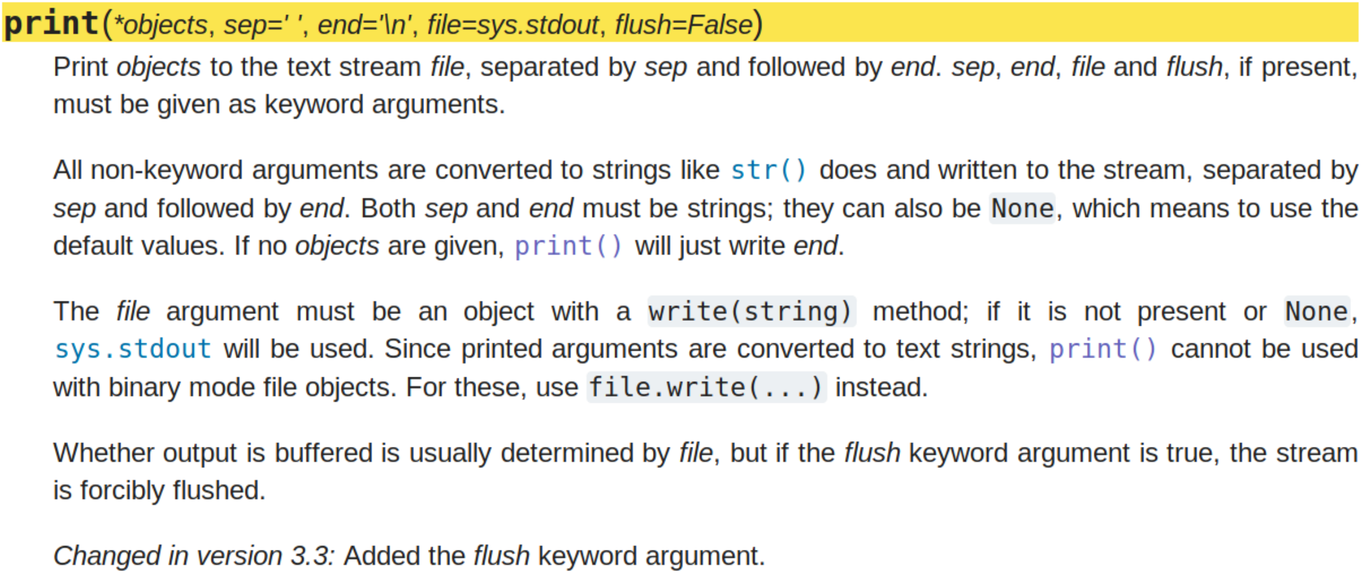
- Python provides a way to write text to the **standard output** stream
  - What is standard output?
- Calculator will be a **command-line** application
  - Command-line applications make use of **text-based** input and output
  - Under Windows, we will run our application in **command-prompt**
  - Under Linux and OSX, we will use a terminal (e.g. **bash**)
- To write to the standard output stream in Python, we make use of the **print** function.

Output – print function

From the official Python website: https://docs.python.org/3/library/functions.html#print

Documentation tells us that **print**:

- Takes 5 parameters, for now, we will only use the first one
- Notice that argument **file** is set to **sys.stdout** (the standard output stream!)
- Prints **objects** to the standard output stream



print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argument is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument.

Output – using the print function

- We can print objects, for instance, one string object:

```
print('foo’)
```

- two string objects:

```
print('foo', 'bar’)
```

- two string objects:

```
print('foo' + '1', 'bar’)
```

The outputs that we can expect in our console are respectively:

```
foo
foo bar
foo1 bar
```

Input

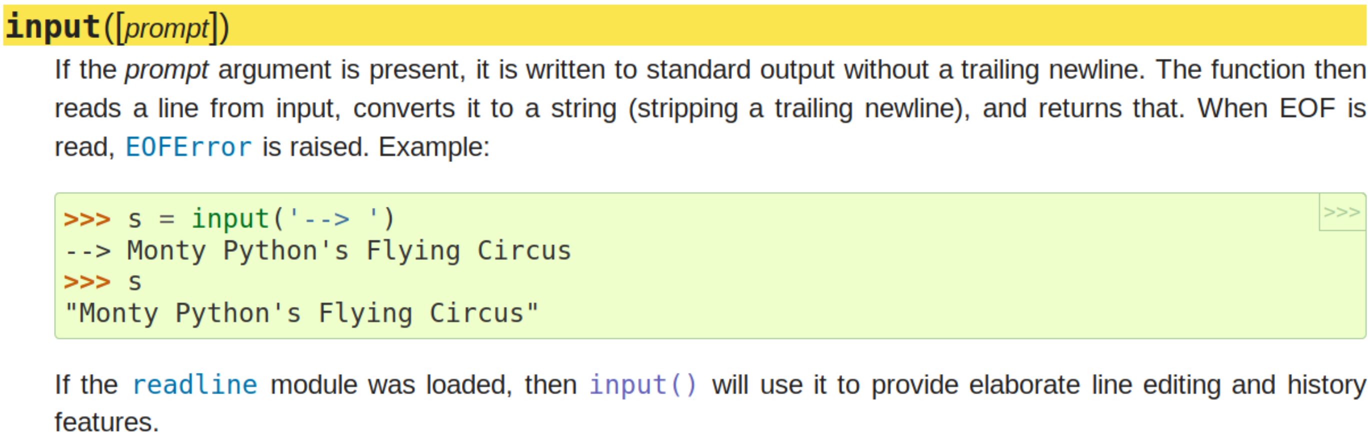
- Python provides a way to read text from the standard input stream.
  - What is standard input?
- We can instruct our application to **read** text input from the console
  - The application will wait until text followed by a **new line** is entered
  - Thus input text by writing and hitting the **enter** key
- To read from the standard input stream in Python, we make use of the **input** function.

Input function

From the official Python website: https://docs.python.org/3/library/functions.html#input

Documentation tells us that input:

- Takes an optional argument prompt that prints the message prompt to the standard output
- Returns the text input entered into the console



input([*prompt*])

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Input – using the function

- We can print objects, for instance, one string object:

```
age = input('your age is? ')
print('your age is: ', age)
```

- The outputs that we can expect in our console is:

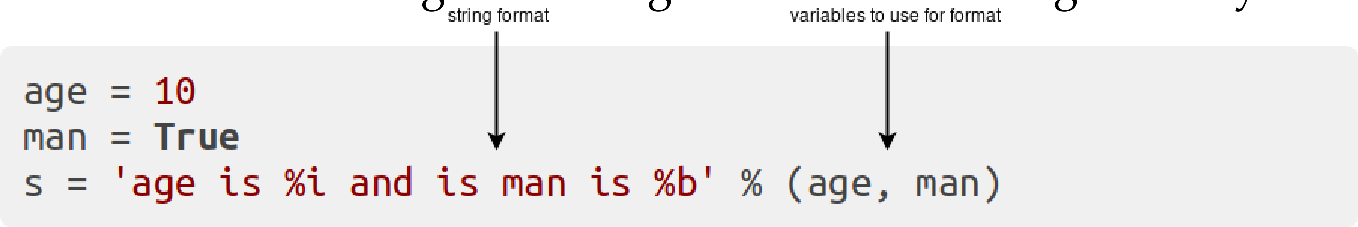
```
your age is? 42
your age is: 42
```

String formatters

- Example: age is supplied as a second object to the print function.

  print('your age is: ', age)

- We can also use string formatting to format our string in a way to include the age variable.



```
age = 10
man = True
s = 'age is %i and is man is %b' % (age, man)
```

The output that we can expect when printing s is:

age is 10 and is man is True

- The string format specifies the type of the variables that should be formatted.
  In this example, we see two such variables referred to be %i (integer) and %b (boolean).

Strir

- Exampl

print('yo

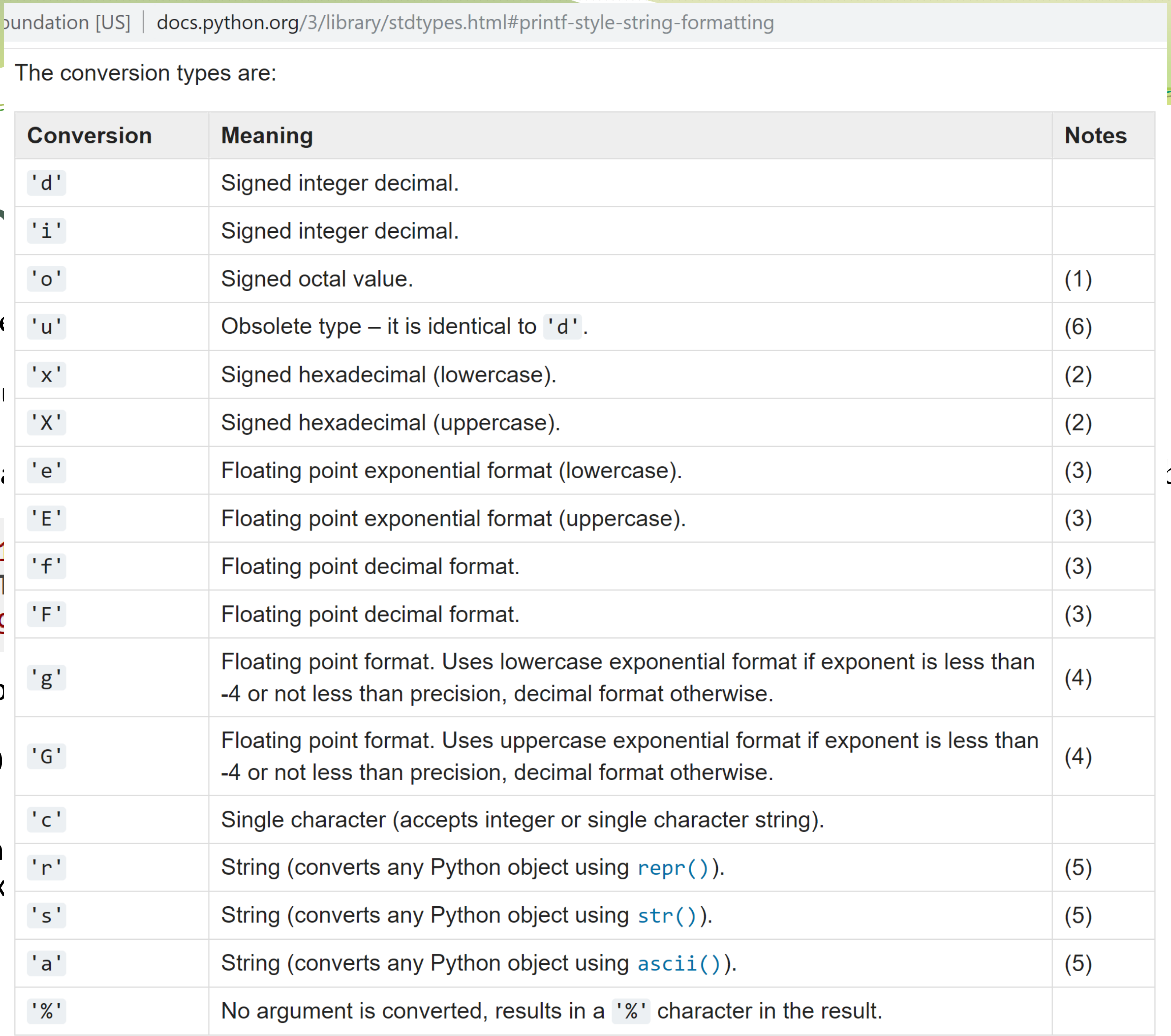
- We can a ble.

```
age = 1
man = T
s = 'ag
```

The outp

age is 10

- The strin
In this ex



undation [US] | docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

The conversion types are:

Conversion	Meaning	Notes
`'d'`	Signed integer decimal.	
`'i'`	Signed integer decimal.	
`'o'`	Signed octal value.	(1)
`'u'`	Obsolete type – it is identical to `'d'`.	(6)
`'x'`	Signed hexadecimal (lowercase).	(2)
`'X'`	Signed hexadecimal (uppercase).	(2)
`'e'`	Floating point exponential format (lowercase).	(3)
`'E'`	Floating point exponential format (uppercase).	(3)
`'f'`	Floating point decimal format.	(3)
`'F'`	Floating point decimal format.	(3)
`'g'`	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'G'`	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'c'`	Single character (accepts integer or single character string).	
`'r'`	String (converts any Python object using `repr()`).	(5)
`'s'`	String (converts any Python object using `str()`).	(5)
`'a'`	String (converts any Python object using `ascii()`).	(5)
`'%'`	No argument is converted, results in a `'%'` character in the result.	

Assignment

- Please see schedule on course website:
  - Warm-up assignment with debugger
  - Implement main loop, version command, help command and quit command

- Deliver assignment **both** in person and automated testing