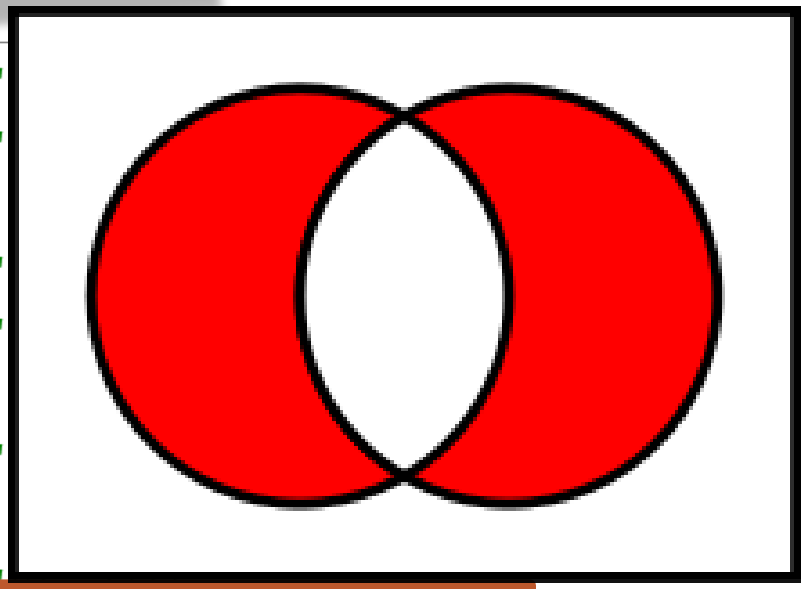
INTRODUCTION TO PYTHON

LECTURE 3: ABSTRACTION AND FUNCTIONS

Asem Elshimi

$$A \triangle B \equiv (A \setminus B) \cup (B \setminus A)$$

```
s1 = set('Hello')      # => {'
s2 = set('world')      # => {'
# difference
s1 - s2                # => {'
s2 - s1                # => {'
# union
s1 | s2              # => {'H'
# intersection
s1 & s2                # => {'
# symmetric difference
s1 ^ s2                # => {'H', 'e', 'w', 'r', 'd'}
```



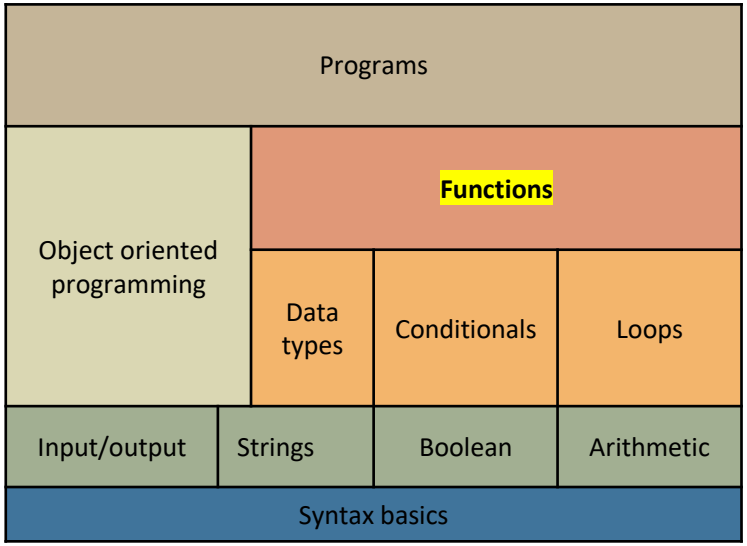
Where are we?

So far:

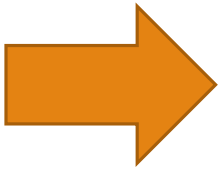
- Conditionals.
- Loops.
- Data structures:
  - Sets, Dicts, Strings, Tuples, lists

Today:

- Abstraction and decomposition.
- Functions()



Abstraction



Abstraction: hides low level details.

Source: https://www.ifixit.com/Teardown/Changhong+UD42YC5500UA+4K+42-Inch+LED+LCD+TV+Teardown/64167

Decomposition

Decomposing a big process into smaller processes.

Functions and abstraction

A **function** is a block of code which only runs when it is called.

What do they do?

- Reuse code more than once
- Hide computations in local variables
- Improve readability via abstraction

Q: *Do functions affect the logic of the program?*

Let's talk syntax

The way you did it in C

```c
#include <stdio.h>
/* function definition*/
bool IsEven(int num); {
/* local variable declaration */
int result;
if ((num%2)==0)
printf( "Input is even");
result = True;
else
printf( "Input is odd");
result = False;
return result;
}


int main () {
/* global variable definition */
int a = 5;
/* calling a function to find if even */
ret = IsEven(a);
return 0;
}
```

Python

```python
def Is_even(i):
        """
         input i, a positive integer
        returns true if number is even
        """
        print("inside the function")
        return i%2==0


Is_even(5)
```

Let's talk syntax

Keyword Name Argument/s

```
def Is_even( i ):
    """
    input i, a positive integer
    returns true if number is even
    """
    print("inside the function")
    return i%2==0
```

Specification (docstring)

Function body

```
Is_even(5)
```

Calling the function in the code using its name and passing values for its parameters.

Courtesy of Dr. Ana Bell. MIT 6.0001 Introduction to Computer Science and Programming in Python, Fall 2016

Keyword and positional arguments

positional keyword

```
def sum3(x, y, z=0):
    """returns a sum of three numbers
    But doesnt complain if it only gets two"""
    return x+y+z

sumxy=sum3(4, 5)

sumxyz=sum3(4, 5, 10)
sumk1=sum3(x=4, y=5, z=10)
sumk2=sum3(y=5, z=10, x=4)
```

Equivalent!

Enhancing readability.

Keyword argument for `int()`

```
int('100')              # => 100
int('100', 16)          # => 256
int('100', base=8)      # => 64
```

None

```
def do_nothing():
    #empty function
    x=1


print(do_nothing())     # => None
```

return

```
def divide( a, b ):
    #returns dividend
    #and remainder
    div=A//b
    rem=A%b
    return div
    return rem

divide(5,4)
```

```
def divide( a, b ):
    #returns dividend
    #and remainder
    div=A//b
    rem=A%b
    return (div,rem)


(w,z)=divide(5,4)
```

No **return** : returns None

Python interpreter wont see anything after first **return**.

return statement only takes in one "object."

Branching and **return**

```
def absoluteValue(x):
    if x<0:
        return -x
    elif x >=0:
        return x
```

```
def absoluteValue(x):
    retValue= 0
    if x<0:
        retValue= -x
    elif x >=0:
        retValue= x
    return retValue
```

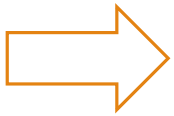
Better to make sure every path has a return.

Local & global variables

```
x = 5
z=3

def foo():
    y = 6
    print("local y:", y)
    x = 10
    print("local x:", x)
    print("local z:", z)


foo()
print("global x:", x)
print("global z:", z)
```



```
local y: 6
local x: 10
local z: 3
global x: 5
global z: 3
```

Advice: Stick to local variables

Code visualization

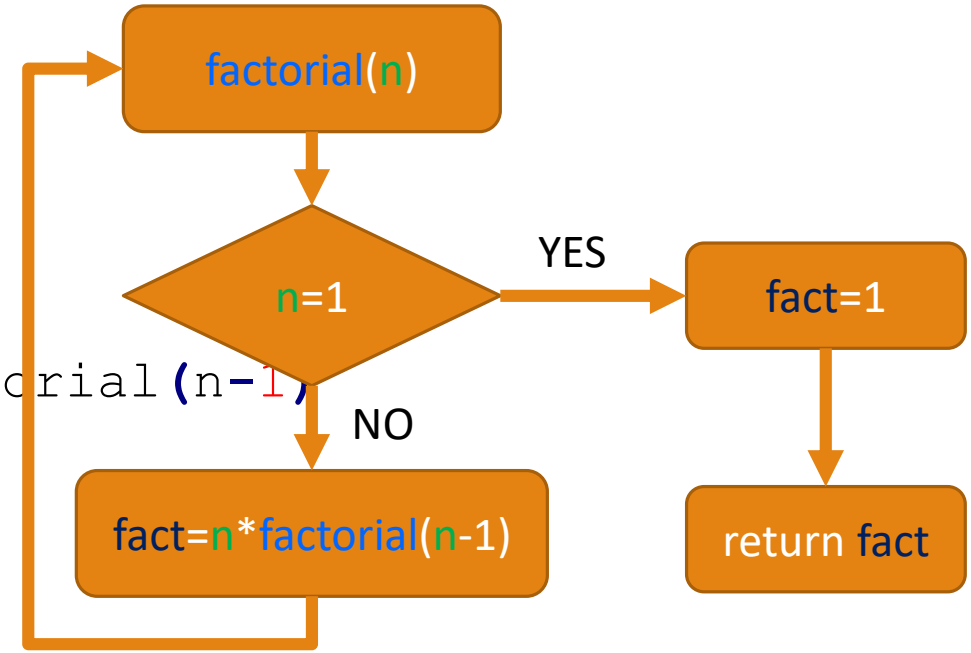
```
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print a,b,x,y

a,b,x,y = 1,15,3,4
foo(17,4)
print a,b,x,y
```

http://www.pythontutor.com/

Recursive functions: factorial

```
def factorial(n):
    """Function to return the factorial
    of a number using recursion, assumes positive
    nonzero input"""
    if n == 1:
        return n
    else:
        return n*factorial(n-1)
```



Sum of a list

```
def sum(nums):
    if nums:
        retval = 0
        for i in nums:
            retval += i
        return retval

sum([1, 2, 3])      # => 6
sum([])             # => None
sum(None)           # => None
```

Proper way to check if nums is non-empty and not None

On the else path, we don't return anything

Variable number of arguments

```
def func(*args):
    for x in args:
        print(x)
```

Variable number of arguments are packed into a tuple

```
func(0)          # => 0
func(1, 2, 3)    # => 1
                 #    2
                 #    3
```

Lambda functions

Func name args expression

```
x = lambda a : a + 10
print(x(5))                    #=>15
```

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))              #=>13
```

Anonymous functions

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))                    #=>22


mytripler = myfunc(3)

print(mytripler(11))                    #=>33
```

Suggested readings

Currying (Partial argument functions.)

Generators (**`yield`** instead of **`return`**.)

Getting the docstring

```
help(Is_even)
#==>Help on function Is_even in module __main__:
#==>Is_even(i)
#==>     input i, a positive integer returns true
#==> if number is even
```

main() routine

```
print("Hello")
#some code
def main():
    #main routine
    print("python main function")

main() #Not the best way to call main
print("__name__ value: ", __name__)
#==> __name__ value:  __main__
if __name__ == '__main__':
    main()
```

__name__ is an implicit variable value (main or module)

Importing libraries

```
from my_file1.py import *

#my_file1 includes few functions:
#sum3 and Is_even etc

sum3(4,5)
Is_even(5)
```

Notice different syntax for **`import`**.

Libraries that we are going to cover

Numpy

Pandas

MatplotLib

Approach to complex programs

5 MIN BREAK

Programming expectations and reality

Approach to complex programs

- Think big picture first.
- Decompose the program into modules:
  - Each can be debugged separately.
  - Document input/output constraints and behavior.
- Test modules
- Integrate modules into main program.
- Test main program.

Module development

1. High level.
2. Start small, and make small incremental changes.
3. Use placeholders and temporary variables for parts under development.
4. Consolidate.

```
def distance(x1,y1,x2,y2):
    return 0.0
```

Clear documentation:

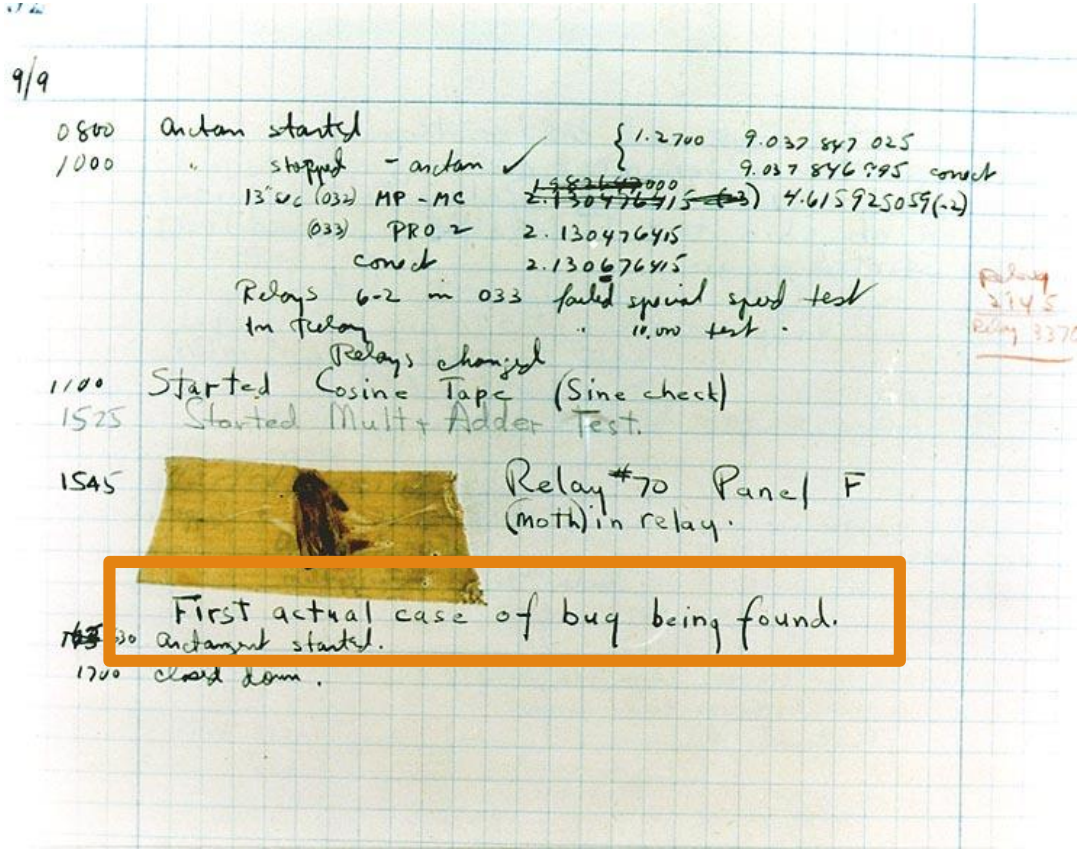
- Comments.
- Variable and function names.

Testing

1. Syntax test: no angry text when running code.
2. Unit testing: each module separately
3. Integration testing: This is the last step, not the first.

- Approaches for testing:
  - Intuition testing:
    - Construct intuitive edge cases.
    - Or, use random data for inputs (could be slow.)
  - Black box testing.
  - Glass box testing.

Debugging



By Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. - U.S. Naval Historical Center Online Library Photograph NH 96566-KN The above link is no longer valid on 13.04.2017, the image available here., Public Domain, https://commons.wikimedia.org/w/index.php?curid=165211

Where things go wrong?

Syntactic errors:

- common and easily caught.
- Interpreter angry text

Different meaning from what the programmer intended.

- Program runs just fine, but output is incorrect

Static semantic errors

- Causes unpredictable behavior

Debugging tools

Built in to Spyder.

Python tutor.

`print` statement.

Use your brain, be systematic in your hunt.

ERROR MESSAGES – EASY

- trying to access beyond the limits of a list

```
test = [1,2,3]   then   test[4]        → IndexError
```

- trying to convert an inappropriate type

```
int(test)                              → TypeError
```

- referencing a non-existent variable

```
a                                      → NameError
```

- mixing data types without appropriate coercion

```
'3'/4                                  → TypeError
```

- forgetting to close parenthesis, quotation, etc.

```
a = len([1,2,3]
print(a)                               → SyntaxError
```

Logic errors - Hard

Keep copies of running code

Take a step back:

- Meditate.
- Go on a run. (Austin marathon?)
- Shower.

Explain the code to someone else:

- Preferably someone who doesn't know programming.

Debugging code examples

Don't: Do:

- Write entire program
- Test entire program
- Debug entire program

- Write a function
- Test the function, debug the function
- Write a function
- Test the function, debug the function
- *** Do integration testing ***

- Change code
- Remember where bug was
- Test code
- Forget where bug was or what change you made
- Panic

- Backup code
- Change code
- Write down potential bug in a comment
- Test code
- Compare new version with old version

Exceptions

exceptions syntax

```
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

Exceptions are very useful in scripting
(Usually hard to anticipate everything about arguments)

Handling **except**ions

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
finally:
    print('Goodbye, world!')
```

Raising exceptions

```
def get_ratios(L1, L2):
```

```
raise <exceptionName>(<arguments>)
```

rs

```
raise ValueError("something is wrong")
```

keyword

name of error you want to raise

optional, but typically a string with a message

```
    except ZeroDivisionError:
        ratios.append(float('nan')) #nan = not a number
    except:
        raise ValueError('get_ratios called with bad arg')
    return ratios
```

Assertions

```
def avg(grades):
    """ takes in a list of numbers and
     returns their average"""
    assert len(grades) != 0, 'no grades data'
    return sum(grades)/len(grades)
```

Q: What is better about assertions?

Summary

Abstraction and decomposition.

Functions syntax.

Local and global variables.

Coding practices.

Exceptions.