# Computer Programming I

Lecture 5:
Correctness and testing.
More classes

Hans  Karlsson

# Correctness, testing and errors

## What does it mean for a code to be correct?

The code runs, withour crashes, unless the crash is part of the specification.

The result of running the code is correct, and matchin the specification.

The code is reliable, and gives a valid result every time it is run (can be different if there is an element of randomness, or error for correct input).

The specification is a detailed description of what inputs are supposed, and what outputs are guaranteed, and how errors are handled.

## A function giving error is not wrong

If a function crashes the program, it may be working perfectly well.

If the input arguments are not within its supported rang, a well written function tends to det4ect this and provide an error signalling that incorrect input has been given.

It would be an error to not give an error.

## Error handling mechnisms in Python

Returning a special value (None, -1,…) etc.

str.find(substr) returns -1 is the substring was not found

Often not technically an eror, and may not be an good mehtod since it can hide code that is wrong, or used in a wrong way. ”Sweeping dirt under the carpet”

assert(condition)
Introduces a stated assumption that the condition is true. Should only be used to guard against programmer error, and not to catch bad input from users,

## Exceptions

Exceptions are errors that are raised when an error is detected, that can not be handled locally.

A function

```
import numbers
def square(x)
    if not isinstance(x, numbers.Number):
        raise ValueError('Argument does not have a squarable type.')
    return x*x
```

## Exceptions

```
>>> square(’abc’)
Traceback (most recent call last):
  File ”<stdin>”, line 7, in <module>
  File ”<stdin>", line 4, in square
    raise ValueError('Argument does not have a squarable type.')
ValueError: Argument does not have a squarable type.

>>>try:
... square('abc')
... except ValueError as e:
    ... print(e)
>>> Argument does not have a squarable type.
```

## Exceptions

```
def f(x):
    if isinstance(x,str):
        raise ValueError('Error: x is a string’)
     return x*x

def g(x):
    return x + f(x)

def h(x):
    try:
        return g(x)
    except ValueError as e:
        print(e)
```

The exception jumps out of f, and continues to exit g until it it caught in .

```
>>>print(h(4))
 20
>>> print(h(’a’))
Error: x is a string
None
```

## Example: Sorting

What does it mean when we say that a list has been sorted?

How do we know if the list actually si sorted properly after we have called sorted_lst = sorted(lst)

What happens if lst = [’abs’, 3] ?
We would expect an error!

```
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

## Specification of sorting

The list contains the same (number of) elements as the unsorted list.
We are not allowed to lose any elements.

We should get an error, rather than incorrect output if elements that can't be compared are contained in the list to be sorted.

All elements in the list should be ordered:
lst[n] >= lst[n+1] for all n

## How could we test a sorting function?

We cold write a function is_sorted(lst,unsorted_lst) that checks id lst
is a sorted version of the unsorted_lst.

Can testing guarantee correctnedd in general? No. Not at all. It can only raise the confidence that some code does what it is supposed to, but never provide a proof, unless exhaustive testing is possible.

Example: Take a function that has a single parameter with 1 000 000 different possible values in its domain. Then we could try all those inputs, and conclude that the function is proved to be correct.

# Classes User Defined Datatypes

## Classes

Classes provide a way to create new datatypes with data and logic/functionality combined in a single object.
Instances of classes can then be created to form objects, with attributes/state/data/member variables and methods.

# Classes

```
class Person:
        def __init__(self, name, address):
            self.name = name
                self.address = address

        def move(self, new_address):
            self.adress = new_address

        def __str__(self):
            return f'({self.name}:{self.address})’
```

```
>>> p = Person('Nisse','Uppsala')
>>> print(p)
(Nisse:Uppsala)
>>> p.move('Tierp')
>>> print(p)
(Nisse:Tierp)
```

## Classes

Now you can work with person objects as whole entitites:
These objects can be processed and handled similarily to the built in datatypes.

You can store them in lists, tuples, dictionaries:
SyntaxError: invalid syntax

```
>>> lst = [Person('Nisse','Uppsala'), Person('Stina','Stockholm')]
>>> tpl = (Person('Nisse','Uppsala'), Person('Stina','Stockholm'))
>>> dict = {'Friend': Person('Nisse','Uppsala'), 'Teacher': Person('Stina','Stockholm')}
```

# Classes

Class name

Attribute / Member variable

```
class Person:
        def __init__(self, name, address):
            self.name = name
                self.address = address

        def move(self, new_address):
            self.adress = new_address

        def __str__(self):
            return f'({self.name}:{self.address})'
```

Constructor – initializes the object. Called when the name of the class is invoked by Person()

A non-standard method that changes the state/data of a person-object.
Called like: p.move()

__str__ function is called when objects are printed, etc. Generates a string representation that can be displayed. See also __repr__.

## Classes

Classes can be made mutable, by creating methods that change the attributes/ data of an object.

It is often good to make classes immutable if you can,because that makes dealing with them easier.

However, sometimes it is a better solution to allow mutability i.e. to have the ability to change the class state/data after creation of an object.