CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College
Chapter 1

1.1 Introduction, 1.2 Functional Abstraction

In-Class Practice
When we are working on a large project it is important to limit the level of details to be considered at any given moment.

The process of ignoring some details while concentrating on those that are relevant to the problem at hand is called abstraction.
A functional abstraction (or simply abstraction) is the external view of a function, as seen from the program that calls it. This view is sometimes called the function’s interface.

The function interface consists of its specification (what the function does) and its signature (function name, list of parameter types, and the type of the value returned by the function).

**Interface Example: sqrt**

```python
def sqrt(x):
    """Computes the square root of x."""
```
A functional abstraction (or simply abstraction) is the external view of a function, as seen from the program that calls it. This view is sometimes called the function’s interface.

The function interface consists of its specification (what the function does) and its signature (function name, list of parameter types, and the type of the value returned by the function).

Interface Example: sqrt

```python
def sqrt(x):
    """Computes the square root of x.""
```
A functional abstraction (or simply abstraction) is the external view of a function, as seen from the program that calls it. This view is sometimes called the function’s interface.

The function interface consists of its specification (what the function does) and its signature (function name, list of parameter types, and the type of the value returned by the function).

**Interface Example: sqrt**

```python
def sqrt(x):
    """Computes the square root of x.""
```
Design by Contract

The program calling a function can be considered its client. Design by contract is formal specification of the preconditions (true before the function call) and postconditions (true after the function call).

Preconditions

Preconditions are what the client must provide if the function is to be expected to perform its task, as the client’s part of the contract.

Postconditions

Postconditions must be true after the function call, as the function’s part of the contract.
Design by Contract

The program calling a function can be considered its client. Design by contract is formal specification of the preconditions (true before the function call) and postconditions (true after the function call).

Preconditions

Preconditions are what the client must provide if the function is to be expected to perform its task, as the client’s part of the contract.

Postconditions

Postconditions must be true after the function call, as the function’s part of the contract.
Design by Contract

The program calling a function can be considered its client. Design by contract is formal specification of the preconditions (true before the function call) and postconditions (true after the function call).

Preconditions

Preconditions are what the client must provide if the function is to be expected to perform its task, as the client’s part of the contract.

Postconditions

Postconditions must be true after the function call, as the function’s part of the contract.
### Interface Example: `sqrt`

```python
def sqrt(x):
    """Computes the square root of x.""
```

### Doesn’t Really Help the Client

- what types of values does the function accept?
- does it return the result?
- how does it behave on “incorrect” input and what input is “incorrect”?  

### Another Issue

The implementer found a better algorithm/method for computing a square root. If we keep the abstraction barrier firmly in place, both the client code and implementation can change dramatically, but everything will continue to function properly.
Chapter 1

1.1 Introduction, 1.2 Functional Abstraction

In-Class Practice

BACK TO SQRT EXAMPLE

**Interface Example: sqrt**

```python
def sqrt(x):
    """Computes the square root of x."""
```

**Doesn’t Really Help the Client**

- What types of values does the function accept?
- Does it return the result?
- How does it behave on “incorrect” input and what input is “incorrect”?

**Another Issue**

The implementer found a better algorithm/method for computing a square root. If we keep the abstraction barrier firmly in place, both the client code and implementation can change dramatically, but everything will continue to function properly.
INTERFACE EXAMPLE: sqrt

```python
def sqrt(x):
    """Computes the square root of x."""
```

DOESN’T REALLY HELP THE CLIENT

- what types of values does the function accept?
- does it return the result?
- how does it behave on “incorrect” input and what input is “incorrect”?

ANOTHER ISSUE

The implementer found a better algorithm/method for computing a square root. If we keep the abstraction barrier firmly in place, both the client code and implementation can change dramatically, but everything will continue to function properly.
MODIFICATION TO SQRT’S INTERFACE

IMPROVED INTERFACE

```python
def sqrt(x):
    """Computes the square root of x.

    pre: x is an integer or a float, x ≥ 0
    post: returns the non-negative square root of x.""
```

WHERE TO PLACE SPECIFICATIONS IN CODE

- regular comments (start with #)
- docstrings
  (at the top of module, or immediately after function/class heading)
- docstrings should contain information for clients
- internal comments should contain information intended for the implementers
MODIFICATION TO SQRT’S INTERFACE

IMPROVED INTERFACE

```python
def sqrt(x):
    """Computes the square root of x.

    pre: x is an integer or a float, x ≥ 0
    post: returns the non-negative square root of x.""
```

WHERE TO PLACE SPECIFICATIONS IN CODE

- regular comments (start with #)
- docstrings
  (at the top of module, or immediately after function/class heading)
- docstrings should contain information for clients
- internal comments should contain information intended for the implementers
Defensive Programming is writing code that checks that all preconditions are met before allowing the program to proceed.

Testing can be performed in various ways: conditional statements, raising exceptions, and making assertions.
Testing for preconditions

**Conditional Statements**

A negative value returned means error.

```python
def sqrt(x):
    if x < 0:
        return -1
...
```

**drawback**: Each violation of a function precondition needs its own conditional statement in the calling program (client), which is inefficient, i.e. client program can become riddled with decision structures that constantly check whether an error has occurred.
 Exception Handling

This is better. Different types of error are handled together.

```python
try:
    y = sqrt(x)
except ValueError:
    print('bad parameter for sqrt')

def sqrt(x):
    if x < 0:
        raise ValueError('math domain error')
    if type(x) not in (type(1), type(1l),
    type(1.0)):
        raise TypeError('number expected')
...
Testing for preconditions

Assertions in Python

Often it is important only to detect if any error occurred but as early as possible. Python provides the `assert` statement.

```python
def sqrt(x):
    assert x >= 0 and type(x) in (type(1), type(1l), type(1.0))
    ...
```

`assert` takes a Boolean expression and raises `AssertionError` exception if the expression doesn’t evaluate to `True`.

**drawback:** takes a few CPU cycles to check these preconditions. With assertions, checking can be turned off when the code is compiled for production (with `-O` switch on the command line), but this is sometimes risky.
Top-down Design is the decomposition of a single task into several smaller ones.

Each task can then be written as a single function.

This can be repeated, producing an abstraction hierarchy in which high-level functions call lower level functions until the programming language itself provides its built-in functions at the bottom level.
Top-down Design

API

An application programming interface or API is the interface for a collection of functions performing related tasks. They are at the same level of abstraction, and when implemented they can call each other.
In-Class Practice

Example

Example: The task is to calculate some statistics (high score, low score, mean, standard deviation) for a given set of data (student exam scores). This overall task can be divided up:

- get scores
- calculate minimum score
- calculate maximum score
- calculate average score (mean)
- calculate standard deviation
- find the median

Once this “top level” has been designed, each subtask can be designed separately. The top level only uses the abstraction of each function.
A side effect is an unexpected result of a function call which modifies some variable or object in the environment of the function call.

These should always be documented as postconditions.

Many bugs are caused by undocumented side effects.

see the definition of the std.deviation function on page 16 (Section 1.2.4) in the book.
see the handout