Introduction to Course

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Guiding professors

- Lect. PhD. Arthur Molnar
- Lect. PhD. Radu Gaceanu
- Lect. PhD. Mircea Ioan-Gabriel
- Lect. PhD. Andrei Mihai
- Assist. Briciu Anamaria
- Assist. Imre Zsigmond
- Assist. Sergiu Nistor

Schedule

- **Lecture**: 2 hours/week (online)
- **Seminar**: 2 hours/week (physical presence/1 group online)
- **Laboratory**: 2 hours/week (physical presence/1 subgroup online)
- **Consultation**: optional, each teacher has a weekly time slot (will be announced on Teams)

Course materials

- **Teams**, **General** channel, **Files** section
- **FP** repository on GitHub Classroom

Contact us

Best way is using **Teams** chat

Objectives

What should you gain from this course?

- Learn key programming concepts
- Learn the basic concepts of software engineering (design, implementation and maintenance of software systems)
- Learn to use basic software tools such as IDE's, documentation generators, testing tools
- Acquire and improve your programming style.
- Learn the basics of programming using the Python language

Course content

How is this course organized?

- Programming in the large
- Programming in the small

Programming in the large

1. Procedural programming
2. Modular Programming
3. Test Driven Development
4. Design Principles for Modular Programs
5. User Defined Types and Exceptions
6. Introduction to UML
7. Design Principles for Object Oriented Programs
8. Program Testing. Refactoring.
9. Layered architecture. Inheritance.
10. Intro to building GUIs

Programming in the small

11. Recursion
12. Computational complexity
13. Searching. Sorting
14. Problem solving methods

Bibliography

1. Kent Beck - **Test Driven Development: By Example**; Addison-Wesley Longman, 2002.
2. Kleinberg and Tardos **Algorithm Design**; Pearson Educational; 2014 (http://www.cs.princeton.edu/ wayne/kleinberg-tardos/)
3. Martin Fowler - **Refactoring. Improving the Design of Existing Code**; Addison-Wesley, 1999. (http://refactoring.com/catalog/index.html)
4. Frentiu, M., H.F. Pop, Serban G. - **Programming Fundamentals**; Cluj University Press, 2006
5. Online Python resources - https://docs.python.org/3/reference/index.html, https://docs.python.org/3/library/index.html, https://docs.python.org/3/tutorial/index.html

Activity and grading

- **40%** - Laboratory work (assignments and tests **(L)**
- **20%** - Written exam (during exam session) **(W)**
- **40%** - Practical test (during exam session) **(T)**
- **0 - 0.5p** Seminar activity (bonus to laboratory grade)
- **0 - 1p** Additional laboratory activity (bonus to laboratory grade)

Passing the course

- Mandatory attendance to enter examination during 2022
- **L** grade $\geq$ 5 to enter examination during regular session
- **L**, **T** and **W** grades all $\geq$ 5 to pass the course

Activity and grading

Grading example

Suppose your grades are:

- Laboratory - 7
- Written - 7.50
- Practical - 6.80
- Seminar bonus - 0.30
- Laboratory bonus - 1

Your grade is calculated as: $0.4 * (7 + 0.3 + 1) + 0.2 * 7.5 + 0.4 * 6.8 = 7.54$, final grade is 8

About the Practical Exam

About the Practical Exam

- Only **working** functionalities are graded
- Everything required for implementation will be studied
- Each problem will be interesting, in its own way
- Getting the extra points during the semester will help improve your grade

Course Rules

- Seminar attendance mandatory **(10/14)**
- Laboratory attendance mandatory **(12/14)**
- Without making attendance you can't enter the exam this year!
- Detailed rules for laboratory activities are on the **General** channel, **Files** section
- **Be honest, solve the graded assignments by yourself, do not plagiarize!**

Introduction to Python

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Hardware and software

- **Hardware** -*computers* (desktop, mobile, etc) and related *devices*
- **Software** -*programs* or *systems* which run on hardware
- **Programming language** - notation that defines syntax and semantics of programs

What computers do

- Storage and retrieval
  - Internal memory
  - Hard disk, memory stick
- Operations
  - Processor
- Communication
  - Keyboard, mouse, display
  - Network connector

Lecture 01

Lect. PhD. Arthur Molnar

Introduction to Python

Data in Python

Simple Data Types

Compound Data Types

Variables, expressions and statements

- **Python** - a high level programming language. It is a great language for beginner programmers!
- **Python interpreter** - a program which allow us to run/interpret new programs.
- **Python standard library**: built-in functions and types

Python 101

Python is:

- A modern programming language
- Simple to write and understand
- An **interpreted** language
- A **garbage collected** language
- A language that support multiple paradigms: structured, object-oriented, functional and aspect oriented programming are all on the menu!
- A language with great support and many available libraries

Lecture 01

Lect. PhD. Arthur Molnar

Introduction to Python
Data in Python
Simple Data Types
Compound Data Types
Variables, expressions and statements

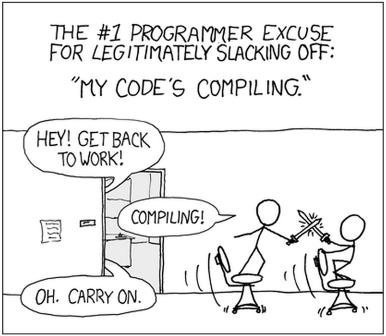
Python is...

Simple to write and understand

```
myList = []
while True:
    x = int(input("Enter item (-1 to finish):"))
    if x == -1:
        break
    myList.append(x)
return myList
```

Python is...

An **interpreted** language

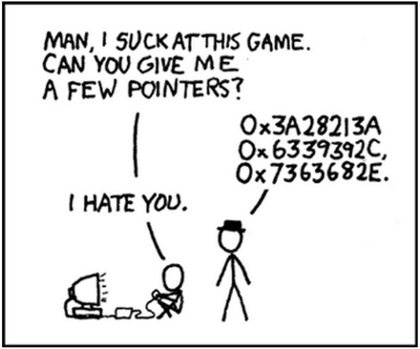


[1]

[1] https://xkcd.com/303/

Python is...

A **garbage collected** language



[2]

[2] https://xkcd.com/138/

Python mantra[3]:

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Flat is better than nested
- Sparse is better than dense
- Readability counts

[3] https://www.python.org/dev/peps/pep-0020/

What do you need?

1. Python 3
2. GitHub Desktop (**OR** use git in command line, **OR** git integration with IDEs)
3. An IDE (Eclipse + PyDev **OR** VS Code **OR** PyCharm)

OR

1. Use the PythonBox - a virtual machine we've prepared as a backup solution for the exam, which you can also use from home[4]

[4] http://www.cs.ubbcluj.ro/~arthur/PythonBox/

Basic elements of a Python program

- **Lexical elements** - a Python program is divided into a number of **lines**.
- **Comments** - start with a hash (#) character and ends at the end of the line.
- **Identifiers** (or **names**) - are sequences of characters which start with a letter (a..z, A..Z) or an underscore (_) followed by zero or more letters, underscores, and digits (0..9).
- **Literals** - are notations for constant values of some built-in types.

Demo

Basic elements of a Python program

ex01_BasicSyntax.py

Data vs. Information

- **Data** - collection of symbols stored in a computer (e.g. 123 decimal number or 'abc' string are stored using binary representations)
- **Information** - interpretation of data for human purposes (e.g. 123, 'abc')

Python data model

All data in Python programs is represented by objects, an **object** being Python's abstraction for data.
An **object** has:

- an **identity** - we may think of of it as the object's address in memory.
- a **type** - which determines the operations that the object supports and also defines the possible values.
- a **value**.

Types

- **Types** classify values. A type denotes a **domain** (a set of possible values) and **operations** on those values.
- **Numbers** - are immutable, so once created, their values cannot be changed.

Identity, value and type

Recall what is a *name* and an *object* (*identity*, *type*, *value*).

- mutable objects: lists, dictionaries, sets
- immutable: numbers, strings, tuples

We determine the identity and the type of an object using the built-in functions:

- **id**(object)
- **type(object)**, **isinstance(object, type)**

Standard types in Python (1/3)

int[5]:

- Represents the mathematical set of integers (positive and negative, unlimited precision)

bool:

- Represents the the truth values True and False.

float:

- Represents the mathematical set of double precision floating point numbers.

[5] https://docs.python.org/3/library/stdtypes.html

Standard types in Python (2/3)

Sequence types[6]

- Finite ordered sets indexed by non-negative numbers
- Let a be a sequence.
  - len(a) returns the number of items
  - a[0], a[1], ..., a[len(a)-1] represent the set of items
- Examples: [1, 'a']

string

- A string is an immutable sequence
- The items of a string are Unicode characters

[6] https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

Standard types in Python (3/3)

list[7]

- Mutable sequence of elements
- Typically used to store collections of homogeneous items
- Every item has a predecessor and successor

tuple[8]

- Immutable sequence
- Typically used to store collections of homogeneous items

dict[9]

- Mapping between unique keys and values

---

[7] https://docs.python.org/3/library/stdtypes.html#list
[8] https://docs.python.org/3/library/stdtypes.html#tuple
[9] https://docs.python.org/3/library/stdtypes.html#dict

Demo

Basic compound types

ex02_BasicCompoundTypes.py

List

Lists represent finite ordered sets indexed by non-negative numbers.

Operations:

- Creation
- Accessing values (index, len), changing values (**lists are mutable**)
- Removing items (pop), inserting items (insert)
- Slicing
- Nesting
- Generate list using range(), list in a for loop
- Lists as stacks (append, pop)

Tuple

Tuples are immutable sequences. A **tuple** consists of a number of values separated by commas.
Operations:

- Packing values (creation)
- Nesting
- Empty tuple
- Tuple with one item
- Sequence unpacking

Dictionary

A **dictionary** is an unordered set of (key, value) pairs with unique keys. The keys must be immutable.
Operations:

- Creation
- Getting the value associated to a given key
- Adding/updating a (key, value) pair
- Removing an existing (key, value) pair
- Checking whether a key exists

Variables and expressions

NB!

Variables are reserved memory locations to store values

- A **variable** has:
  - Name
  - Type
    - Domain
    - Operations

A variable is introduced in a program using a name binding operation - assignment.

Variables and expressions

- **Expression** - a combination of explicit *values*, *constants*, variables, *operators*, and *functions* that are interpreted according to the particular*rules of precedence*, which computes and then *produces/returns* another value.
- **Examples**:
  - numeric expression: $1 + 2$
  - boolean expression $1 < 2$
  - string expression: '1' + '2'

Statements

NB!

Statements are the basic operations of a program. A program is a sequence of statements

Statements

- **Assignment**
  - Assignments are used to (re)bind names to values
  - Bind name:
    - x = 1 *#is a variable (of type int)*
  - Rebind name:
    - x = x + 2 *#a new value is assigned to x*
  - Rebind name of mutable sequences:
    - y = [1, 2] *#mutable sequence*
    - y[0] = -1 *#the first item is bound to -1*
- **Block**
  - A block is a section of a program that is executed as a unit
  - A sequence of statements is a block
  - Blocks of code are denoted by line indentation

Demo

Controlling program flow

ex03_ProgramFlow.py

Procedural Programming

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Procedural programming

- A **programming paradigm** is a fundamental style of computer programming.
- **Imperative programming** is a programming paradigm that describes computation in terms of statements that change a program state.
- **Procedural programming** is imperative programming in which the program is built from one or more procedures (also known as subroutines or functions).

What is a function

A self contained block of statements that:

- Has a *name*,
- May have a list of (formal) *parameters*,
- May *return* a value
- Has a specification which consists of:
  - A *short description*
  - *Type and description of parameters*
  - Conditions imposed over input parameters (*precondition*)
  - Type and description for the return value
  - Conditions that must be true after execution (*post-condition*).
  - Any Exceptions raised

What is a function


```python
def maximum(x, y):
    """
    Return the maximum of two values
    input: x, y - the parameters to compare
    output: The largest of the parameters
    Error: TypeError - parameters cannot be compared
    """
    if x > y:
        return x
    return y
```

What is a function

- Can you tell what the function below does?
- Did it take more than a few seconds?

```
def f(c):
    b = []
    while not sol(b) and c != []:
        cand = next(c)
        c.remove(cand)
        if acceptable(b + [cand]):
            b.append(cand)
    if sol(b):
        found(b)
    return None
```

NB!

A function without specification is not complete!

What is a function

Every non-trivial, non-UI function written by you should:

- Use meaningful names (function name, variable names)
- Provide specification
- Include comments
- Have a test function (will come later)

What is a function

```
def greedy(c):
    '''
    Generic greedy algorithm
    input: c - set of candidates
    output: solution of generic problem
    '''
    # The empty set is the candidate solution
    b = []
    while not solution(b) and c != []:
        # Select best candidate (local optimum)
        candidate = selectMostPromising(c)
        c.remove(candidate)
        # If the candidate is acceptable, add it
        if acceptable(b + [candidate]):
            b.append(candidate)
    if solution(b):
        return b
    # In case no solution
    return None
```

What is a function

- A **function definition** is an executable statement introduced using the keyword **def**.
- The function definition does not execute the function body; this gets executed only when the function is called. A function definition defines a user-defined function object.

```
def maximum(x, y):
    """
    Return the maximum of two values
    input: x,y - the parameters to compare
    output: The largest of the parameters
    Error: TypeError - parameters cannot be compared
    """
    if x > y:
        return x
    return y
```

Variable scope

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. All variables defined at a particular indentation level or scope are considered local to that indentation level or scope

- Local variable
- Global variable

Demo

Variable scope

ex04_VariableScope.py

Variable scope

Rules to determine the scope of a particular name (variable, function name):

- A name defined inside a block is visible only inside that block
- Formal parameters belong to the scope of the function body (visible only inside the function)
- A name defined outside a function (at the module level) belongs to the module scope
- When a name is used in a code block, it is resolved using the nearest enclosing scope.

Variable scope

At any time during execution, names are resolved using:

- The innermost scope, which is searched first, contains the local names (inside the block)
- The scopes of any enclosing functions, which are searched starting with the nearest enclosing scope
- The next-to-last scope contains the current module's global names
- The outermost scope (searched last) is the namespace containing built-in names

Variable scope

- Use the **globals()** and **locals()** functions to figure out the scope of each variable

Recap

What other python built-in functions do you know?

Calls

A **block** is a piece of Python program text that is executed as a unit. Blocks of code are denoted by line indentation. A **function body** is a block. A block is executed in an *execution frame*. When a function is invoked a new execution frame is created.

Execution frames

`http://www.pythontutor.com/visualize.html`

Calls

An execution frame contains:

- Some administrative information (used for debugging)
- Determines where and how execution continues after the code block's execution has completed
- Defines two namespaces, the local and the global namespace, that affect execution of the code block.
- A *namespace* is a mapping from names (identifiers) to objects. A particular namespace may be referenced by more than one execution frame, and from other places as well.

Calls

- Adding a name to a namespace is called binding a name (to an object); changing the mapping of a name is called rebinding.
- Removing a name is unbinding.
- Namespaces are functionally equivalent to dictionaries (and often implemented as dictionaries).

Discussion

What did the output of locals(), globals() look like?

Parameter passing

- **Formal parameter** - an identifier for an input parameter of a function. Each call to the function must supply a corresponding value (argument) for each mandatory parameter
- **Actual parameter** - a value provided by the caller of the function for a formal parameter.
- The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called (arguments are passed *by object reference*)

Parameter passing

- **Pass by value** - the argument is evaluated, and a copy of the evaluation result is bound to the formal parameter of the function
- **Pass by reference** - function receives a reference to the actual argument, rather than a copy to its value
- **Side effect** - a function that modifies the caller's environment (beside producing a value) is said to have side effects

Demo

Parameter passing

ex05_ParameterPassing.py

Parameter passing

Discussion

What are the advantages and disadvantages of pass by value and pass by reference?

Parameter passing

How about in Python?

Object references are passed by value

Passing parameters

What happened in the studied example?

- At first, Python behaves like call-by-reference
- When you change a variable's value, it "switches" to call-by-value

Demo

Side Effects

ex06_SideEffects.py

A Working Program

ex07_RationalCalculator.py

Introduction to Software Development

Lect. PhD. Arthur Molnar

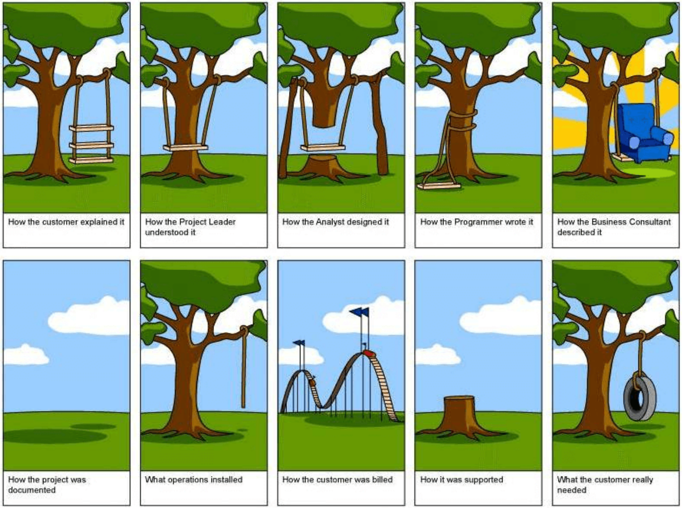
Babes-Bolyai University

Overview

Consultations schedule

- Each professor has weekly consultation hours
- This is the time and place to ask for extra help
- There is no grading!
- Schedule will be posted on MS Teams

Introduction to software development



Basic roles in software engineering

Programmers/Developers

- Use computers to *write/develop* programs for users

Testers/QA:

- Check the program to discover errors

Clients/stakeholders:

- Everyone affected by the outcome of a project

Users

- *Run programs on their computers*

Basic roles in software engineering

Software development process

An approach to building, deploying, and maintaining software. It indicates:

- What tasks/steps must be taken during development.
- In which order?

Basic roles in software engineering

A **software development process** is an approach to building, deploying, and maintaining software.

What we will use

Simple feature-driven development process

Example problem statement

A *problem statement* is a short description of the problem being solved.

Calculator

A *teacher* (client) needs a program for *students* (users) who learn or use rational numbers. The program shall help students make basic arithmetic operations

Demo

Simple feature-driven development

ex07_RationalCalculator.py

Requirements

Requirements - define in detail what is needed from the client perspective. Requirements define:

- What the client needs.
- What the system must include to satisfy the client's needs.

Requirements

Requirements guidelines

- Good requirements ensure your system works like your customers expect. (don't create problems to solve problems!)
- Capture the **list of features** your software is supposed to do.
- The list of features must clarify the problem statement ambiguities.

Features

A **feature** is a small, client-valued function:

- expressed in the form <**action**> <*result*> <object>,
  - action - a function that the application must provide
  - result - the result obtained after executing the function
  - object - an entity within the application that implements the function
- and typically **can be implemented within a few hours** (in order to be easy to make estimates).

F1.	**Add** number to calcularor
F2.	**Clear** calculator
F3.	**Undo** last operation

Simple feature-driven development

- An **iteration** is a set period of time within a project in which you produce a stable, executable version of the product, together with supporting documentation.
- An **iteration** will result in a working and useful program for the client (will interact with the user, perform some computation, show results)

Simple feature-driven development

- Build a feature list from the problem statement
- Plan iterations (at this stage, an iteration may include a single feature)
- For each iteration
    - Model planned features
    - Implement and test features

Simple feature-driven development

- **Example iteration plan**

Iteration	**Planned feature**
I1	F1. **Add** number to calcularor
I2	F2. **Clear** calculator
I3	F3. **Undo** last operation

Iteration modelling

At the beginning of each iteration you must understand the work required to implement it. You must **investigate/analyze** each feature in order to determine work items/tasks. Then, work items are scheduled. Each work item will be independently implemented and tested.

Iteration modelling

Iteration 1 - Add a number to calculator

- For simple programs (e.g. Calculator), running scenarios help developers understand what must be implemented.
- A **running scenario** shows possible interactions between users and the program under development.

Lecture 03

Lect. PhD. Arthur Molnar

Consultations schedule

Introduction to software development

Basic notions

Simple feature-driven development process

How to approach Assignments 3 and 4

Iteration 1 - Add a number to calculator

	User	Program	Description
a		0	Shows total
b	1/2		Adds number to calculator
c		1/2	Shows total
d	2/3		Adds number to calculator
e		5/6	Shows total
f	1/6		Adds number to calculator
g		1	Shows total
h	6/6		Adds number to calculator
i		2	Shows total

Work items/tasks

- Define a task for each operation not already provided by the platform, e.g. T1, T2.
- Define a task for implementing the interaction between User and Program, e.g. T4.
- Define a task for implementing all operations required by UI, e.g. T3.
- Determine dependencies between tasks (e.g. T4–> T3–> T2–>T1, where –> means depends on).
- Schedule items based on the dependencies between them.

Work items/tasks

Possible tasks for calculator application

Task	Description
T1	Compute the GCD of two integers
T2	Add two rational numbers
T3	Implement init, add and total operations
T4	Implement user interface

Test Cases

Test case - A set of *test inputs*, *execution conditions*, and *expected results* that you identify to evaluate a particular part of a program.

Inputs: a,b	**gcd(a,b)**
2,3	1
2,4	2
6,4	2
0,2	2
2,0	2
24,9	3
-2,0	ValueError
0,-2	ValueError

How to approach Assignments 3 and 4

- You have to implement a command-based user interface
- Commands must work **exactly** as provided
- Code must be divided into functions
- Each function must only do one thing
- Functions do I/O, or calculations, but not both!
- Non-UI functions must have specification
- Must be turned in no later than week 7

Introduction to Unit Testing. Exceptions

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

1. Introduction to unit testing

2. Exceptions
   - Exception handling
   - Specifications and exceptions

Testing

What is testing?

Observing the behavior of a program over many executions.

- We execute the program for some input data and compare the result we obtain with the known correct result.
- **Questions**:
  - How do we choose input data?
  - How do we know we have run enough tests?
  - How do we know the program worked correctly for a given test? (known as the oracle problem)

Testing

- Testing cannot prove program correctness, and cannot identify all defects in software. However, what it **can** prove is incorrectness, if at least one test case gives wrong results.
- **Problems with testing**
  - We cannot cover a function's input space
  - We have to design an oracle as complex as the program under test
  - Certain things are practically outside of our control (e.g. platform, operating system and library versions, possible hardware faults)

Unit Testing

- Tests that verify the functionality of a specific section of code, usually at function level.
- Testing is done in isolation. Test small parts of the program independently

How to test a function

1. Write the function's specification
2. Create a test function called $test\_ < function\_name >$ that has no input parameters, does no return anything and calls no functions except the one under test (e.g. $test\_add\_student()$)
3. Add test cases to the test function using Python's assert[1] keyword
4. Run the test function. It should fail with an *AssertionError*
5. Write the code for the function under test
6. Test functions that do not raise *AssertionError* must complete quietly

[1] `https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement`

Exceptions

An **exception** is an event that disrupts normal program flow

- Exceptions are present and used in many programming languages
- They are raised by code to signal an exceptional situation
- Your code will both raise (create) exceptions as well as "treat" them

NB!

The presence of an exception does not automatically mean that there's an error in the code

Exceptions

Most programming languages that support exceptions[2] use a common terminology and syntax

- Raising or throwing exceptions
- Catching or treating an exception
- Exception propagation
- **try** / **raise** (throw) / **except** (catch) keywords

[2] https://docs.python.org/3/tutorial/errors.html#exceptions

Exception handling

Exception handling is the process of handling error conditions in a program systematically by taking the necessary action.

```
try:
    # code that may raise exceptions
except ValueError:
    # code that handles the situation
```

Exception handling

A few points from the Python syntax above

- If you want to catch exceptions, the code has to be in a **try** - **except** block
- Exceptions are caught according to type
- A try block can catch **one**, **several** or **all** exception types
- Creating exceptions in your code is done using the **raise** keyword
- You can provide additional arguments (such as an error message) to exceptions you raise

Exception handling

Where is an exception handlded:

1. The function where the exception was raised
2. Any function that called the raising function (transitively)
3. The Python runtime, in which case program execution stops

Discussion

If the phrase *"unhandled exception has occurred in you application..."* sounds familiar, now you understand what happened!

Exceptions

Demo

Exceptions example, **ex08_Exceptions.py**

Exception handling

When to use exceptions?

- Signal an exceptional situation - the function is unable to do its work (e.g. function preconditions are violated, or the function encountered a situation in which it cannot make progress - a required file was not found, was not accessible, etc.)
- Enforce function preconditions
- Generally speaking, you should **not use** exceptions to control program flow!

Function specification

Is a way to abstract **functions** that only works if we provide:

- Meaningful function name
- Short description (the problem solved by the function)
- Type and meaning of each input parameter
- Conditions over the input parameters (**preconditions**)
- Type and meaning of each output parameter
- Relation between input and output parameters (**post condition**)
- **Exceptions** raised by the function

Exceptions and function specification

- **Precondition** - condition that must be true prior to running a section of code
- **Post condition** - condition that must be true after running a section of code

```
def gcd(a, b):
    '''
    Return the greatest common divisor of two
        positive integers
    a,b - integers
    Return the greatest common divisor of a and b
    Raise ValueError if a <= 0 or b <= 0
    '''
```

Modular Programming

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Modules

Modular programming - a software design technique that increases the extent to which software is composed of independent, interchangeable components called **modules**, each of which does one aspect within the program and contains everything necessary to accomplish this.

Modules

Modules are:

- Independent
- Interchangeable

Modules

Discussion

Why is modular programming needed? Advantages and drawbacks...

Modules

- Allows grouping related functionalities
- Allows easier delivery and deployment of related functionalities
- Helps with solving naming conflicts

Modules in Python

A Python module[1] - a file containing Python statements and definitions (executable statements).

- **Name**: The file name is the module name with the suffix ".py" appended
- **Docstring**: triple-quoted module doc string that defines the contents of the module file. Provide summary of the module and a description about the module's contents, purpose and usage.
- **Executable statements**: function definitions, module variables, initialization code

[1] https://docs.python.org/3/tutorial/modules.html

Importing modules

In order to use a module it must be imported first. The import statement:

1. Searches the global namespace for the module. If the module exists, it is already imported and nothing more needs to be done.
2. Searches for the module.
3. Variables and functions defined in the module are inserted into a new symbol table (a new namespace). Only the module name is added to the current symbol table

Module search path

Where does the **'import spam'** statement search for module *spam.py*?

- Built-in modules with the given name
- Directories in the *sys.path* variable:
  - Directory containing the input script
  - Directories specified by environment variable **PYTHONPATH**
  - Directories specified by the environment variable **PYTHONHOME**, an installation-dependent default path

If the module name can't be found anywhere, an **ImportError** exception is raised.

Demo

Modules

ex09_modules

Demo

Environment Variables

This website has more info on accessing and changing environment variables in the Windows OS - `www.computerhope.com/issues/ch000549.htm`

Learning more about modules

- **dir(module_name)** can be used to examine the module's symbol tables.
- **help(module_name)** can be used to get help on the module, its data types and functions.
- **pydoc** - A module that allows you to save extracted documentation to HTML format. Best used in command line at the operating system prompt.

Packages

- Packages[2] are a way of structuring Python's module namespace by using "dotted module names"
- **A.B** denotes submodule **B** found in package **A**.
- The same rules apply for importing packages as with modules
- On the drive, directory hierarchies represent packages, so **B.py** will be found in a directory called **A**
- Each package directory contains an *__init__.py* file, telling Python to interpret it as a collection of modules
- *__init__.py* can be empty, or include package initialization code.

[2] https://docs.python.org/3/tutorial/modules.html#packages

Required modules for Assignment 4

Create modules for:

- **User interface** - Functions related to user interaction. Contains input and data validation, print operations. This is the only module where input/print operations are present.
- **Functions** - Contains functions required to implement program features
- **Start** - Code that starts the program by calling the required UI function(s)

Demo

Code review

The code in the following archive is a modular implementation of the calculator program for rational numbers: **ex10_modular_calc**

User Defined Types

Lect Phd. Arthur Molnar

Babes-Bolyai University

Overview

User defined types

NB!

Types classify values. A type denotes a **domain** (a set of values) and **operations** on those values.

User defined types

- **Object oriented programming** - a programming paradigm that uses objects that have data and which "talk" to each other to design applications.

Why define new types?

Let's review the modular calculator example:

1. Issues with global variables, if they exist:
   - You can easily break global vars!
   - They make testing difficult
   - Managing the relation between them is difficult
2. Issues without global variables:
   - The state of the calculator is exposed to the world
   - The state has to be transmitted as parameter to every function

User defined types - classes

Class - a construct used as a template to create instances of itself - referred to as class instances, class objects, instance objects or simply **objects**. A class defines constituent members which enable these class instances to have *state* and behaviour.

Classes in Python

- Defined using the keyword **class** (as in many other languages)
- The class definition is an executable statement.
- The statements inside a class definition are usually function definitions, but other statements are allowed
- When a class definition is entered, a new namespace is created, and used as the local scope - thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

User defined types - objects

Object - in object-oriented programming, an object refers to a particular instance of a class, and is a combination of variables, functions and other data structures. Objects support two kinds of operations: **attribute (data or method) references** and **instantiation**.

User defined types - objects

1. Object instantiation - uses the reserved function notation of **__init__**
   - The instantiation operation creates an empty object that is of the type of the given class
   - A class may define a special method named __init__, used to create an instance of that class (e.g. class $->$ object)
   - In Python, use **self** to refer to that instance (in many other languages, it is the **this** keyword)

User defined types - objects

2. Attribute references (method or field)
    - Uses the "dot-notation", not dissimilar to package.module names.
    - We have instance variables/methods and class variables/methods
    - Instance variables are specific to an object (each object has its own instance)
    - Class variables are specific to a class (they are shared by all instances of that class)
    - The variable referencing the object specifies on which instance the call is made, in the case of instance variables

Existing data types

ex11_existingDataTypes.py

Fields, Methods

- Fields
  - Variables that store data specific to an instance or a class (see the slide above)
  - Can be objects themselves
  - They come into existence first time they are assigned to
- Methods
  - Functions in a class that can access values from a specific instance.
  - In Python the method will automatically receive a first argument: the current instance
  - All instance methods need to have the **self** argument

Fields, Methods

Demo

A first example using classes in Python - **ex12_pythonClassParticularities.py**

Demo

Let's create a new data type - RationalNumber. (Source code is in **ex13_rationalNumberBasic.py**)

Special methods

- **__str__** - converts the current object into a string type (good for printing)
- **__eq__** - test (logical) equality of two objects
- **__ne__** - test (logical) inequality of two objects
- **__lt__** - test x $<$ y
- Many others at[1]

[1] `https://docs.python.org/3/reference/datamodel.html`

Special methods - operator overloading

- **__add__(self, other)** - to be able to use "+" operator
- **__mul__(self, other)** - to be able to use the "*" operator
- **__setItem__(self,index, value)** - to make a class behave like an array/dictionary, use the "[]"
- **__getItem__(self, index)** - to make a class behave like an array
- **__len__(self)** - overload len
- **__getslice__(self,low,high)** - overload slicing operator
- **__call__(self, arg)** - to make a class behave like a function, use the "()"

Special methods - example

Demo

We should make our rational number type a bit more useful. (Source code is in **ex14_rationalNumberOperators.py**)

Python scope and namespace

NB!

- A *namespace* is a mapping from names to objects.
- Namespaces are implemented as Python dictionaries
  - Key: name
  - Value - Object
- Remember **globals()** and **locals()** ?

Python scope and namespace

- A class introduces a new namespace
- Methods and fields of a class are in a separate namespace (the namespace of the class)
- All the rules (bound a name, scope/visibility, formal/actual parameters, etc.) related to the names (function, variable) are the same for class attributes (methods, fields). Keep in mind that the class has its own namespace

Class vs instance attributes

- Instance attributes
  - The **self** reference decides for what object the attribute is accessed
  - Each instance has its own set of fields
- Class attributes
  - Attributes that are unique to the class
  - They are shared by all instances of the same class
  - In most languages, they are referred to as "static" fields, or methods
  - In Python, the **@staticmethod** decorator is used
  - Static methods do not receive the **self** reference

Class vs instance attributes

Demo

ex15_instanceVsClassAttributes.py

Class vs instance attributes

Discussion

Can you think of examples where class attributes are more suitable rather than instance attributes?

Encapsulation

- A set of rules or guidelines that you will use when deciding on the implementation of new data types
- What we will cover
  - Encapsulation
  - Information hiding
  - Abstract data types

Encapsulation

- The **state** of the object is the data that represents it (in most cases, the class fields)
- The **behaviour** is represented by the class methods
- Encapsulation means that **state** and **behaviour** are kept together, in one **cohesive** unit

Information hiding

- The internal representation of an object needs to be hidden from view outside of the object's definition
- Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state
- Divide the code into a public interface, and a private implementation of that interface

Information hiding

- Defining a specific interface and isolate the internals to keep other modules from doing anything incorrect to your data
- Limit the functions that are visible (part of the interface),so you are free to change the internal data without breaking the client code
- Write to the Interface, not the the Implementation
- If you are using only the public functions you can change large parts of your classes without affecting the rest of the program

Information hiding

Public and private members - data hiding in Python

- We need to protect (hide) the internal representation (the implementation)
- Provide accessors (getter) to the data
- Encapsulations is particularly important when the class is used by others

Information hiding

Public and private members - data hiding in Python

- Nothing in Python makes it possible to enforce data hiding - it is all based upon convention. use the convention: _name or __name for fields, methods that are " private"
- A name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.
- A name prefixed with two underscores (e.g. __spam) is private and name mangling (its actual name is replaced by the Python runtime) is employed

Guidelines

- Upper application layers do not have to know about implementation details of the methods or the internal data representation used by the code they call
- Code must work even when the implementation or data representation are changed
- Function and class specification have to be independent of the data representation and the method's implementation **(Data Abstraction)**

Abstract data types

- Operations are specified independently of their implementation
- Operations are specified independently of the data representation
- Abstract data type is a Data type + Data Abstraction + Data Encapsulation

Week 7 Test

- First test will be during week 7's lecture (dry run during week 6 lecture)
- You will be given a problem statement to solve in 80 minutes
- Test is open book, but must be taken individually
- Solutions will be checked for plagiarism
- Use modular programming, functions, but not classes
- Weight is 20% of laboratory grade (around the same as the first 5 lab assignments)

Design Principles for Modular Programs

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

1. Design principles for modular programs
   - Single Responsibility Principle
   - Separation of Concerns
   - Dependency
   - Layered Architecture

Organizing source code

What does it mean to organize source code?

- Determine what code goes where ... d'uh!
- We split code into functions, classes and modules
- The purpose of this section is to discuss a few principles that help us do it correctly

Modules

Discussion

What do we mean by **organizing the code correctly?**

Organizing source code

We use a few key design principles that help determine how to organize source code

- Single responsibility principle
- Separation of concerns
- Dependency
- Coupling and cohesion

Single responsibility principle

- Each function should be responsible for one thing
- Each class should represent one entity
- Each module should address one aspect of the application

Single responsibility principle - functions

Let's take the function below as example

- Implements user interaction
- Implements computation
- Prints

```
def filter_score(scoreList):
  st = input("Start score:")
  end = input("End score:")
  for score in scoreList:
    if score.get_value() > st and score.get_value()
        < end:
      print(score)
```

Single responsibility principle - functions

Why could the **filter_score** function change?

- The program's input format or channel changes
  - e.g. menu/command based UI as in Assignment 2/3-4
  - How about GUI/web/mobile/voice-based UI?
- The filter has to be updated

NB!

The **filter_score** function has 2 responsibilities

Single responsibility principle - modules

How did we characterize a module?

[modules] ... each of which accomplishes one aspect within the program and contains everything necessary to accomplish this.

Single responsibility principle - modules

Discussion

Is there any similarity between how we design a function and a module?

Single responsibility principle

Multiple responsibilities are...

- Harder to understand and use
- Difficult/impossible to test
- Difficult/impossible to reuse
- Difficult to maintain and evolve

Separation of concerns

- Separate the program into distinct sections
- Each section addresses a particular concern
- **Concern** - information that affects the code of a computer program (e.g. computer hardware that runs the program, requirements, function and module names)
- Correctly implemented, leads to a program that is easy to test and from which parts can be reused

Separation of concerns - example

Let's take the function below as example **(again!)**

```
def filter_score(scoreList):
  st = input("Start score:")
  end = input("End score:")
  for score in scoreList:
    if score.get_value() > st and score.get_value()
        < end:
         print(score)
```

Separation of concerns - the UI

The refactored function below only addresses the UI, functionalities are delegated to the **filter_score** function

```
def filter_score_ui(scoreList):
  st = input("Start score:")
  end = input("End score:")
  result = filter_score(scoreList,st,end)
  for score in result:
    print(score)
```

Separation of concerns - the test

The **filter_score** function can be tested using a testing function such as the one below

```
def test_filter_score():
  lst = [person("Ana",100)]
  assert filter_score(l,10,30)==[]
  assert filter_score(l,1,30)==lst
  lst = [person("Anna",100), person("Ion",40),
      person("P",60)]
  assert filter_score(lst,3,50)==[person("Ion",40)]
```

Separation of concerns - the operation

The **filter_score** function only has one responsibility!

```
def filter_score(lst, st, end):
    '''
    Filter participants
    lst - list of participants
    st, end - integer scores
    return list of participants filtered by score
    '''
    rez = []
    for p in lst:
        if p.get_score() > st and p.get_score() < end:
            rez.append(p)
    return rez
```

Separation of concerns

NB!

These design principles are in many cases interwoven!

Dependency

What is a **dependency**?

- Function level - a function invokes another function
- Class level - a class method invokes a method of another class
- Module level - any function from one module invokes a function from another module

Example

Say we have functions **a**,**b**,**c** and **d**. **a** calls **b**, **b** calls **c** and **c** calls **d**.
What might happen if we change function **d** ?

Coupling

- **Coupling** - a measure of how strongly one element is connected to, has knowledge of, or relies on other elements
- More connections between one module and others, the harder to understand that module, the harder to re-use it in another situation, the harder to test it and isolate failures
- **Low coupling** - facilitates the development of programs that can handle change because they minimize the interdependency between functions/modules

Cohesion

- **Cohesion** - a measure of how strongly related and focused the responsibilities of an element are.
- A module may have:
  - **High Cohesion**: it is designed around a set of related functions
  - **Low Cohesion**: it is designed around a set of unrelated functions
- A cohesive module performs a single task within an application, requiring little interaction with code from other parts of the program.

Cohesion

- Modules with less tightly bound internal elements are more difficult to understand
- Higher cohesion is better

NB!

Cohesion is a more general concept than the single responsibility principle, but modules that follow the SRP tend to have high cohesion.

Cohesion

NB!

Simply put, a cohesive module should do just one thing - **now where have I heard that before... ?**

How to apply these design principles

- **Separate concerns** - divide the program into distinct sections, so that each addresses a separate concern
- Make sure the modules are **cohesive** and **loosely coupled**
- Make sure that each module, class have **one responsibility**, or that there is only one reason for change

Layered Architecture

We employ the layered architecture pattern keeping in mind the detailed design principles

Layered Architecture

Structure the application to:

- **Minimize module coupling** - modules don't know much about one another, makes future change easier
- **Maximize module cohesion** - each module consists of strongly inter-related code

Layered Architecture

Layered Architecture - an architectural pattern that allows you to design flexible systems using components

- Each layer communicates only with the one immediately below
- Each layer has a well-defined interface used by the layer immediately above (hide implementation details)

Layered Architecture

Common layers in an information system architecture

- **User interface, Presentation** - user interface related functions, classes, modules
- **Domain, Application Logic** - provide application functions determined by the use-cases
- **Infrastructure** - general, utility functions or modules
- **Application coordinator** - start and stop application, instantiate components

Layered Architecture

Demo

Examine **ex16_rationalCalculator**

Exceptions and layered architecture

How do we integrate exceptions into layered architecture programs?

NB!

- UI module(s) should not do a lot of processing
- Non-UI modules should not have any UI input/output

Our solution:

- We create an exception instance with an argument or error message
- We catch exceptions in the UI and display the corresponding message

Testing. Debugging. Refactoring.

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Program testing

What is testing?

Testing is observing the behavior of a program over many executions.

Program testing

- We execute the program for some input data and compare the result we obtain with the known correct result.
- **Questions**:
  - How do we choose input data?
  - How do we know we have run enough tests?
  - How do we know the program worked correctly for a given test? (known as the oracle problem)

Program testing

- Testing cannot prove program correctness, and cannot identify all defects in software. However, what it **can** prove is incorrectness, if at least one test case gives wrong results.
- **Problems with testing**
  - We cannot cover a function's input space
  - We have to design an oracle as complex as the program under test
  - Certain things are practically outside of our control (e.g. platform, operating system and library versions, possible hardware faults)

Testing Approaches

Exhaustive testing

- Check the program for all possible inputs.
- Impractical for all but mostly trivial functions.
- Sometimes used with more advanced techniques (e.g. symbolic execution) for testing small, but crucial sections of a program (e.g. an operating system's network stack)

Testing Approaches

Boundary value testing

- Test cases use the extremes of the domain of input values, typical values, extremes (inside and outside the domain).
- The idea is that most functions work the same way for most possible inputs, and to find most of those possibilities where functions use different code paths.

Testing Approaches

Random testing, pairwise (combinatorial) testing, equivalence partitioning

- And the list goes on...

Testing Methods

Black box testing

- The source code is not available (it is in a "black", non-transparent box)
- The selection of test case data for testing is decided by analyzing the specification.

White box testing

- The source code is readily available (it is in a transparent box) and can be consulted when writing test cases.
- Selecting test case data is done by analyzing program source code. We select test data such that all code, or all execution paths are covered.
- When we say *"have 95% code coverage"* (Assignment6-8, bonus) it is white-box testing.

Demo

White and Black-box testing

Examine the test code in **ex16_blackBoxWhiteBox.py**

Advantages and drawbacks

Black box testing

+ Efficient for large code-bases
+ Access to source code is not required
+ Separation between the programmer's and the tester's viewpoint
- You do not know how the code was written, so test coverage might be low, testing might be inefficient

Advantages and drawbacks

White box testing

+ Knowing about the code makes writing it **AND** testing it easier
+ Can help find hidden defects or to optimize code
+ Easier to obtain high coverage
- Problems with code that is completely missing
- Requires access to source code
- Requires good knowledge of source code

White and Black-box testing

NB!

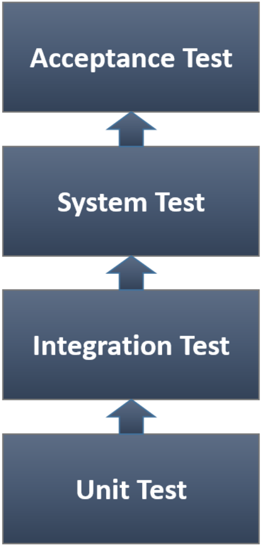
It's not a matter of which box is better, it's more like you have to make do with what you've got!

Testing levels

Testing Levels

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test

Testing levels



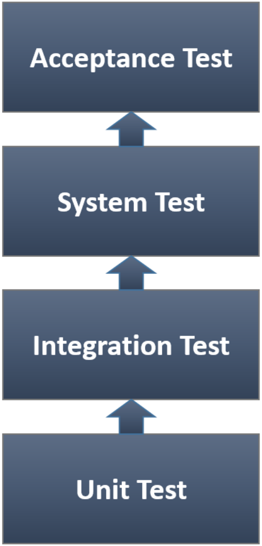
Unit Test

- Refers to tests that verify the functionality of a specific section of code, usually at function level.
- Testing is done in isolation. Test small parts of the program independently

Integration Test

- Test different parts of the system in combination
- In a bottom-up approach, it is based on the results of unit testing.

Testing levels



System Test

- Considers the way the program works as a whole.
- After all modules have been tested and corrected we need to verify the overall behavior of the program

Acceptance Test

- Check that the system complies with user requirements and is ready for use

Testing levels

Remember!

- What you did in Assignments 2, 3-4, 5 and 6-8 is unit testing.
- When you checked that your program worked through its UI, it was integration/system testing.

Automated testing

Automated testing

- Test automation is the process of writing a computer program to do testing that would otherwise need to be done manually.
- Use of software to control the execution of tests, comparison of actual outcomes to predicted outcomes, setting up test preconditions

PyUnit - Python unit testing framework

The unittest[1] module supports:

- Test automation
- Sharing setup and shutdown code for tests
- Aggregation of tests into collections
- Independence of tests from the reporting framework (another instance of the *single responsibility principle*)

[1] https://docs.python.org/3/library/unittest.html

Demo

PyUnit

Run the unit test in **ex17_PyUnit.py** in an IDE that supports this (e.g. PyCharm CE)
NB! This has to be run as a unit-test, and not a regular Python program

PyUnit - Python unit testing framework

The unittest module supports:

- Tests are implemented using classes derived from **unittest.TestCase**
- Test methods should start with the characters **test**
- We now use special methods instead of **assert** statements directly - **assertTrue()**, **assertEqual()**, **assertRaises()** and many more[2]
- The **setUp()** and **tearDown()** methods are run before and after each test method, respectively.

[2] `https://docs.python.org/3/library/unittest.html#assert-methods`

Automated testing

Discussion

How can we know when our test are "good enough" ?

The Coverage module

One (of the simpler) ways is to use code coverage

- Measure how much of the entire code was executed during the tests
- 0% coverage means no lines of code were executed
- 100% means **ALL** lines of code were executed at least once
- There exist tools which can measure and report this automatically

The coverage module

- PyCharm Professional can be used to gather coverage information by installing the *coverage*[3] module.

[3] `https://coverage.readthedocs.io/en/coverage-5.3/`

The coverage module

... or we can use it in command line

1. pip install coverage # installs the **coverage.py** module
2. open a **cmd/terminal** into your project's folder
3. **coverage** run -m unittest discover -p *.py && **coverage** report[4]
4. **coverage** html produces pretty printed output

[4] `https://stackoverflow.com/questions/47497001/python-unit-test-coverage-for-multiple-modules`

Test Driven Development Steps

Test Driven Development (TDD)

TDD requires developers to create automated unit tests that clarify code requirements before writing the code.

- Steps to apply TDD[5]:
  1. Create automated test cases
  2. Run the test (will fail)
  3. Write the minimum amount of code to pass that test
  4. Run the test (will succeed)
  5. Refactor the code

[5] Kent Beck. *Test Driven Development: By Example. Addison-Wesley Longman, 2002*. See also Test-driven development. http://en.wikipedia.org/wiki/Test-driven_development

Writing functions for TDD

1. Create a test
   - Define a test function ($test\_f()$) which contains test cases written using assertions.
   - Concentrate on the **specification** of **f**.
   - Define f: name, parameters, precondition, post-condition, and an empty body.

Writing functions for TDD

2. Run all tests and see that the new one fails
   - Your program has many functions, so it will also have many **test functions**
   - At this stage, ensure the new **test_f() fails**, while previously written test function pass
   - This shows that the test is actually executed and that it tests the correct function

Writing functions for TDD

3. Write the body of function **f()**
   - Writing the test before the function obliged you to clarify its specification
   - Now you concentrate on correctly implementing the function code
   - At this point, do not concentrate on technical aspects such as duplicated code or optimizations

Writing functions for TDD

4. Run all tests and see them succeed
   - Re-run the test you created at step 1
   - Now, you can be confident that the function meets its specification

Writing functions for TDD

5. Refactor code
   - **Code refactoring** is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior"[6].
   - **Code smell** is any symptom in the source code of a program that possibly indicates a deeper problem:
     - **Duplicated code**: identical or very similar code exists in more than one location.
     - **Long method**: a method, function, or procedure that has grown too large.

[6] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999. See also http:// refactoring.com/catalog/index. html

Writing functions for TDD

Lecture 08

Lect. PhD. Arthur Molnar

Program testing
Approaches
Black-box and White-box
Testing Levels
Automated testing

Test Driven Development (TDD)
Steps
Testing exception handling
Why TDD?

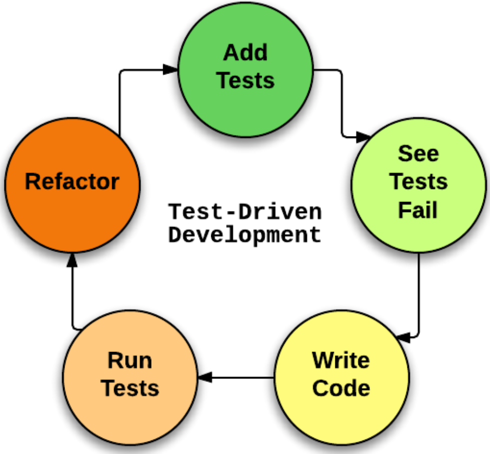
Debugging

Refactoring
Coding style
Refactoring
How to refactor

Discussion

How do I know my tests are good enough?

Test Driven Development (TDD)



[7]

[7] http://joshldavis.com/2013/05/27/difference-between-tdd-and-bdd/

Demo

Test Driven Development

ex18_TestDrivenDevelopment1.py

Test Driven Development

ex19_TestDrivenDevelopment2.py

Test cases for exceptions

How do we integrate exceptions into our test cases?

- Sometimes, a function works correctly if it raises an exception, and this must be tested

Demo

Test cases for functions raising exceptions, **ex19_TestDrivenDevelopment2.py**, test function **test_find_goldbach_primes**

Thoughts on TDD

- TDD is designed to take you out of the mindset of writing code first, and thinking later
- It forces you to **think** what each part of the program has to do
- It makes you analyse boundary behaviour, how to handle invalid parameters before writing any code

Debugging

Debugging

When you are the detective in a crime movie where you are also the murderer (various sources)

Debugging

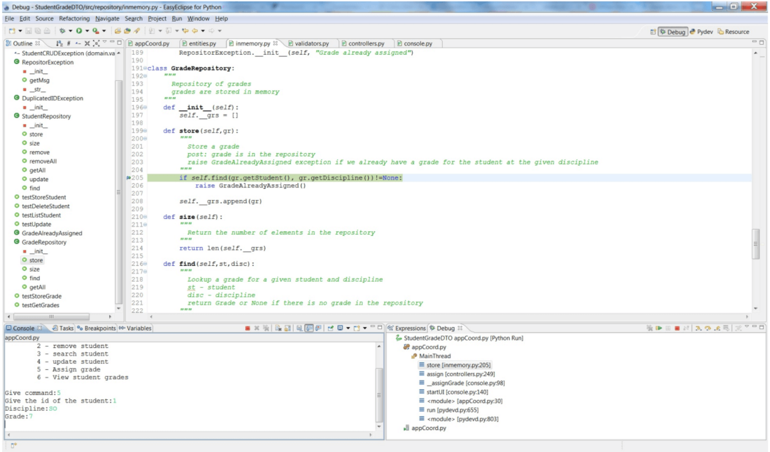
Debugging

The activity that must be performed when testing indicates the presence of errors, to identify errors, and rewrite the program with the purpose of eliminating them.

Two major approaches to debugging

- Using print statements
- Using the IDE

Eclipse debug perspective - Example



Eclipse debug perspective

Debug view

- View the current execution trace (stack trace)
- Execute step by step, resume/pause execution

Variables view

- View variable values

Program inspection

- Anyone can write code that computers understand. It's about writing code that humans also understand!
- Programming style consist of all the activities made by a programmer for producing code easy to read, easy to understand, and the way in which these qualities are achieved

Program inspection

- Readability is considered the main attribute of style.
- A program, like any publication, is a text must be read and understood by another programmer. The element of coding style are:
  - Comments
  - Text formatting (indentation, white spaces)
  - Specification
  - Good names for entities (classes, functions, variables) of the program
    - Meaningful names
    - Use naming conventions

Naming conventions

- Specific to each language, for Python they are encoded in the PEP-0008[8]
- Class names use camel case notation: Student, StudentRepository
- Variable names: student, nr_elem
- Function names: get_name, get_address, store_student
- constants are capitalized: MAX_LENGTH

[8] https://www.python.org/dev/peps/pep-0008

Refactoring

Refactoring

The process of changing the software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure.

- It is a disciplined way to clean up code that minimizes the chances of introducing bugs.
- When you need to add a new feature to the program, and the program's code is not structured in a convenient way for adding the new feature, first refactor the code to make it easy to add a feature, then add the feature

Why refactoring

- Improves the design of the software
- Makes software easier to understand
- Helps you find bugs
- Helps you program faster

Bad smells

When is refactoring needed?

- Duplicated code
- Long method/class
- Long parameter list (more than 3 parameters is seen as unacceptable)
- Comments

Sample code to refactor

The following file contains some examples of code that is good candidate for refactoring **ex20_refactoring.py**

Refactoring methods

1. **Rename Method** - *The name of a method does not reveal its purpose.*
2. **Consolidate Conditional Expression** - *You have a sequence of conditional tests with the same result.* Combine them into a single conditional expression and extract it.
3. **Consolidate Duplicate Conditional Fragments** - *The same fragment of code is in all branches of a conditional expression.* Move it outside the expression.

Refactoring methods

4. **Decompose Conditional** - *You have a complicated conditional (if-then-else) statement*. Extract methods from the condition, then part, and else parts.
5. **Inline Temp** - *You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings*. Replace all references to that temp with the expression.
6. **Introduce Explaining Variable** - *You have a complicated expression*. Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

Refactoring methods

7. **Remove Assignments to Parameters** - *The code assigns to a parameter*. Use a temporary variable instead.
8. **Remove Control Flag** - *You have a variable that is acting as a control flag for a series of boolean expressions*. Use a break or return instead.
9. **Remove Double Negative** - *You have a double negative conditional*. Make it a single positive conditional

Refactoring methods

10. **Replace Nested Conditional with Guard Clauses** - *A method has conditional behavior that does not make clear what the normal path of execution is.* Use Guard Clauses for all the special cases.
11. **Replace Temp with Query** - *You are using a temporary variable to hold the result of an expression.* Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.

Refactoring classes

12. **Encapsulate Field** - *There is a public field.* Make it private and provide accessors.
13. **Replace Magic Number with Symbolic Constant** - *You have a literal number with a particular meaning.* Create a constant, name it after the meaning, and replace the number with it.
14. **Extract Method** - *You have a code fragment that can be grouped together.* Turn the fragment into a method whose name explains the purpose of the method.

Refactoring classes

15. **Move Method** - *A method is, or will be, using or used by more features of another class than the class on which it is defined.* Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.
16. **Move Field** - *A field is, or will be, used by another class more than the class on which it is defined.* Create a new field in the target class, and change all its users.

GRASP Patterns (intro)

Arthur Molnar

Babes-Bolyai University

Overview

1. GRASP Patterns
   - High Cohesion
   - Low Coupling
   - Information Expert
   - GRASP Controller
   - Protect Variation
   - Creator
   - Pure Fabrication
2. GRASP Patterns in layered architecture
3. Domain-driven Design (intro)
   - Entities
   - Value Objects
   - Aggregates
   - Data Transfer Objects

GRASP Patterns

What are they?

General Responsibility Assignment Software Patterns (or **Principles**) consists of guidelines for assigning responsibility to classes and objects in object oriented design.

- The answer to **how** is layered architecture that we've "encouraged ☺" you to use in your assignments.
- The answer to **why** are these patterns.
- Their main goal is to make software easy to understand (by humans) and easy to change (by humans).

High Cohesion

Main idea

Create functions, classes and modules so that they have a single, well defined responsibility.

High Cohesion

- **High Cohesion** - attempts to keep objects focused, manageable and understandable.
- High cohesion means that the responsibilities of a given element are strongly related and highly focused.
- Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system.
- Low cohesion is a situation in which an element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and adverse to change

Low Coupling

Main idea

Minimize the dependencies between functions, classes and modules. A function call is a good example of a dependency.

Low Coupling

Low Coupling dictates how to assign responsibilities to support

- Low impact in a class when the source code for other classes is changed
- Good potential for reusing already written and tested code
- Good code readability by avoiding spaghetti code

Low Coupling

Forms of coupling:

- **TypeX** has an attribute (field) that refers to a **TypeY** instance, or **TypeY** itself.
- **TypeX** has a method which references an instance of **TypeY**, or **TypeY** itself, by any means. (parameter, local variable, return value, method invocation)
- **TypeX** is a direct or indirect subclass of **TypeY**.

Information Expert

Main idea

How do I decide where the code for a functionality goes.

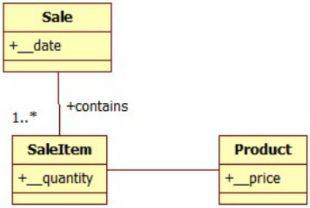
Information Expert

Assign a responsibility to the class that has the information necessary to fulfill the responsibility.

- Used to determine where to delegate responsibilities. These responsibilities include methods, computed fields and so on.
- Assign responsibilities by looking at a given responsibility, determine the information needed to fulfil it, and then figure out where that information is stored.
- Information Expert leads to placing responsibility on the class with the most information required to fulfil it

Information Expert

Point of Sale application



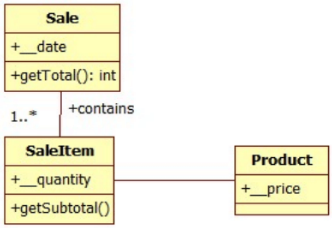
Who is responsible with computing the total?

Wee need all the SaleItems to compute the total.

Information Expert → **Sale**

Information Expert

Point of Sale application



According to the Expert

SaleItem should be responsible with computing the subtotal (quantity * price)

Information Expert

Point of Sale application

1. Maintain encapsulation of information
2. Promotes low coupling
3. Promotes highly cohesive classes
4. Can cause a class to become excessively complex

GRASP Controller

Main idea

Create a class that has a method for each user action. The first layer below the UI, its job is to *control* how the application fulfills required functionalities.

GRASP Controller

- Decouple the event sources (usually the application UI) from the objects that actually handle the events.
- It is the first object beyond the UI layer that receives and *controls* system operation.
- The controller should delegate to other objects the work that needs to be done

Protect Variation

Main idea

Create a class where I can control allowed object variations.

Protected Variations

- Make sure current or future variations in the system do not cause major problems with system operation.
- Create new classes to encapsulate such variations.
- The protect variation pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability to a separate class.

Protected Variations

Task: validate student, possible validation designs

- Class member function in Student that returns true/false
- Static function returning the list of errors
- Separate class that encapsulate the validation algorithm

Validator class

The protected variations pattern protects elements from variations on other elements (objects) by wrapping the focus of instability to a separate class

Creator

Main idea

How to decide who is responsible for creating domain entity objects.

Creator

- Object creation is one of the most common activities in an object-oriented system.
- Deciding who is responsible for this determines the relations between classes.
- Also, on non garbage-collected platforms, who is responsible for destroying objects? (object ownership)

Creator

- A class **B** should be responsible for creating instances of class **A** if one, or preferably more, of the following apply:
  - Instances of B contains or compositely aggregates instances of A
  - Instances of B record instances of A
  - Instances of B closely use instances of A
  - Instances of B have the initializing information for instances of A and pass it on creation.

Pure Fabrication

Main idea

Create a class that represents a persistent object store.

Pure Fabrication

- Assign a highly cohesive set of responsibilities to an artificial class that does not represent anything in the problem domain, in order to support high cohesion, low coupling, and reuse
- **Problem**: store **Student** (in memory, file or database)
  - Information expert says that *Student* is the "expert" to perform this operation
  - However, adding this functionality to *Student* means that we have to change domain entities when we want to update how we store domain entities.
  - We extract this functionality, breaking the information expert pattern to achieve good cohesion and coupling (and keep the force in balance ☺)

Layered architecture

Layer	Main ideas
All	High cohesion (each layer, class has a single, well-defined responsibility), low coupling (dependencies between layers and their classes are reduced and made clear)
User Interface	"Thin", contains as little functionality as possible
Controller	Application logic, the GRASP controller, uses the creator pattern
Domain	Entities from program domain
Validators	Protect variation, separate into its own class
Repository	Pure fabrication, responsible for storing domain entity objects

Entities

Entity

An object that is not defined by its attributes, but rather by a thread of continuity and its identity.

- If an object represents something with continuity and identity, it is something that is tracked through different states (or even across different implementations) it is an entity
- Usually has a correspondent in the real world

Entities

- Attributes of the entity may change but the identity remain the same
- Mistaken identity can lead to data corruption.
- Define what it means to be the same thing (e.g. if two objects have the same *type* and *id*)

Value Objects

Value Object

It describes a characteristic. It has no identity.

- An object that represents a descriptive aspect of the domain with no conceptual identity
- When you care only about the attributes of an element of the model:
  - *Address* is a good candidate for a value object; it is entirely determined by its attributes.
  - *Money* is another good example; each sum of equal value in the same currency are equal.
  - *Recipe ingredients* might be a good example; you can use any actual ingredients, as long as they are the right type and quantity.

Entities vs. Value Objects

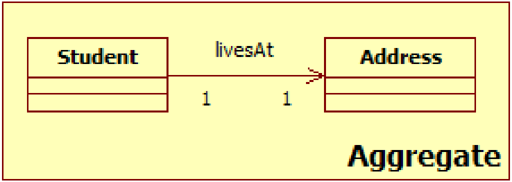
Discussion

- **Student** is an entity
- **Address** is a value object

Why isn't Student a value object?

Aggregates

- Cluster the **entities** and **value objects** into **aggregates** and define boundaries around them.
- Choose one **entity** to be the root of each aggregate, and control access to the objects inside the boundary using the root.
- Allow external objects to hold references to the root only.
- **e.g.** - only *StudentRepository*, **NOT** *AddressRepository*.



Data Transfer Objects

- **Data Transfer Objects (DTO)** are object used to carry data between processes.
- In the case where communication between processes is expensive (e.g. over the Internet), it makes sense to bundle up the data and send it in one go.
- DTO's have no behaviour, they only contain data, so should not require testing

NB!

Since our programs do not employ processes, we are not using DTO's exactly as intended. However, in real life you will find application layers on different machines/architectures.

UML. Files. Inheritance. GUI

Arthur Molnar

Babes-Bolyai University

Overview

UML Diagrams

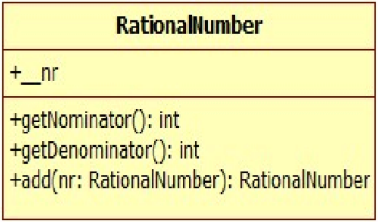
UML (Unified Modeling Language)

Standardized general-purpose modeling language in object-oriented software engineering.

- Includes a set of graphic notation techniques to create visual models of object-oriented software.
- It is language and platform agnostic (this is the whole point ☺)

Class Diagrams

UML Class diagrams - describe the structure of a system by showing the system's classes, their attributes, and the relationships between them.



Class Diagrams

```
class Rational:
    def __init__(self,a,b):
        '''
        Initialize a rational number
        a,b integers
        '''
        self.__nr = [a,b]
    def getDenominator(self):
        '''
        Denominator getter
        '''
        return self.__nr[1]
    def getNominator(self):
        '''
        Nominator getter
        '''
        return self.__nr[0]
```

Class Diagrams

In the diagram classes are represented using boxes which contain three parts:

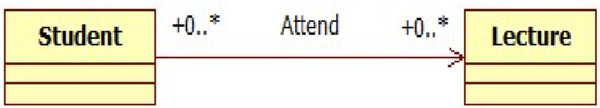
- Upper part holds the name of the class
- Middle part contains the attributes of the class
- Bottom part contains the methods or operations

Relationships

- A relationship is a general term covering the specific types of logical connections found on class diagrams.
- A *Link* is the basic relationship among objects. It is represented as a line connecting two or more object boxes.

Associations

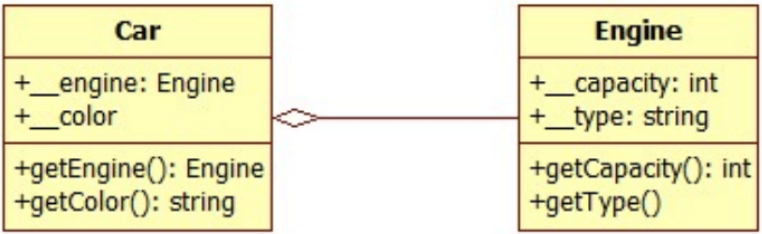
Binary associations (with two ends) are normally represented as a line, with each end connected to a class box.



An association can be named, and the ends of an association can be annotated with role names, ownership indicators, multiplicity, visibility, and other properties. Association can be Bi-directional as well as uni-directional.

Aggregation

Aggregation - an association that represents a part-whole or part-of relationship.



Aggregation

Aggregation - an association that represents a part-whole or part-of relationship.

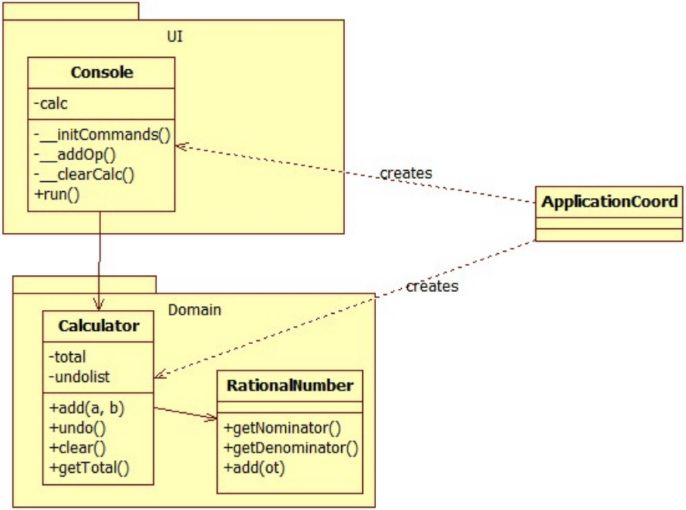
```
class Car:
    def __init__(self, eng, col):
        '''
        Initialize a car
        eng - engine, col - string, i.e 'white'
        '''
        self.__eng = eng
        self.__color = col
class Engine:
    def __init__(self, cap, type):
        '''
        Initialize the engine
        cap - positive integer, type - string
        '''
        self.__capacity = cap
        self.__type = type
```

Dependency, Package

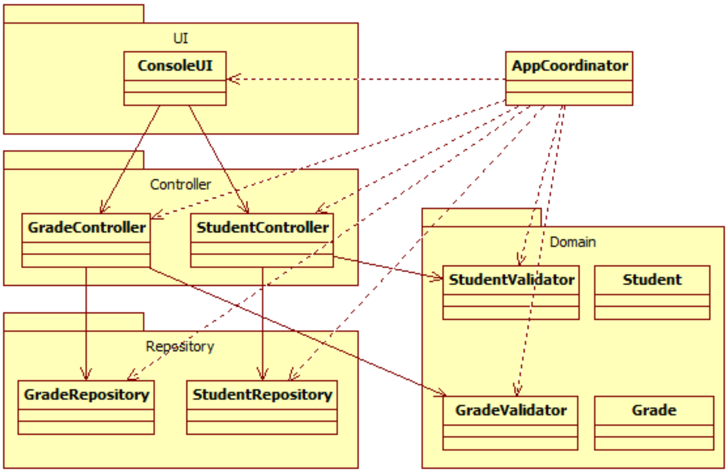
Dependency - a relationship in which one element, the client, uses or depends on another element, the supplier

- Create instances
- Have a method parameter
- Use an object in a method

Dependency, Package



UML class diagram for a Student Management app



Files

- The information on your computer is persisted using files.
- Files contain data, organized using certain rules (the file format).
- Files are organized in a hierarchical data structure over the file system, where directories (in most cases files, themselves) contain directories and files
- Operations for working with files: open (for read/write), close, read, write, seek.
- Files can be **text files** (directly human-readable) or **binary files**[1].

[1] Insert 10 types of people joke here ☺

Files

Possible problems when working with files

- Incorrect path/file given results in error.
- File does not exist or the user running the program does not have access to it.
- File is already open by a different program (e.g. when you try to delete a file in Windows but it does not allow you)

Text files in Python

Common operations[2]

- Built in function: **open(filename,mode)** returns a file object.
- *filename* - string representing the path to the file (absolute or relative path)
- *mode*:
  - "**r**" - open for read
  - "**w**" - open for write (overwrites the existing content)
  - "**a**" - open for append
  - "**b**" - binary file (e.g. "**rb**" is read-mode, binary file)

[2] https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

Text files in Python

Methods:

- *write(str)* - write the string to the file
- *readline()* - read a line from the file, return as a string
- *read()* - read the entire file, return as a string
- *close()* - close the file, free up any system resources

Text files in Python

Exception:

- **IOError** - raised exception if there is an input/output error.

Text files in Python

Demo

A simple example to get you started with reading and writing text files in Python. (**ex21_textFiles.py**).

Pickle is a Python module for saving/loading objects from a binary file[3]

- *load(f)* - load the data from the file
- *dump(object,file)* - write the *object* to the given file in pickle's own format
- In order to use Pickle, you must **f.open()** using "**rb**" and "**wb**" (read binary and write binary, respectively)

[3] `https://docs.python.org/3/library/pickle.html#module-pickle`

Object serialization with Pickle

Demo

A simple example to get you started with Pickle is in (**ex22_pickleFiles.py**).

Inheritance

- Classes can inherit attributes and behavior (i.e., previously coded algorithms associated with a class) from pre-existing classes called **base classes** (or superclasses, or parent classes)
- The new classes are known as **derived classes** or **subclasses** or child classes. The relationships of classes through inheritance gives rise to a hierarchy.

NB!

Inheritance defines an **is a** relationhip between the derived and base classes.

Inheritance for code reuse

- We can reuse code that already exist in another class.
- We can replace one implementation with another, more specialized one.
- With inheritance, base class behaviour can be inherited by subclasses. It not only possible to call the overridden behaviour (method) of the ancestor (superclass) before adding other functionalities, one can override the behaviour of the ancestor completely.

Inheritance in Python

- Syntax: **DerivedClassName**(BaseClassName)[4]:
- DerivedClass will inherit:
  - Fields
  - Methods
- If a requested attribute (field,method) is not found in the class, the search proceeds to look in the base class
- Derived classes may override methods of their base classes.
- An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name.
- There is a simple way to call the base class method directly: call *BaseClassName.methodname(self,arguments)*

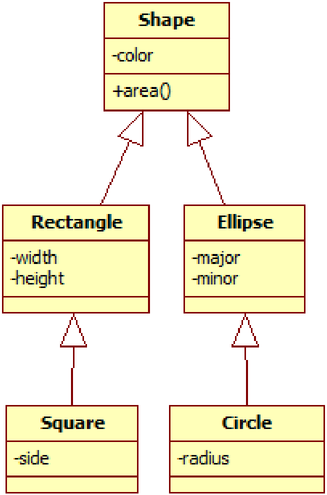
[4] https://docs.python.org/3/tutorial/classes.html#inheritance

Demo

Inheritance in Python

Examine the source code in **ex23_inheritance.py**

Demo - UML class diagram



Inheritance

NB!

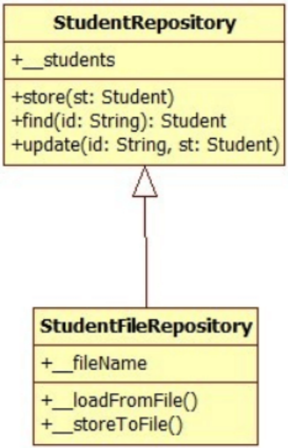
- The generalization relationship ("**is a**") indicates that one of the two related classes (the subclass) is considered to be a specialized form of the other.
- Any instance of the subtype is also an instance of the superclass.

Case Study I - File Repositories

- We would like to load/save problem entities to persistent storage.
- We already have a repository implementation, we're only missing the persistent storage functionality.
- We use **inheritance** to create a more specialized repository implementation, one that saves to/loads from files.

File Repositories

This is the UML class diagram for the repository implementation for the **Student** entity.



File Repositories

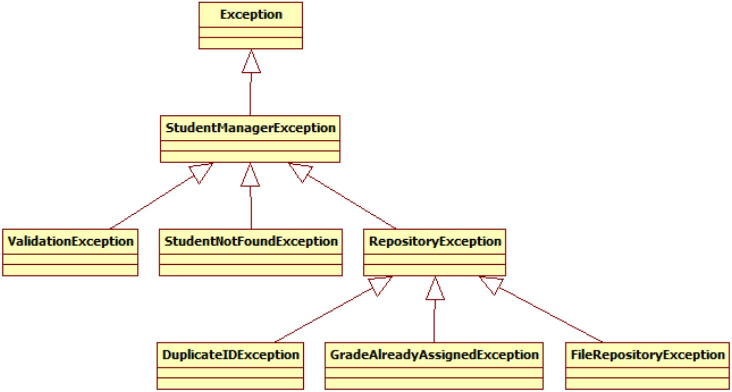
NB!

The application must work with either a *memory*, a *text file* or a *binary-file* backed repository implementation. Remember, modules are **independent** and **interchangeble**

Case Study II - Exception hierarchies

- We use exceptions to handle errors and special situations in the application
- Our exception classes are derived from **Exception**, a class that comes with the Python environment
- To handle different situations, most applications implement their own exception hierarchy

Example from a student management application



Exception hierarchies - example

What happens when we initialize the *Repository*?

- In memory implementation does not raise exceptions
- File-based implementation might raise *IOError* (input file not found, open another program, etc.)
- Database-backed implementation might raise *SQLConnectorException* (database server not started or cannot be reached)
- NoSQL database implementation might raise *CouchbaseException* (database server not started or cannot be reached)

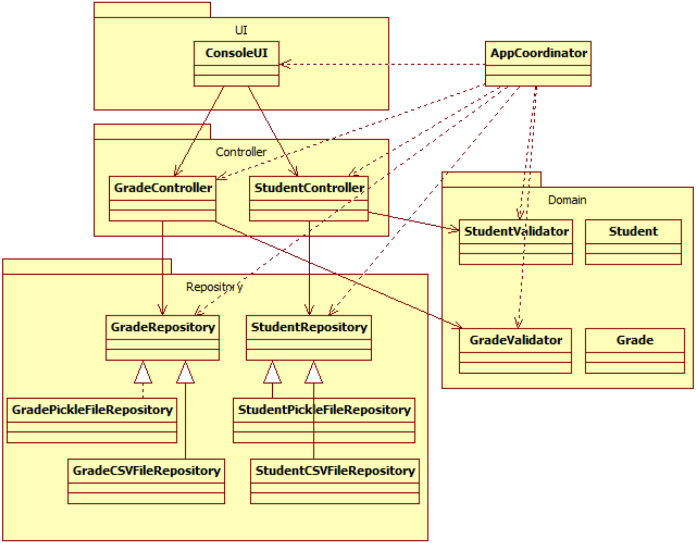
Exception hierarchies - example (cont.)

- Higher layers (*Controller*, *UI*) have to be independent from lower-layer implementations
- We cannot make the *Controller* or *UI* handle each possible exception type that a *Repository* might raise.

Solution

Define a *RepositoryException*. The repository code catches exception types that could be raised (e.g. *IOError*, *SQLConnectorException*) and re-raises them in the form of a *RepositoryException*

UML class diagram for student management app



About GUIs

- GUI applications are built using tool sets such as TkInter, AWT, Swing, SWT, WPF, JavaFX, QT and many, many more
- What these libraries provide
  - Graphical components such as buttons, lists, tables, and so on (also called **widgets**).
  - Management of events (e.g. what happens when you click a button)

About GUIs

- To build your first GUI, you must essentially take three steps
  1. Build the window and fill it with widgets
  2. Tell the GUI library which events you want to handle and how (known. Basically when an event is encountered (e.g. a button is clicked) a function is called.
  3. Start the main event loop.

About GUIs

What to consider

- The GUI code must be contained within the program's presentation layer
- Your program must work both with a GUI as well as using a console UI
- Switching between them must be (very) easy

Demo

Without further ado

Let's examine the code from **ex24_gui**

Recursion. Computational complexity

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Recursion

Circular definition

In order to understand recursion, one must first understand recursion.

What is recursion?

- A recursive definition is used to define an object in terms of itself.
- A recursive definition of a function defines values of the functions for some inputs in terms of the values of the same function for other inputs.
- Recursion can be:
  - **Direct** - a function **p** calls itself
  - **Indirect** - a function **p** calls another function, but it will be called again in turn

Demo

Recursion

Examine the source code in **ex25_recursion.py**

Recursion

- **Main idea**
  - Base case: simplest possible solution
  - Inductive step: break the problem into a simpler version of the same problem plus some other steps
- **How recursion works**
  - On each method invocation a new symbol table is created. The symbol table contains all the parameters and the local variables defined in the function
  - The symbol tables are stored in a stack, when a function is returning the current symbol tale is removed from the stack
- **Recursion and stack memory**
  - Stack memory size is allocated by the compiler/runtime environment
  - Compilers can optimize recursive computation (e.g. see Ackermann's function)

Recursion

- **Advantages**
  - + Clarity
  - + Simplified code
- **Disadvantages**
  - – Large recursion depth might run out of stack memory
  - – Large memory consumption in the case of branched recursive calls (for each recursion a new symbol table is created - see Ackermann's function)

Computational complexity

What is it?

Studying algorithm efficiency mathematically

- **We study algorithms with respect to**
  - Run time required to solve the problem
  - Extra memory required for temporary data
- **What affects runtime for a given algorithm**
  - Size and structure of the input data
  - Hardware
  - Changes from a run to another due to hardware and software environment

Running time example

As a first example, lets take a well-understood function: computing the nth term of the Fibonacci sequence

- What is so special about it?
  - Easy to write in most programming languages
  - Iterative and recursive implementation comes naturally
  - Different run-time complexity!

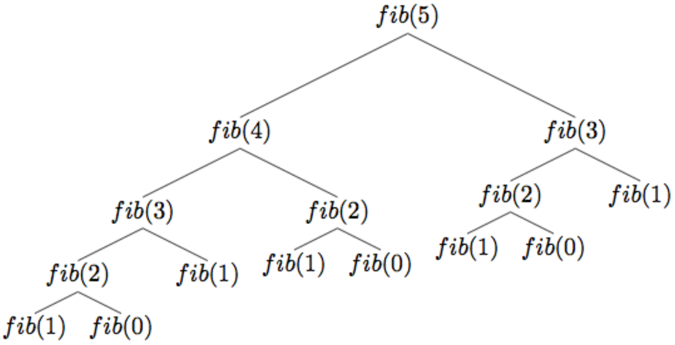
Demo

Computational complexity

Examine the source code in **ex26_complexity.py**[1]

[1] To run the example, install the texttable component from https://github.com/foutaise/texttable

Overcalculation in recursive Fibonacci



Demo

Discussion

How can overcalculation be eliminated?

Memoization

Examine the source code in **ex27_complexityOptimized.py**[2]

[2] To run the example, install the texttable component from https://github.com/foutaise/texttable

Efficiency of a function

What is function efficiency?

The amount of resources they use, usually measured in either the space or time used.

- **Measuring efficiency**
  - **Empirical analysis** - determines exact running times for a sample of specific inputs, but cannot predict algorithm performance on all inputs.
  - **Asymptotic analysis** - mathematical analysis that captures efficiency aspects for all possible inputs but cannot provide execution times.
- **Function run time is studied in direct relation to data input size**
- We focus on asymptotic analysis, and illustrate it using empirical data.

Complexity

- **Best case (BC)**, for the data set leading to minimum running time $\text{BC(A)} = \min_{I \in D} E(I)$
- **Worst case (WC)**, for the data set leading to maximum running time $\text{WC(A)} = \max_{I \in D} E(I)$
- **Average case (AC)**, average running time of the algorithm $\text{AC(A)} = \sum_{I \in D} P(I)E(I)$

Legend

A - algorithm; **D** - domain of algorithm; **E(I)** - number of operations performed for input **I**; **P(I)** the probability of having **I** as input data

Complexity

Observation

Due to the presence of the **P(I)** parameter, calculating average complexity might be challenging

Run time complexity

The essence

- How the running time of an algorithm increases with the size of the input at the limit: **if** $n \to \infty$**, then** $3n^2 \approx n^2$
- We compare algorithms using the magnitude order of the number of operations they make

Run time complexity

- Running time is not a fixed number, but rather a function of the input data size n, denoted $T(n)$.
- Measure basic steps that the algorithm makes (e.g. number of statements executed).

\+ It gets us within a small constant factor of the true runtime most of the time.

\+ Allows us to predict run time for different input data

− Does not exactly predict true runtime

Run time complexity

- **Example:**

$$T(n) = 13 * n^3 + 42 * n^2 + 2 * n * \log_2 n + 3 * \sqrt{n}$$

- Because $0 < \log_2 n < n, \forall n > 1$, and $\sqrt{n} < n, \forall n > 1$, we conclude that the n^3 term dominates for large n.
- Therefore, we say that the running time $T(n)$ grows *"roughly on the order of n^3"*, and we write it as $T(n) \in O(n^3)$.
- Informally, the statement above means that *"when you ignore constant multiplicative factors, and consider the leading term, you get n^3"*.

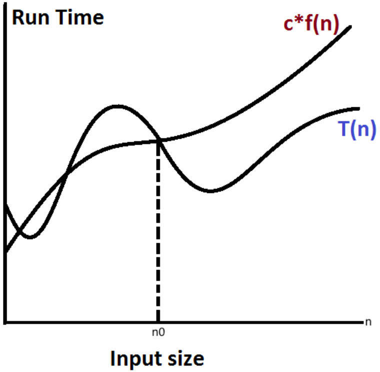
"Big-O" notation

We denote function $f : \mathbb{N} -> \mathbb{R}$, and by T the function that gives the number of operations performed by an algorithm, $T : \mathbb{N} -> \mathbb{N}$.

Definition, "Big-oh" notation

We say that $T(n) \in O(f(n))$ if there exist c and n_0 positive constants independent of n such that
$0 \leq T(n) \leq c * f(n), \forall n \geq n_0$.

"Big-O" notation



- In other words, $O(n)$ notation provides the asymptotic upper bound.

"Big-O" notation

Alternative definition, "Big-oh" notation

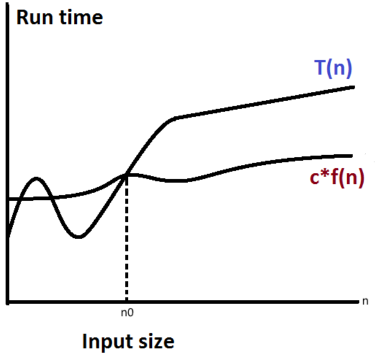
We say that $T(n) \in O(f(n))$ if $\lim_{n->\infty} \frac{T(n)}{f(n)}$ is 0 or a constant, but not ∞.

- If $T(n) = 13 * n^3 + 42 * n^2 + 2 * n * \log_2 n + 3 * \sqrt{n}$, and $f(n) = n^3$, then $\lim_{n->\infty} \frac{T(n)}{f(n)} = 13$. So, we say that $T(n) \in O(n^3)$.
- The O notation is good for putting an upper bound on a function. We notice that if $T(n) \in O(n^3)$, it is also $O(n^4)$, $O(n^5)$, since the limit will then go to 0.
- To be more precise, we also introduce a lower bound on complexity.

"Big-omega" notation

Definition, "Big-omega" notation

We say that $T(n) \in \Omega(f(n))$ if there exist c and n_0 positive constants independent of n such that
$0 \leq c * f(n) \leq T(n), \forall n \geq n_0$.



"Big-omega" notation

Alternative definition, "Big-omega" notation

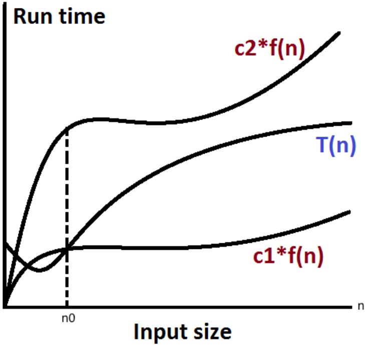
We say that $T(n) \in \Omega(f(n))$ if $\lim_{n->\infty} \frac{T(n)}{f(n)}$ is a constant or ∞, but not 0.

- If $T(n) = 13 * n^3 + 42 * n^2 + 2 * n * \log_2 n + 3 * \sqrt{n}$ and $f(n) = n^3$, then $\lim_{n->\infty} \frac{T(n)}{f(n)} = 13$. So, we say that $T(n) \in \Omega(n^3)$.
- The Ω notation is used for establishing a lower bound on an algorithm's complexity.

"Big-theta" notation

Definition, "Big-theta" notation

We say that $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$, i.e. there exist c_1,c_2 and n_0 positive constants, independent of n such that $c_1 * f(n) \leq T(n) \leq c_2 * f(n), \forall n \geq n_0$.



"Big-theta" notation

Alternative definition, "Big-theta" notation

We say that $T(n) \in \Theta(f(n))$ if $\lim_{n->\infty} \frac{T(n)}{f(n)}$ is a constant (but not 0 or ∞).

- If $T(n) = 13 * n^3 + 42 * n^2 + 2 * n * \log_2 n + 3 * \sqrt{n}$ and $f(n) = n^3$, then $\lim_{n->\infty} \frac{T(n)}{f(n)} = 13$. So, we say that $T(n) \in \Theta(n^3)$. This can also be deduced from $T(n) \in O(n^3)$ and $T(n) \in \Omega(n^3)$
- The run time of an algorithm is $\Theta(f(n))$ if and only if its worst case run time is $O(f(n))$ and best case run time is $\Omega(f(n))$.

Summations


```
for i in data_list:
    # do something here...
```

- Assuming that the loop body takes f(i) time to run, the total running time is given by the summation

$$T(n) = \sum_{i=1}^{n} f(i)$$

Observation

Nested loops naturally lead to nested sums.

Summation

Solving summations breaks down into two basic steps

- Simplify the summation as much as possible - remove constant terms and separate individual terms into separate summations.
- Solve each of the remaining simplified sums.

Summation - examples


```
def f(n):
    s = 0
    for i in range(1,n+1):
        s=s+1
    return s
```

- $T(n) = \sum_{i=1}^{n} 1 = n \Rightarrow T(n) \in \Theta(n)$
- BC/AC/WC complexity is the same

Summation - examples


```python
def f(n):
    i = 0
    while i <= n:
        # do something here...
        i += 1
```

- $T(n) = \sum_{i=1}^{n} 1 = n \Rightarrow T(n) \in \Theta(n)$
- BC/AC/WC complexity is the same

Summation - examples

```python
def f(l):
    '''
    Return True if list contains an even number
    '''
    poz = 0
    while poz < len(l) and l[poz]%2 !=0:
        poz += 1
    return poz<len(l)
```

- BC - first element is even number, $T(n) = 1$, $T(n) \in \Theta(1)$
- WC - no even number in list, $T(n) = n$, $T(n) \in \Theta(n)$

Summation - examples


```python
def f(l):
    '''
    Return True if list contains an even number
    '''
    poz = 0
    while poz < len(l) and l[poz]%2 !=0:
        poz += 1
    return poz<len(l)
```

- AC - the **while** can be executed $1, 2, ...n$ times, with same probability (lacking additional information). The number of steps is then the average number of iterations: $T(n) = \frac{1+2+..+n}{n} = \frac{n+1}{2} \Rightarrow T(n) \in \Theta(n)$

Summation - examples

```
def f(n):
    for i in range(1,2*n-1):
        for j in range(i+2,2*n+1):
            # do something...
```

- $T(n) = \sum_{i=1}^{2n-2} \sum_{j=i+2}^{2n} 1 = \sum_{i=1}^{2n-2} (2n - i - 1)$
- $T(n) = \sum_{i=1}^{2n-2} 2n - \sum_{i=1}^{2n-2} i - \sum_{i=1}^{2n-2} 1$
- $T(n) = 2n * \sum_{i=1}^{2n-2} 1 - \frac{(2n-2)(2n-1)}{2} - (2n-2)$
- $T(n) = 2 * n^2 - 3 * n + 1 \in \Theta(n^2)$.

Summation - examples

```
def f():
    for i in range(1,2*n-1):
        j = i+1
        cond = True
        while j<2*n and cond:
            # do something ...
            if someCondition:
                cond = False
```

- Best Case - while executed once,

$$T(n) = \sum_{i=1}^{2n-2} 1 = 2n - 2 \in \Theta(n)$$

- Worst Case - while executed $2n - i - 1$ times,

$$T(n) = \sum_{i=1}^{2n-2} (2n - i - 1) = ... = 2n^2 - 3n + 1 \in \Theta(n^2)$$

Summation - examples

```
def f():
    for i in range(1,2*n-1):
        j = i+1
        cond = True
        while j<2*n and cond:
            # do something ...
            if someCondition:
                cond = False
```

- Average Case - for a given i the "while" loop can be executed $1, 2, ..., 2n - i - 1$ times, average steps: $c_i = \frac{1+2+...+2n-i-1}{2n-i-1} = ... = 2n - i$
- $T(n) = \sum_{i=1}^{2n-2} c_i = \sum_{i=1}^{2n-2} 2n - i = ... \in \Theta(n^2)$
- Overall complexity is therefore $\Theta(n^2)$

Summation - important sums

- Constant series $\sum_{i=1}^{n} 1 = n$
- Arithmetic series $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Quadratic series $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{2}$
- Harmonic series $\sum_{i=1}^{n} \frac{1}{i} = \ln(n) + O(1)$
- Geometric series $\sum_{i=1}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1$

Common complexities

- **Constant time**: $T(n) \in O(1)$. It means that run time does not depend on size of the input. It is very good complexity.
- $T(n) \in O(\log_2 \log_2 n)$. This is also a very fast time, it is practically as fast as constant time.
- **Logarithmic time**: $T(n) \in O(\log_2 n)$. It is the run time of binary search and height of balanced binary trees. About the best that can be achieved for data structures using binary trees. Note that $\log_2 1000 \approx 10$, $\log_2 1000^2 \approx 20$.

Common complexities

- **Polylogarithmic time**: $T(n) \in O((\log_2 n)^k)$.
- **Liniar time**: $T(n) \in O(n)$. It means that run time scales liniarly with the size of input data.
- $T(n) \in O(n * \log_2 n)$. This is encountered for fast sort algorithms, such as merge-sort and quick-sort.

Common complexities

- **Quadratic time**: $T(n) \in O(n^2)$. Empirically, ok with n in the hundreds but not with n in the millions.
- **Polynomial time**: $T(n) \in O(n^k)$. Empirically practical when k is not too large.
- **Exponential time**: $T(n) \in O(2^n), O(n!)$. Empirically usable only for small values of input.

Recurrences

What is a recurrence?

A recurrence is a mathematical formula defined recursively.

Example I - Node count of complete 3-ary tree

- A **recurrence** is a mathematical formula that is defined recursively.
- For example, let us consider the problem of determining the number $N(h)$ of nodes of a complete 3-ary tree of height h. We can observe that $N(h)$ can be described using the following recurrence:

$$\begin{cases} N(0) = 1 \\ N(h) = 3 * N(h-1) + 1, h \geq 1 \end{cases}$$

Example I - Node count of complete 3-ary tree

The explanation is given below:

- The number of nodes of a complete 3-ary tree of height 0 is 1.
- A complete 3-ary tree of height $h, h > 0$ consists of a root node and 3 copies of a 3-ary tree of height $h - 1$. If we solve the above recurrence, we obtain that:

$$N(h) = 3^h * N(0) + (1 + 3^1 + 3^2 + ... + 3^{h-1}) = \sum_{i=0}^{h} 3^i.$$

Example II - Recursive list summation

```
def recursiveSum(l):
    '''
    Compute the sum of numbers in a list
    l - input list
    return int, the sum of the numbers
    '''
    # base case
    if l == []:
        return 0
    # recursion step
    return l[0] + recursiveSum(l[1:])
```

- n represents list length
- In this case, the reccurence is:

$$T(n) = \begin{cases} 1, n = 0 \\ T(n-1) + 1, n > 0 \end{cases}$$

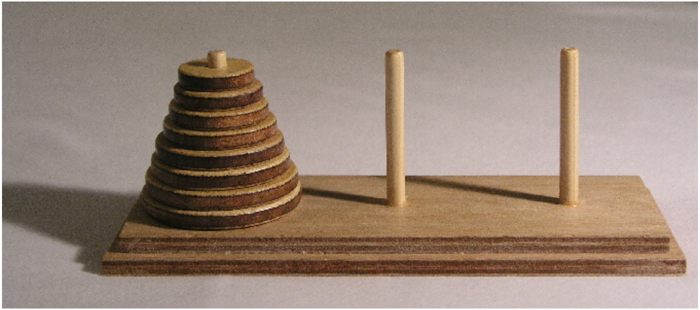
Example II - Recursive list summation

- Solving the reccurence:
  $$T(n) = \begin{cases} 1, n = 0 \\ T(n-1) + 1, n > 0 \end{cases}$$
- $T(n) = T(n-1) + 1$
- $T(n-1) = T(n-2) + 1$
- $T(n-2) = T(n-3) + 1 \Rightarrow T(n) = n + 1 \in \Theta(n)$

Example III - Tower of Hanoi

Legend says there is an Indian temple containing a large room with three posts and surrounded by 64 golden discs. Brahmin priests, acting out an ancient prophecy, are moving these discs since time immemorial, according to the rules of the Brahma. According to the legend, when the last move is completed, **the world will end**.

Example III - Tower of Hanoi



Example III - Tower of Hanoi

A mathematical game. Starts with three rods and a number of discs of increasing radius placed on one of them. The objective of the game is to move all the discs to another rod, observing the following rules:

- You can only move one disk at a time
- You can only move the uppermost disc from a rod
- You cannot place a larger disc on a smaller one

Example III - Tower of Hanoi

So ... are we safe (for now)? Let's study this:

- **Mathematically**
- Empirically

Example III - Tower of Hanoi

The idea of the algorithm (for **n** discs):

- Move **n-1** discs from source to intermediate stick
- Move the last disc to the destination stick
- Solve problem for **n-1** discs

Example III - Tower of Hanoi

```
def hanoi(n, x, y, z):
    '''
    n - number of disks on the x stick
    x - source stick
    y - destination stick
    z - intermediate stick
    '''
    if n==1:
        print("disk 1 from ",x, " to ",y)
        return
    hanoi(n-1, x, z, y)
    print("disk ",n, " from ",x," to ",y)
    hanoi(n-1, z, y, x)
```

- The recurrence is:

$$T(n) = \begin{cases} 1, n = 1 \\ 2T(n-1)+1, n > 1 \end{cases}$$

Example III - Tower of Hanoi

- Solving the recurrence:
  T(n) $= \begin{cases} 1, n = 1 \\ 2T(n-1) + 1, n > 1 \end{cases}$
- $T(n) = 2T(n-1) + 1,\ T(n-1) = 2T(n-2) + 1,$ $T(n-2) = 2T(n-3) + 1, \ldots,\ T(1) = T(0) + 1$
- $T(n) = 2T(n-1) + 1,\ 2T(n-1) = 2^2T(n-2) + 2,$ $2^2T(n-2) = 2^3T(n-3) + 2^2, \ldots, 2^{n-2}T(2) =$ $2^{n-1}T(1) + 2^{n-2}$
- We have $T(n) = 2^{n-1} + 2^0 + 2^1 + 2^2 + \ldots + 2^{n-2}$
- Therefore $T(n) = 2^n - 1 \in \Theta(2^n)$

Example III - Tower of Hanoi

So ... are we safe for now? Let's study this:

- Mathematically
- **Empirically**

Demo

Recursion

Examine the source code in **ex28_hanoi.py**

Space complexity

What is the space complexity of an algorithm?

The space complexity estimates the quantity of memory required by the algorithm to store the input data, the final results and the intermediate results. As the time complexity, the space complexity is also estimated using "O" and "Omega" notation.

- All the remarks from related to the asymptotic notations used in running time complexity analysis are valid for the space complexity, also.

Example I - List summation

```
def iterative_sum(l):
    '''
    Compute the sum of numbers in a list
    l - input list
    return int, the sum of the numbers
    '''
    res = 0
    for nr in l:
        rez += nr
    return rez
```

- We need memory to store the numbers, so $T(n) = n \in \Theta(n)$.

Example I - List summation

```
def recursive_sum(l):
    '''
    Compute the sum of numbers in a list
    l - input list
    return int, the sum of the numbers
    '''
    # base case
    if l == []:
        return 0
    # recursion step
    return l[0] + recursive_sum(l[1:])
```

- The recurrence is:

$$T(n) = \begin{cases} 0, n = 1 \\ T(n-1) + n - 1, n > 1 \end{cases}$$

Complexity overview

1. If there is Best/Worst case
   - Describe Best case
   - Compute complexity for Best Case
   - Describe Worst Case
   - Compute complexity for Worst case
   - Compute average complexity (if possible)
   - Compute overall complexity (if possible)
2. If Best $=$ Worst $=$ Average
   - Compute complexity

Searching. Sorting. Lambda expressions.

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Searching

- Data are available in the internal memory, as a sequence of records $(k_1, k_2, ..., k_n)$
- Search a record having a certain value for one of its fields, called the **search key**.
- If the search is successful, we have the position of the record in the given sequence.
- We approach the search problem's two possibilities separately:
  - Searching with unordered keys
  - Searching with ordered keys

Searching - unordered keys

Problem specification

- **Data**: $a, n, (k_i, i = 0, .., n-1)$, where $n \in \mathbb{N}, n \geq 0$.
- **Results**: p, where $(0 \leq p \leq n-1, a = k_p)$ or $p = -1$, if key is not found.

Searching - unordered keys


```python
def search_seq(el, l):
    '''
    Search for an element in list
    el - element
    l - list of elements
    Return the position of the element, -1 if not
        found
    '''
    poz = -1
    for i in range(0, len(l)):
        if el == l[i]:
            poz = i
    return poz
```

Computational complexity is $T(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$

Searching - unordered keys

```
def search_seq(el, l):
    '''
    Search for an element in list
    el - element
    l - list of elements
    Return the position of the element, -1 if not
        found
    '''
    i = 0
    while i<len(l) and el!=l[i]:
        i += 1
    if i<len(l):
        return i
    return -1
```

What is the difference between this and the previous version?

Searching - unordered keys

- Best case: the element is at the first position, $T(n) \in \Theta(1)$.
- Worst case: the element is in the n-1 position, $T(n) \in \Theta(n)$.
- Average case: if distributing the element uniformly, the loop can be executed $0, 1, .., n-1$ times, so $T(n) = \frac{1+2+..+n-1}{n} \in \Theta(n)$.
- Overall complexity is $O(n)$

Searching - ordered keys

Problem specification

- **Data**: $a, n, (k_i, i = 0, .., n-1)$, where $n \in \mathbb{N}, n \geq 0$, and $k_0 < k_1 < ... < k_{n-1}$;
- **Results**: p, where $(p = 0$ and $a \leq k_0)$ or $(p = n$ and $a > k_{n-1})$ or $(0 < p \leq n-1)$ and $(k_{p-1} < a \leq k_p)$.

Searching - ordered keys

```
def search_seq(el, l):
    '''
    Search for an element in list
    el - element
    l - list of ordered elements
    Return the position of the first occurrence, or
        position where element can be inserted
    '''
    if len(l) == 0: return 0
    poz = -1
    for i in range(0,len(l)):
        if el<=l[i]:
            poz = i
    if poz == -1: return len(l)
    return poz
```

Computational complexity is $T(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$

Searching - ordered keys


```
def search_succesor(el, l):
    '''
    Search for an element in list
    el - element
    l - list of ordered elements
    Return the position of the first occurrence, or
        position where element can be inserted
    '''
    if len(l)==0 or el<=l[0]:
        return 0
    if el>=l[-1]:
        return len(l)
    i = 0
    while i<len(l) and el>l[i]:
        i += 1
    return i
```

Searching - ordered keys

- Best case: the element is at the first position, $T(n) \in \Theta(1)$.
- Worst case: the element is in the n-1 position, $T(n) \in \Theta(n)$.
- Average case: if distributing the element uniformly, the loop can be executed $0, 1, .., n-1$ times, so $T(n) = \frac{1+2+..+n-1}{n} \in \Theta(n)$.
- Overabll complexity is $O(n)$

Searching algorithms

- *Sequential search*
  - Keys are successively examined
  - Keys may not be ordered
- *Binary search*
  - Uses the divide and conquer technique
  - Keys are ordered

Recursive binary-search algorithm


```python
def binary_search(key, data, left, right):
    '''
    Search for an element in an ordered list
    key - element to search
    left, right - bounds of the search
    Return insertion position of key that keeps list
        ordered
    '''
    if left >= right - 1:
        return right
    middle = (left + right) // 2
    if key < data[middle]:
        return binary_search(key, data, left, middle
            )
    else:
        return binary_search(key, data, middle,
            right)
print(binary_search(2000,data,0,len(data)))
```

Recursive binary-search function

```
def search(key, data):
    '''
    Search for an element in an ordered list
    key - element to search
    data - the list
    Return insertion position of key that keeps list
        ordered
    '''
    if len(data) == 0 or key < data[0]:
        return 0
    if key > data[-1]:
        return len(data)
    return binary_search(key, data, 0, len(data))
```

Binary-search recurrence

- The recurrence: $\mathrm{T(n)} = \begin{cases} 1, n = 1 \\ T(\frac{n}{2}) + 1, n > 1 \end{cases}$

Iterative binary-search function


```python
def binary_search(key, data):
    '''

    - specification -
    '''
    if len(data) == 0 or key < data[0]:
        return 0
    if key > data[-1]:
        return len(data)
    left = 0
    right = len(data)
    while right - left > 1:
        middle = (left + right) // 2
        if key <= data[middle]:
            right = middle
        else:
            left = middle
    return right
```

Search problem runtime complexity

Algorithm	Best case	Average	Worst case	Overall
Sequential	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Succesor	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$O(n)$
Binary-search	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	$O(\log_2 n)$

Searching in Python

Collections and search

Examine the source code in **ex29_search.py**

Iterators

Examine the source code in **ex30_iterators.py**

The sorting problem

Sorting

Rearrange a data collection in such a way that the elements of the collection verify a given order.

- **Internal sort** - data to be sorted are available in the internal memory
- **External sort** - data is available as a file (on external media)
- **In-place sort** - transforms the input data into the output, only using a small additional space. Its opposite is called out-of-place.
- **Sorting stability** - we say that sorting is stable when the original order of multiple records having the same key is preserved

Demo

Stable sort example

Examine the source code in **ex31_stableSort.py**

The sorting problem

- Elements of the data collection are called records
- A record is formed by one or more components, called fields
- A key K is associated to each record, and is usually one of the fields.
- We say that a collection of n records is:
  - Sorted in increasing order by the key K: if $K(i) \leq K(j)$ for $0 \leq i < j < n$
  - Sorted in decreasing order: if $K(i) \geq K(j)$ for $0 \leq i < j < n$

Internal sorting

Problem specification

- **Data**: n, K, where $K = (k_1, k_2, ..., k_n), k_i \in \mathbb{R}, i = 1, n$
- **Results**: K', where K' is a permutation of K, having sorted elements: $k'_1 \leq k'_2 \leq ... \leq k'_n$.

Sorting algorithms

A few algorithms that we will study:

- Selection sort
- Insertion sort
- Bubble sort
- Quick sort

Selection Sort

- Determine the element having the minimal key, and swap it with the first element.
- Resume the procedure for the remaining elements, until all elements have been considered.

Selection sort algorithm


```python
def selection_sort(data):
    for i in range(len(data)):
        min_index = i
        # Find smallest element in the rest of the
            list
        for j in range(i+1,len(data)):
            if data[j] < data[min_index]:
                min_index = j
        data[i], data[min_index] = data[min_index],
            data[i]
```

Selection sort - time complexity

- The total number of comparisons is
$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$
- Independent of the input data size, what are the best, average, worst-case computational complexities?

Selection sort - space complexity

- **In-place** algorithms. Algorithms that use a small (constant) quantity of additional memory.
- **Out-of-place** or not-in-space algorithms. Algorithms that use a non-constant quantity of extra-space.
- The additional memory required by selection sort is $O(1)$.
- Selection sort is an in-place sorting algorithm.

Direct selection sort


```python
def direct_selection_sort(data):
    for i in range(0, len(data) - 1):
        # Select the smallest element
        for j in range(i + 1, len(data)):
            if data[j] < data[i]:
                data[i], data[j] = data[j], data[i]
```

Direct selection sort

- Overall time complexity: $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$

Insertion Sort

- Traverse the elements.
- Insert the current element at the right position in the subsequence of already sorted elements.
- The sub-sequence containing the already processed elements is kept sorted, so that, at the end of the traversal, the whole sequence is sorted.

Insertion Sort - Algorithm


```python
def insert_sort(data):
    for i in range(1, len(data)):
        index = i - 1
        elem = data[i]
        # Insert into correct position
        while index >= 0 and elem < data[index]:
            data[index + 1] = data[index]
            index -= 1
        data[index + 1] = elem
```

Insertion Sort - time complexity

- Maximum number of iterations (worst case) happens if the initial array is sorted in a descending order:

$$T(n) = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Insertion Sort - time complexity

- Minimum number of iterations (best case) happens if the initial array is already sorted:

$$T(n) = \sum_{i=2}^{n} 1 = n - 1 \in \Theta(n)$$

Insertion Sort - Space complexity

- Time complexity - The overall time complexity of insertion sort is $O(n^2)$.
- Space complexity - The complexity of insertion sort is $\theta(1)$
- Insertion sort is an **in-place** sorting algorithm.

Bubble Sort

- Compares pairs of consecutive elements that are swapped if not in the expected order.
- The comparison process ends when all pairs of consecutive elements are in the expected order.

Bubble Sort - Algorithm


```python
def bubble_sort(data):
    done = False
    while not done:
        done = True
        for i in range(0, len(data) - 1):
            if data[i] > data[i+1]:
                data[i], data[i+1] = data[i+1], data
                    [i]
                done = False
```

Bubble Sort - Complexity

- **Best-case** running time complexity order is $\theta(n)$
- **Worst-case** running time complexity order is $\theta(n^2)$
- **Average** running-time complexity order is $\theta(n^2)$
- **Space complexity**, additional memory required is $\theta(1)$
- Bubble sort is an *in-place* sorting algorithm.

Quick Sort

Based on the *divide and conquer* technique

1. **Divide:** partition array into 2 sub-arrays such that elements in the lower part $\leq$ elements in the higher part.

Partitioning

Re-arrange the elements so that the element called pivot occupies the final position in the sub-sequence. If i is that position: $k_j \leq k_i \leq k_l$, for $Left \leq j < i < l \leq Right$

2. **Conquer:** recursively sort the 2 sub-arrays.
3. **Combine:** trivial since sorting is done in place.

Quick Sort - partitioning algorithm


```python
def partition(data, left, right):
    pivot = data[left]
    i = left
    j = right
    while i != j:
        # Find an element smaller than the pivot
        while data[j] >= pivot and i < j:
            j -= 1
        data[i] = data[j]
        # Find an element larger than the pivot
        while data[i] <= pivot and i < j:
            i += 1
        data[j] = data[i]
    # Place the pivot in position
    data[i] = pivot
    return i
```

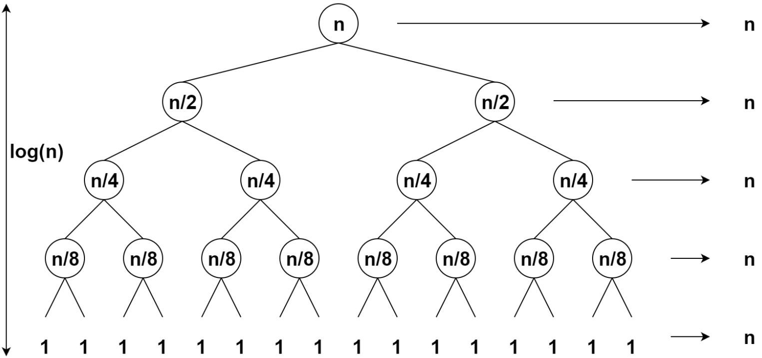
Quick Sort - algorithm


```python
def quick_sort(data, left, right):
    # Partition the list
    pos = partition(data, left, right)
    # Order left side
    if left < pos - 1:
        quick_sort(data, left, pos - 1)
    # Order right side
    if pos + 1 < right:
        quick_sort(data, pos + 1, right)
```

Quick Sort - time complexity

- The run time of quick-sort depends on the distribution of splits
- The partitioning function requires linear time
- **Best case**, the partitioning function splits the array evenly: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, $T(n) \in \Theta(n \log_2 n)$

Quick Sort - best partitioning

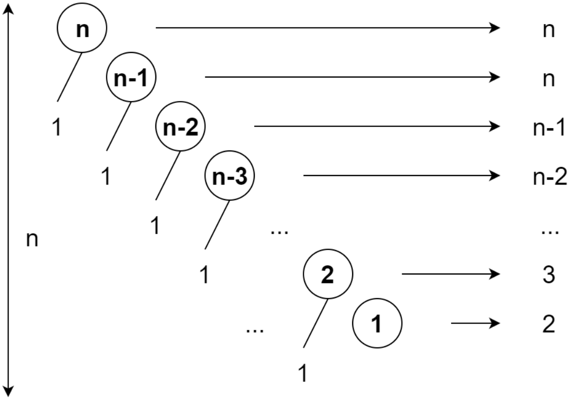


- We partition n elements $\log_2 n$ times, so $T(n) \in \Theta(n \log_2 n)$

Quick Sort - worst partitioning

- In the worst case, function Partition splits the array such that one side of the partition has only one element: $T(n) = T(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) \in \Theta(n^2)$

Quick Sort - Worst case



- Worst case partitioning appears when the input array is sorted or reverse sorted, so n elements are partitioned n times, $T(n) \in \Theta(n^2)$

Sorting runtime complexity

Algorithm	Worst case	Average
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$
Quick sort	$\Theta(n^2)$	$\Theta(n \log_2 n)$

Demo

Sorting

Examine the source code in **ex32_sort.py**

Lambda expressions

Lambda expressions

Small anonymous functions, that you define and use in the same place.

- Syntactically restricted to a single expression.
- Can reference variables from the containing scope (just like nested functions).
- They are *syntactic sugar* for a function definition.

Demo

Lambda Expressions

Examine the source code in **ex33_lambdas.py**

Problem solving methods

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Problem solving methods

- Strategies for solving more difficult problems, or general algorithms for solving certain types of problems
- A problem may be solved using more than one method - you have to select the most efficient one
- In order to apply one of the methods described here, the problem needs to satisfy certain criteria

Divide and conquer - steps

- **Divide** - divide the problem (instance) into smaller problems of the same structure
  - Divide the problem into two or more disjoint sub problems that can be resolved using the same algorithm
  - In many cases, there are more than one way of doing this
- **Conquer** - resolve the sub problems recursively
- **Combine** - combine the problems results

Remember

Typical problems for Divide & Conquer

Divide and conquer - general


```
def divide_conquer(data):
    if size(data) < a:
        # solve the problem directly
        # base case
        return rez
    # decompose data into d1, d2, ..., dk
    rez_1 = divide_conquer(d1)
    rez_1 = divide_conquer(d1)
    ...
    rez_k = divide_conquer(dk)
    # combine the results
    return combine(rez_1, rez_2, ..., rez_k)
```

Divide and conquer - general

When can divide & conquer be applied?

- A problem P on the data set D may be solved by solving the same problem P on other data sets, $d_1, d_2, ..., d_k$, of a size smaller than the size of D.

The running time for solving problems in this manner may be described using recurrences.

$$\mathrm{T(n)} = \begin{cases} \text{solve trivial problem, } n \text{ small enough} \\ k * T(\frac{n}{k}) + \text{divide time} + \text{combine time, otherwise} \end{cases}$$

Step 1 - Divide

- Simplest way: divide the data into 2 parts *(chip and conquer)*: data of size 1 and data of size n-1
- Example: Find the maximum

```
def find_max(data):
    '''
    Find the greatest element in the list
    input: data - list of elements
    output: maximum value
    '''
    if len(data) == 1:
        return data[0]
    # Divide into subproblems of sizes 1 and (n-1)
    max_val = find_max(data[1:])
    # Combine
    if max_val > data[0]:
        return max_val
    return data[0]
```

Step 1 - Divide

- Calculating time complexity
- The recurrence is: T(n) = $\begin{cases} 1, n = 1 \\ T(n-1) + 1, \text{otherwise} \end{cases}$

T(n)=T(n-1)+1,
T(n-1)=T(n-2)+1,
T(n-2)=T(n-3)+1, so
$T(n) = 1 + 1 + 1 + ... + 1 = n$, $T(n) \in \theta(n)$

Step 1 - Divide

- Divide into k subproblems of size n/k

```
def find_max(data):
    '''
    Find the greatest element in the list
    input: data - list of elements
    output: maximum value
    '''
    if len(data) == 1:
        return data[0]
    # Divide into two subproblems of size n/2
    mid = len(data) // 2
    max_left = find_max(data[:mid])
    max_right = find_max(data[mid:])
    # Combine
    return max(max_left, max_right)
```

Step 1 - Divide

- The recurrence is: T(n) $= \begin{cases} 1, n = 1 \\ 2 * T(\frac{n}{2}) + 1, \text{otherwise} \end{cases}$

$T(2^k) = 2 * T(2^{k-1}) + 1$
$2 * T(2^{k-1}) = 2^2 * T(2^{k-2}) + 2$
$2^2 * T(2^{k-2}) = 2^3 * T(2^{k-3}) + 2^2$
As we can divide the data into halves a number of **k** times, $n = 2^k$, then $k = log_2 n$ and $T(n) =$
$1 + 2^1 + 2^2 + 2^3 + ... + 2^k = \frac{2^{k+1}-1}{2-1} = 2^{k+1} - 1 = 2n \in \Theta(n)$

Divide and conquer - Example

- Compute x^k, where $k \geq 1$ is an integer
- Simple approach: $x^k = x * x * ... * x$, k-1 multiplications. Time complexity?
- Divide and conquer approach

$$x^k = \begin{cases} x^{\frac{k}{2}} * x^{\frac{k}{2}}, k \text{ is even} \\ x^{\frac{k}{2}} * x^{\frac{k}{2}} * x, k \text{ is odd} \end{cases}$$

Divide and conquer - Example

```python
def power(x, k):
    '''
    Calculate x to the power of k
    '''
    if k == 0:
        return 1
    if k == 1:
        return x
    # Divide
    aux = power(x, k // 2)
    # Conquer
    if k % 2 == 0:
        return aux ** 2
    else:
        return aux * aux * x
```

Divide and conquer - applications

- Binary-Search ($T(n) \in \theta(\log_2 n)$)
  - Divide - compute the middle of the list
  - Conquer - search on the left or for the right
  - Combine - nothing
- Quick-Sort ($T(n) \in \theta(n * \log_2 n)$)
  - Divide - partition the array into 2 subarrays
  - Conquer - sort the subarrays
  - Combine - nothing
- Merge-Sort ($T(n) \in \theta(n * \log_2 n)$)
  - Divide - divide the list into 2
  - Conquer - sort recursively the 2 list
  - Combine - merge the sorted lists

Backtracking

- Applicable to search problems
- Generates all the solutions, if they exist
- Does a depth-first search through all possibilities that can lead to a solution
- A general algorithm/technique - must be customized for each individual application.

Remember

Backtracking usually has exponential complexity

Generate and test

- **Problem**. Let n be a natural number. Print all permutations of numbers $1, 2, ..., n$.
- First solution - **generate & test** - generate all possible solutions and verify if they represent a solution

NB!

This is **not** backtracking

Generate and test - iterative

```
def perm3():
    for i in range(0,3):
        for j in range(0,3):
            for k in range(0,3):
                # A possible solution
                possible_sol = [i,j,k]
                if i!=j and j!=k and i !=k:
                    # Is solution
                    print possible_sol
```

NB!

This is **not** backtracking

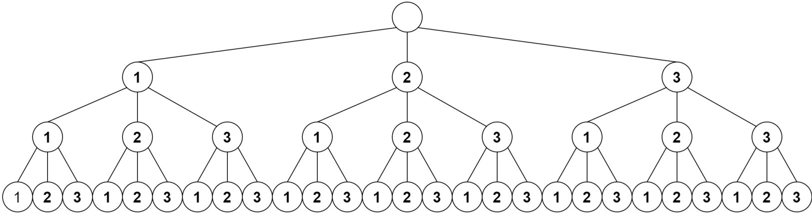
Generate and test - recursive

- Generate and test recursive - using recursion to generate all the possible list (candidate solutions)

```
def generate(x, DIM):
    if len(x) == DIM:
        print(x)
    if len(x) > DIM:
        return
    x.append(0)
    for i in range(0,DIM):
        x[-1] = i
        generate(x[:], DIM)
print(generate([], 3))
```

Generate and test

- Generate and test - all possible combinations



What have we learned?

- The total number of checked arrays is 3^3 , and in the general case n^n
- First the algorithm assigns values to all components of the array possible, and only afterwards checks whether the array is a permutation
- Implementation above is not general. Only works for n=3

Generate and test

- In general: if n is the depth of the tree (the number of variables in a solution) and assuming that each variable has k possible values, the number of nodes in the tree is k^n. This means that searching the entire tree leads to an exponential time complexity - $O(k^n)$

Generate and test

Possible improvements

- Do not construct a complete array in case it cannot lead to a correct solution.
- e.g - if the first component of the array is 1, then it is useless to assign other components the value 1
- Work with a potential array (a partial solution)
- When we expand the partial solution verify some conditions (conditions to continue) - so the array does not contains duplicates

Generate and test

Test candidates - print only solutions

```
def generate(x, DIM):
    if len(x) == DIM and is_set(x):
        print(x)
    if len(x) > DIM:
        return
    x.append(0)
    for i in range(0,DIM):
        x[-1] = i
        generate(x[:], DIM)
print(generate([], 3))
```

- We still generate all possible lists (e.g. starting with 0,0,...)
- We should not explore lists that contains duplicates (they cannot result in valid permutations)

- Recursive backtracking implementation for determining permutations

```
def backtracking(x, DIM):
    if len(x) == DIM:
        print(x)
    x.append(0)
    for i in range(0, DIM):
        x[-1] = i
        if is_set(x):
            # Continue only if we
            # can reach a solution
            backtracking(x[:], DIM)
print(backtracking([], 3))
```

Demo

Backtracking

Example illustrating exponential runtimes are in **ex34_backtracking.py**

Backtracking - Typical problem statements

- **Permutations** - Generate all permutations for a given natural number n
  - result: $x = (x_0, x_1, ..., x_n), x_i \in (0, 1, ..., n-1)$
  - is valid: $x_i \neq x_j$, for any $i \neq j$
- **n-Queen problem** - place n queens on a chess-like board such that no two queens are under reciprocal threat.
  - result: position of the queens on the chess board
  - is valid: no queens attack each other (not on the same rank, file or diagonal)

Backtracking - Theoretical support

Backtracking

The key to many problems that can be solved with backtracking lays in correctly and efficiently representing the elements of the search space

Backtracking - Theoretical support

- Solutions search space: $S = S_1 x S_2 x ... x S_n$
- x is the array which represents the solutions
- $x_{[1..k]}$ in $S_1 x S_2 x ... x S_k$ is the sub-array of solution candidates; it may or may not lead to a solution
- **consistent** - function to verify if a candidate can lead to a solution
- **solution** - function to check whether the array $x_{[1..k]}$ represents a solution of the problem.

Backtracking - recursive


```
def backtracking(x):
    # Add component to candidate
    x.append(0)
    for i in range(0, DIM):
        # Set current component
        x[-1] = i
        if consistent(x):
            if solution(x):
                solution_found(x)
            # Deal with next components
            backtracking(x[:])
```

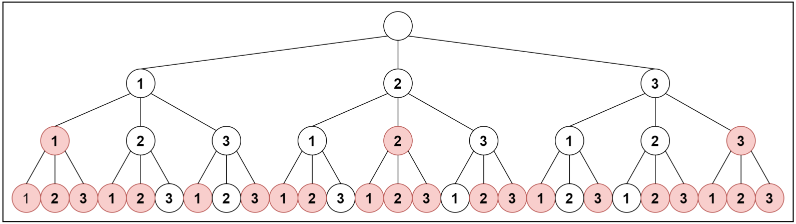
Backtracking - recursive

More generally, when solution components do not have the same domain:

```
def backtracking(x):
    # Add component to candidate
    el = first(x)
    x.append(el)
    while el != None:
        # Set current component
        x[-1] = el
        if consistent(x):
            if solution(x):
                solutionFound(x)
            # Deal with next components
            backtracking(x[:])
        el = next(x)
```

Backtracking

The nodes explored with backtracking for the permutations of 3



Backtracking

- How to use backtracking
- Represent the solution as a vector $X = (x_0, x_1, ..., x_n) \in S_0 x S_1 x ... x S_n$
- Define what a valid solution candidate is (conditions filter out candidates that will not conduct to a solution)
- Define condition for a candidate to be a solution

Greedy

- A strategy to solve optimization problems.
- Applicable where the global optima may be found by successive selections of local optima.
- Allows solving problems without returning to previous decisions.
- Useful in solving many practical problems that require the selection of a set of elements that satisfies certain conditions (properties) and realizes an optimum.
- Disadvantages: Short-sighted and non-recoverable.

Greedy - Sample Problems

The (fractional) knapsack problem

- A set of objects is given, characterized by usefulness and weight, and a knapsack able to support a total weight of W. We are required to place in the knapsack some of the objects, such that the total weight of the objects is not larger that the given value W, and the objects should be as useful as possible (the sum of the utility values is maximal).

The coins problem

- Let us consider that we have a sum M of money and coins (ex: 1, 5, 25) units (an unlimited number of coins). The problem is to establish a modality to pay the sum M using a minimum number of coins

Greedy - General case

- Let us consider the given set C of candidates to the solution of a given problem P. We are required to provide a subset B,($B \subseteq C$) to fulfill certain conditions (called internal conditions) and to maximize (minimize) a certain objective function.

Greedy - General case

- If a subset X fulfills the internal conditions we will say that the subset X is acceptable (possible).
- Some problems may have more acceptable solutions, and in such a case we are required to provide as good a solution as we may get, possibly even the best one, i.e. the solution that realizes the maximum (minimum) of a certain objective function.

Greedy - General case

In order for a problem to be solvable using the Greedy method, it should satisfy the following property:

- If B is an acceptable solution and $X \subseteq B$ then X is an acceptable solution as well.

Greedy - General case

- The Greedy algorithm finds the solution in an incremental way, by building acceptable solutions, extended continuously. At each step, the solution is extended with the best candidate from $C - B$ at that given moment. For this reason, this method is named greedy.
- The Greedy principle (strategy) is
  - Successively incorporate elements that realize the local optimum
  - No second thoughts are allowed on already made decisions with respect to past choices.

Greedy - General case

Assuming that θ (the empty set) is an acceptable solution, we will construct set B by initializing B with the empty set and successively adding elements from C.

- The choice of an element from C, with the purpose of enriching the acceptable solution B, is realized with the purpose of achieving an optimum for that particular moment, and this, by itself, does not generally guarantee the global optimum.

Greedy - General case

- If we have discovered a selection rule to help us reach the global optimum, then we may safely use the Greedy method.
- There are situations in which the completeness requirements (obtaining the optimal solution) are abandoned in order to determine an "almost" optimal solution, but in a shorter time.

Greedy - General case

Greedy technique

- Renounces the backtracking mechanism.
- Offers a single solution (unlike backtracking, that provides all the possible solutions of a problem).
- Provides polynomial running time.

Greedy - General code

```
def greedy(c):
    # The empty set is the candidate solution
    b = []
    while not solution(b) and c != []:
        # Select best candidate (local optimum)
        candidate = select_most_promising(c)
        c.remove(candidate)
        # If the candidate is acceptable, add it
        if acceptable(b + [candidate]):
            b.append(candidate)
        if solution(b):
            solution_found(b)
        # In case no solution
        return None
```

Greedy - General code (explanation)

- **c** - list of candidates
- **select_most_promising()** - returns the most promising candidate
- **acceptable()** - decides if the candidate solution can be extended
- **solution()** - verifies if a candidate represents a solution

Greedy - Essential elements

More generally, the required elements are:

1. A **candidate set**, from which a solution is created.
2. A **selection function**, which chooses the best candidate to be added to the solution.
3. A **feasibility function**, that is used to determine if a candidate can be used to contribute to a solution.
4. An **objective function**, which assigns a value to a solution, or a partial solution.
5. A **solution function**, which will indicate when we have discovered a complete solution.

Greedy - The coins problem

- Let us consider that we have a sum M of money and coins (ex: 1, 5, 25) units (an unlimited number of coins). The problem is to establish a modality to pay the sum M using a minimum number of coins.

Greedy - Coins problem solution

- **Candidate set** - the list of coin denominations (e.g. 1, 5, 10 bani).
- **Candidate solution** - a list of selected coins
- **Selection function** - choose the coin with the biggest value less than the remainder sum
- **Acceptable** - the total sum paid does not exceed the required sum
- **Solution function** - the paid sum is exactly the required sum

Demo

Greedy

The source code for the coins problem can be found in **ex35_coins.py**

Greedy - Remarks

- Before applying Greedy, is is required to prove that it provides the optimal solution. Often, the proof of applicability is non-trivial.
- Greedy leads to polynomial run time. Usually, if the cardinality of the set C of candidates is n, Greedy algorithms have $O(n^2)$ time complexity.

Greedy - Remarks

- There are a lot of problems that can be solved using Greedy, such as determining the shortest path between two nodes in an undirected or directed graph (Dijkstra's and Bellman-Kalaba's algorithms).
- There are problems for which Greedy algorithms do not provide optimal solution. In some cases, it is preferable to obtain a close to optimal solution in polynomial time, instead of the optimal solution in exponential time - these are known as *heuristic algorithms*.

Greedy example - Interval scheduling

- You want to schedule a number of jobs on a computer
- Jobs have the same value, and are characterized by their start and finish times, namely (s_i, f_i) (the start and finish times for job "i")
- Run as many jobs as possible, making sure no two jobs overlap

Greedy example - Interval scheduling

- A Greedy implementation will directly select the next job to schedule, using some criteria
- The crucial question for solving the problem - **how to determine the correct criteria?**

Source

This example, like many others can be found in "Algorithm Design", by Kleinberg & Tardos[1]

[1] https://www.cs.princeton.edu/~wayne/kleinberg-tardos/

Greedy example - Interval scheduling

Some ideas for selecting the next job

- **The job that starts earliest** - the idea being that you start using the computer as soon as possible
- **The shortest job** - the idea is to fit in as many jobs as possible
- **The job that overlaps the smallest number of jobs remaining** - we keep our options open
- **The job that finishes earliest** - we free up the computer as soon as possible

Greedy example - Interval scheduling

Ideas that **don't** work:

- The job that starts earliest
- The shortest job
- The job that overlaps the smallest number of jobs remaining

Greedy example - Interval scheduling

Ideas that **don't** work:

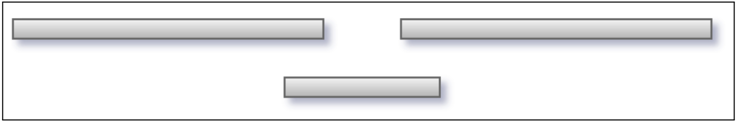
- The job that starts earliest

- The shortest job
- The job that overlaps the smallest number of jobs remaining

Greedy example - Interval scheduling

Ideas that **don't** work:

- The job that starts earliest
- The shortest job

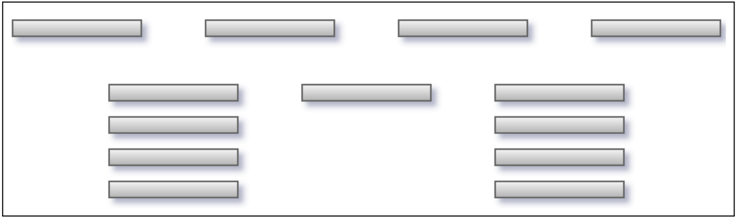


- The job that overlaps the smallest number of jobs remaining

Greedy example - Interval scheduling

Ideas that **don't** work:

- The job that starts earliest
- The shortest job
- The job that overlaps the smallest number of jobs remaining



Greedy example - Interval scheduling

An idea that **works** - Start the job that finishes earliest

```
S = set of jobs
while S is not empty:
    next_job = the job that has the
    soonest finishing time
    add next_job to solution
    remove from S jobs that overlap q
```

NB!

Proving this is done using mathematical induction, but it is beyond our scope.

Further learning resources

Further resources

`http://www.cs.princeton.edu/~wayne/kleinberg-tardos/`

Dynamic programming

Applicable in solving optimality problems where:

- The solution is the result of a sequence of decisions, $d_1, d_2, ..., d_n$.
- The principle of optimality holds.

Dynamic programming:

- Usually leads to polynomial run time.
- Always provides the optimal solution (unlike Greedy).
- Like divide & conquer, it combines the solutions from sub-problems, but also stores intermediate results that are reused multiple times.
- Visualize it as an optimization to applying recursion.

Dynamic programming

- We consider states $s_0, s_1, ..., s_{n-1}, s_n$, where s_0 is the initial state, and s_n is the final state, obtained by successively applying the sequence of decisions $d_1, d_2, ..., d_n$ (using the decision d_i we pass from state $s_i - 1$ to state s_i , for i=1,n):

$$s_0 \xrightarrow{d_1} s_1 \xrightarrow{d_2} s_2 \rightarrow \overset{...}{...} \rightarrow s_{n-1} \xrightarrow{d_n} s_n$$

Dynamic programming

The method makes use of three main concepts:

- Optimality principle
- Overlapping sub problems
- Memoization.

Dynamic programming - Optimality principle

If $d_1, d_2, ..., d_n$ is a sequence of decisions that optimally leads a system from the state s_0 to s_n, then one of the following conditions has to be satisfied:

- $d_k, d_{k+1}, ..., d_n$ is a sequence of decisions that optimally leads the system from state s_{k-1} to state s_n, $\forall k, 1 \leq k \leq n$ **(forward variant)**
- $d_1, d_2, ..., d_k$ is a sequence of decisions that optimally leads the system from state s_0 to the state s_k, $\forall k, 1 \leq k \leq n$ **(backward variant)**
- $d_{k+1}, d_{k+2}, ..., d_n$ and $d_1, d_2, ..., d_k$ are sequences of decisions that optimally lead the system from state s_k to state s_n and, respectively, from state s_0 to state s_k, $\forall k, 1 \leq k \leq n$ **(mixed variant)**

Dynamic programming - Overlapping sub-problems, memoization

- **Overlapping sub-problems** - A problem is said to have overlapping sub-problems if it can be broken down into sub-problems which are reused multiple times
- **Memoization** - Store the solutions to the sub-problems for later use

Dynamic programming - Requirements

- The principle of optimality (in one of the forward, backward or mixed forms) is proven.
- The structure of the optimal solution is defined.
- Based on the principle of optimality, the value of the optimal solution is recursively defined. This means that recurrent relations, indicating the way to obtain the general optimum from partial optima, are defined.
- The value of the optimal solution is computed in a bottom-up manner, starting from the smallest cases for which the value of the solution is known.

Dynamic programming - Longest increasing subsequence

Problem statement

Let us consider the list $a_1, a_2, ..., a_n$. Determine the longest increasing subsequence $a_{i_1}, a_{i_2}, ..., a_{i_s}$ of list a.
e.g. given sequence [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15], a longest increasing subsequence is [0, 2, 6, 9, 11, 15].

- **Optimality principle** - verified in its forward variant.
- **Structure** of the optimal solution - we construct sequences: $L =< L_1, L_2, ..., L_n >$ so that for each $1 \leq i \leq n$ we have that L_i is the length of the longest increasing subsequence ending at i.
- The recursive definition for the value of the optimal solution:
  - $L_i = max\{1 + L_j | A_j$, so that $A_j \leq A_i\}, \forall j = i-1, n-2, ..., 1$

Demo

Longest Increasing Subsequence

Examine the source code in **ex36_longestIncreasingSubsequence.py**

Maximum subarray sum

Problem statement

Calculate the maximum sum of a subarray consisting of consecutive elements. e.g. for subarray **[-2, -5, 6, -2, -3, 1, 5, -6]** the maximum sum is 7 (6-2-3+1+5, as the numbers must be consecutive)

Maximum subarray sum

This is a great problem, because there are several implementations.

1. Naive implementation(s)
2. Divide & conquer
3. Dynamic programming

Maximum subarray sum

Naive implementation(s)

1. Uses 3 loops; one for interval start, one for interval end, one to calculate partial sum. At every step, compare obtained sum with previous maximum.
2. Uses 2 loops; final loop of previous implementation can be eliminated by calculating partial sums using the second loop

Maximum subarray sum

Divide & Conquer implementation

1. **Divide** - The subarray we are searching for can be in one of only 3 places:
   1. Contained in left side of array.
   2. Contained in right side of array.
   3. Subarray includes middle element of array.
2. **Conquer** - Calculate alternatives using 2 recursions and one $O(n)$ algorithm. Final complexity is $O(n * log_2(n))$
3. **Combine** - Return maximum value.

Maximum subarray sum

Dynamic programming implementation

- We iterate over the array once (so $O(n)$ complexity).
- For each index, we calculate the maximum array ending at that position. If the sum is a new maximum, we record it.

Implementation

The source code for all implementations can be found in **ex37_maximumSubarraySum.py**

Dynamic programming - 0-1 knapsack problem

Problem statement

You have a collection of items, each having its own weight and utility. You have a knapsack with capacity W. Which of the items can you pack in order to maximize their utility. You cannot break up items (0-1 property), and you cannot pack the same item more than once (bounded version)

Demo

0-1 knapsack problem

Source code in **ex38_01knapsack.py**

Dynamic programming - Egg dropping puzzle

Problem statement

Suppose you have a number of **n** eggs and a building having **k** floors. Using a minimum number of drops, determine the lowest floor from which dropping an egg breaks it.

Dynamic programming - Egg dropping puzzle

Rules:

- All eggs are equivalent.
- An egg that survives a drop is unharmed and reusable.
- A broken egg cannot be reused.
- If an egg breaks when dropped from a given floor, it will also break when dropped from a higher floor.
- If an egg does not break when released from a given floor, it can be safely dropped from a lower floor.
- You cannot assume that dropping eggs from the first floor is safe, nor that dropping from the last floor is not.

Dynamic programming - Egg dropping puzzle

The problem is about finding the correct strategy to improve the worst case outcome - make sure that the **maximum** number of drops is kept to a minimum (a.k.a minimization of maximum regret).

Let's start with the simplest case...

- Building has **k** floors, and we have **n=1** egg.

Dynamic programming - Egg dropping puzzle

What do we do in the **n=1** case?

- Drop the egg at each floor until it breaks or you've reached the top, starting from first (ground) floor.
- In this case, the maximum number of drops is equal to k, the number of floors the building has.

Dynamic programming - Egg dropping puzzle

So, how about if we have more eggs?

- Building has **k** floors, but now we have **n=2** eggs.

Discussion

How do we keep the maximum number of drops to a minimum?

Dynamic programming - Egg dropping puzzle

Possible strategies for the n=2 case...

1. The **n=1** case was basically linear search, so we can try binary search with the first egg (drop it from the mid-level floor).
2. How about dropping from every 20th floor, starting from ground level?

Dynamic programming - Egg dropping puzzle

Let's try to describe the situation in a structured way...

- Imagine we drop the first egg at a given floor **m**.
- If it breaks, we have a maximum of **(m-1)** drops, starting from ground.
- If it does not break, we increase by **(m-1)** floors, as we have one less drop.
- Following the same logic, at each step we decrease the number of floors by 1.

So, if we have **n=2** eggs and **k=100** floors?

- We solve $m + (m-1) + (m-2) + (m-3) + (m-4) + ... + 1 \geq 100$
- Solution is between 13 and 14, which we round to **14**. First drop is from floor 14

Dynamic programming - Egg dropping puzzle

Egg drops for **n=2** eggs and **k=100** floors...

Drop #	1	2	3	4	5	6	7	8	9	10	11	12
Floor	14	27	39	50	60	69	77	84	90	95	99	100

Dynamic programming - Egg dropping puzzle

Now for the general case - building has **k** floors, and we have **n** eggs.

- **Optimal substructure** - Dropping an egg from floor **x** might result in two cases (subproblems):
  1. **Egg breaks** - Problem is reduced to one with **x-1** floors and one fewer egg.
  2. **Egg is ok** - Problem is reduced to one with **k-x** floors and same number of eggs.

Dynamic programming - Egg dropping puzzle

- **Overlapping subproblems** - Let's create function **eggDrop(n, k)**, where **n** is the number of eggs and **k** the number of floors.
- **eggDrop(n, k)** $=1+\min\{\max($eggDrop(n - 1, x - 1), eggDrop(n, k - x)), with $1 \leq x \leq k\}$

Demo

Egg dropping puzzle

Full source code available in **ex39_eggDroppingPuzzle.py**

Dynamic programming vs. Greedy - Remarks

- Both techniques are applied in optimization problems
- Greedy is applicable to problems for which the general optimum is obtained from partial (local) optima
- Dynamic programming is applicable to problems in which the general optimum implies partial optima.