**MY470 – Computer Programming:**

**Lectures & Seminars**

* Introduce fundamentals of computer programming,
* Cover foundations of computer languages, object-oriented programming, and algorithms,
* learn how to design, write, and debug computer programs and how to evaluate algorithms,
* practice on applications from computational social science and social data science.

Dr. Milena Tsvetkova,

Dr. Patrick Gildersleve,

Alexander Soldatkin (GTA)

Admin – Lucia Pedrioli, [methodology.admin@lse.ac.uk](mailto:methodology.admin@lse.ac.uk)

Admin:

* Lecture Monday
* Email Milena for urgent problem
* Email methodology.admin for extension
* Post & answer clarifying questions on Moodle
* Office hours for questions about programming and research
* 10 2hr lectures
* 10 1.5hr seminars

Software:

* Python (Anaconda distribution) to learn basic concepts in computer science
* R to experience another common language
* Jupyter web app and RStudio to write code
* GitHub to share course documents and assignments

**Assessment**

1. 8 problem sets (50%),
   1. Due Monday 12pm (grades & comments by following Friday)
   2. Problem set each week.
   3. First problem set is formative, then they’re all summative. Uploaded to Github.
   4. Clone assignment from private location to local machine, perform task & push back to cloud. Assignment is accessed from our personal Github, downloaded at 12pm every Monday, and then pushed back to us with comments and marks.
2. Take-home exam (50%)
   1. Substantive Python project requiring demonstration of concepts and skills learnt on course.
   2. Due at 12pm on Monday, 16 January 2023.
   3. Receive data set & questions (open-ended).
3. Assessment criteria
   1. Code runs and does what it is expected to
   2. Capture essence of task with minimal resources and it does the job, simple.
   3. The code is written using the concepts, paradigms and best practices covered in the course.
      1. Legibility – can be read and understood.
      2. Modularity – Complex problem is broken up effectively and fits together
      3. Optimization - Evaluating code algorithmically (which approach to solving problem is most efficient), and how can it be written to run faster.

**Lecture: Week 1 – What is Computation?**

Overview

* Computational thinking and algorithms

**Computational thinking**

Thought process involved in formulating a problem and expressing solution in way that a computer can carry out.

* Conceptualizing problems at multiple levels of abstraction
* Creativity in problem-solving
* Combines mathematical and engineering thinking – dictated by the constraints of physical computing devices.

**Algorithms**

* A well-defined computational procedure that takes values as input and produces values as output.
* Simple steps, control flow & stopping rule.

**Computers**

* Performs calculations & stores the results. That’s it.

**Programming Languages**

* Formal language used to specify a set of instructions for a computer to execute.
  + **Primitive constructs**
  + **Syntax**
  + **Static semantics**
  + **Semantics**

**Markup languages vs. Programming languages**

* Markup: HTML, TeX, XML, Markdown
* Programming: C, Java, Python, R
* Markup often uses program, e.g., browser, to execute code.
* Programming uses computer hardware to execute
* Markup presents but doesn’t transform or manipulate information.

**Primitive Constructs in programming languages**

1. Literals, e.g., ‘470,’ 29L, “Hello”
2. & Infix operators, e.g., / + \* -

**Syntax in programming languages**

1. Rules for forming strings of characters and symbols
2. Programming languages have strict syntax, e.g. mutate(), geom\_bar(), train(), randomForest(), as.tibble()

**Static Semantics in programming languages**

1. Rules for forming meaningful syntactically valid strings
2. E.g., don’t divide numbers by strings.

**Semantics in programming languages**

1. The meaning associated with a syntactically correct string that has no static semantics errors.
2. Programming languages have simple semantics – statements have only one meaning (not true).

**Types of programming languages**

1. Low-level vs. high-level (speak to hardware or layers of parsing)
2. General vs. application-targeted
3. Interpreted vs. compiled (On-the-spot execution vs. need for compiling into machine code).

Python & R = high-level, general & interpreted.

For interpreted languages, statements are executed by ‘shell.’

**Objects, Data Types, and Expressions**

1. Programs manipulate objects
2. Objects have types
   1. Scalar – indivisible (like an integer)
   2. Non-scalar – with internal structures (like a string, which has substrings). These can be concatenated.
3. Expressions combine objects and operators

**Debugging**

1. Google the error on Stackoverflow
2. Use print() systematically.
   1. Compare input in successful and failing runs
   2. Formulate a hypothesis
   3. Design an experiment to test the hypothesis
   4. Keep record of experiment
   5. Repeat.
3. After debugging for hours – stop. Comment your code, explain it to someone else.

**Version Control in Github**

* Code hosting platform for version control & collab.
* Based on Git
  + Version control system for tracking changes in computer files and coordinating work on those files among multiple people.
  + Created in 2005 by Linus Torvalds
* Largest host of source code in world
* Bought by MS in 2018
* Repo, Clone, Branch, Commit, Pull Request, Merge.
* Difference between Forking & Branching.
* Forking creates another copy on the cloud, but it gets more complex to pull changes.
* Browser, command line or GitHub desktop

**Readings: Week 1**

**Introduction to Computation & Programming Using Python**

Computer does two things – calculations and storage of values.

Type() can be used to find out the type of an object.

The symbol >>> is a **shell prompt** that indicates that the interpreter is expectin the user to type some Python code into the shell.

>>> 3 + 2

5

>>> 3.0 + 2.0

5.0

>>> 3 != 2

True

* // denotes integer division. So 6//2 is 3, but 6//4 is 1.
* % is the remainder from integer division, so 6%4 is 2.
* \*\* raises X to the power of Y, so 6\*\*3 is 216, and an integer. However, 6.0\*\*\*3.0 is 216.0, and a float.
* For Boolean expressions, use and, or and not.
* a and b is True is both a and b are True, and False otherwise.
* A or B is True is at least a or b is True, and False if not.
* not a is True if a is False, and False if a is True

In Python, a variable is just a name. An assignment statement (=) associates the name on left to object denoted on the right.

There are a bunch of reserved words that you can’t assign to your variables.

Python allow multiple assignments, the statement: x, y = 2, 3 binds x to 2 and y to 3. All of the expressions on the right-hand side of the assignment are evaluated before any bindings are changed, so:

x, y = 2, 3

x, y = y, x

print(‘x =’, x)

print(‘y =’, y)

will print:

x = 3

y = 2

**Computational Thinking – Jeanette M. Wing**

**Seminar: Week 1**

* **Ensure you have GitHub desktop working**
* **Ensure command prompt is working**
* **Jupyter notebook files have .ipynb extension**
* **Upon downloading Anaconda, you download Python, Visual Studio, Spyder, PyCharm, Jupyter notebooks.**
  + **Jupyter is a server running on my local machine and requires to be shut down, so it’s still using computer resources.**
  + **Click ‘quit’ in top right of browser to do so.**
  + **Close individual notebook with file > close & halt.**
  + **Can also use .py files but we’ll mainly be using Jupyter notebooks**

**Lecture: Week 2 – Data Types in Python**

Scalars:

* Int, float, bool, None
* Operators: arithmetic, Boolean, comparison, assignment, membership

Non-scalars:

* Methods
* Ordered vs. unordered non-scalars
* Mutable vs immutable non-scalars

type(35) = int

type(35.234) = float

type(None) = NoneType

type(‘hjkajda’) = str

**convert between scalar data types**

* Use the name of a type to convert values to that type
* Float(‘1235.23’) = 1235.23
* Int(19140.7007) = 19140 (doesn’t round up)

Modulus is remainder from division

Operators:

* Arithmetic: + - \* / % // \*\*
* Boolean: and or not
* Comparison: == != < > =< =>
* Assignment: =
* Membership: in. ‘c’ in [‘a’,’b’] = FALSE, ‘c’ in [‘a’,’b’,’c’] = TRUE

If we want to ASSIGN a variable, then use single = sign. If we want to **compare** whether two variables are the same, we use a double-equals sign ==.

**Non-Scalar Data Types**

* List: a mutable ordered sequence of values
  + Use [] to define list (square brackets).
* Tuple: an immutable ordered sequence of values
  + Tup = tuple()
* String: an immutable ordered sequence of characters
* Set: a mutable unordered collection of unique values
  + Need {} (curly brackets) and returns unique elements.
  + st = {2,3,7,7,2,1,3}
  + st = {1,2,3,7}
* Dictionary: a set of key/value pairs.
  + Need {} (curly brackets) and gives key and value pairs
  + dict = {‘key1’:1, ‘key2’:2, ‘key3’:2300}
  + Can find elements by key associated with them.

**convert between non-scalar data types**

* Use the name of a type to convert values to that type
* Tuple(1,2,3)

What does mutable mean? It means it can be changed and manipulated! (mutate)

Because non-scalar objects have structure & subsets through indexing, we can evaluate the length of an object with len().

‘’’ Triple quoted strings may span multiple lines –

All associated whitespace will be included

In the string literal.’’’

**Objects have methods associated with them**

* Python object (full stop) method (arguments) = object.method()
* Object.method()
  + Use the period . to link the method to the object.
  + So, if string\_1 = “Hello”, then:
  + string\_1.upper() converts all of the string to upper case.
  + String\_1.replace(substring\_1, substring\_2)
  + String\_1.strip(substring\_1) removes trailing whitespace characters from the beginning and end of a string. This is useful when reading in from a file.
  + String\_1.split(substring\_1) – splits the string into a list, taking the argument as the place to split. By default, this splits on whitespace.
  + String\_1.join() – joins string together by replacing whitespace with hyphens.

Set methods

* + S1.union(S2),S1 | S2 – elements in S1 or S2, or both
  + S1.intersection(S2), S1 & S2 – eleemtns in both S1 & S2
  + S1.difference(S2),

Dictionary Operations: Indexing

* Can search key in index and pull the value associated with it.

**Indexing**

* Lists, tuples and strings are indexed by numbers
* ls[0] # returns first element
* ls[0:2] # last value is NOT included, so need to go 1 wider
* ls[3:5] # shows 3rd and 4th element
* use element[start:end:step] to get sub-sequence starting from index *start*, in steps of *step,* to *end-1.*

**Why use tuples?**

* They are immutable, they can’t be changed
* They use less memory than lists
* They can be used as dictionary keys; lists can’t.
* Good for things that are static, like coordinates etc.

**Mutability**

* Immutable: str, tuple, all scalars
* Does not double data object when amending or overriding object, like R does (or immutable objects).
* Mutable types: list, set, dict
* Convenient, especially In context of lists
  + L.append(e)
  + L.insert(i,e)
  + L.remove(e)
  + L.extend(L1)
  + L.pop(i)
  + L.sort()
  + L.reverse()

**Readings: Week 2**

**Strings & Input function (ch 2.3)**

The operator + is said to be overloaded: it has different meanings depending upon the types of the objects to which it is applied. It means addition when applied to two numbers and concatenation when applied to two strings. The operator \* is also overloaded. It means multiplication for numbers but when applied to an integer and a string, it is a repetition operator. The expression number\*string evaluates to a string with n repeats of S. 2\*’John’ = ‘JohnJohn.’ ‘ha’\*3 = ‘hahaha’

Type checking exists to turn mistakes and careless errors turning into execution-halting commands rather than by letting the programming run in a misleading way. Type checking is not as strong in Python.

Strings share the following operations with all sequence types:

* The length of a string can be found using the len() function. len(‘abc’) = 3
* Indexing can be used to extract individual characters from a string. In Python, all indexing is zero-based. So ‘abc’[0] into the interpreter will cause it to display the string ‘a.’ Typing ‘abc’[3] will produce error: IndexError: string index out of range.
* Negative numbers are used to index from the end of a string. E.g. ‘abc’[-1] is ‘c’
* Slicing is used to extract substrings of arbitrary length. If s is a string, the expression s[start:end] denotes the substring of s that starts at index start and ends at index end-1. So, ‘abc’[1:3] = ‘bc.’
* Why does it end at index end-1 rather than end? So that expressions such as ‘abc’[0:len(‘abc’)] have the value one might expect (due to zero-based indexing). So, ‘abc’[:] = ‘abc’[0:len(‘abc’)]

**Input function**

* Gets input directly from a user.
* Takes string as an argument and displays it as a prompt in the shell.

Python now uses Unicode for character encoding. You can tell Python which coding to use by typing:

* # \_\*\_ coding: *encoding name \_\*\_*
* # \_\*\_ coding: *utf-8 \_\*\_*

**Tuples (ch 5.1)**

Difference between string & tuple is that tuples don’t have to just contain characters.

Literals of type **tuple** can be written like:

T1 = (1,”two”,3) # comma-separated list in parentheses.

T1 = (1) is the long way of saying integer 1.

T1 = (1,) denotes singleton tuple containing 1.

Repetition can be used on tuples, so 3\*(‘a’,2) = (‘a’,2,’ ‘a’,2, ‘a’,2)

Tuples can be concatenated, indexed and sliced:

T1 = (1,”two”,3)

T2 = (T1, 3.25)

Print(T2) = ((1, “two”, 3), 3.25)

Print((T1+T2)) = (1, “two”, 3, (1, “two”, 3), 3.25)

Print((T1 + T2)[3]) = (1, “two”, 3) -----4th element

Print((T1 + T2)[2:5]) = (3, (1,’two’,3),3.25)-------3rd, 4th & 5th element

Notice that T1 within T2 is ONE ELEMENT.

x, y = (3,4) binds x to 3 and y to 4.

A,b,c = ‘xyz’ will bind a to ‘x,’ b to ‘y,’ and c to ‘z.’

**Ranges**

* Like strings and tuples, ranges are immutable. The *range* function returns an object of type *range.* It takes 3 integer arguments: start, stop, step.
* If step is positive, the last element is the largest integer that start + i\*step is less than stop.
* If step is negative, the last element is the smallest integer start + i\*step greater than stop.
* Operations that work on tuples also work on ranges, except concatenation and repetition. E.G., range(10)[2:6][2] evaluates 4
* When the == operator is used to compare objects of type *range,* it returns TRUE if the two ranges represent the same sequence of integers. They must be the same integers in the same order.
* Defined by start/stop/step so very small amount of space taken.

**Lists and Mutability**

* Use square brackets to define lists. Empty list written as []
* Occasionally, this combination of square brackets & using [] to index and slice can cause some confusion.
* [1,2,3,4][1:3][1] evaluates to 3. (subset 2,3,4 from 1:3. Then, [1] selects second element).
* Lists differ from tuples in that they are **MUTABLE.**
* Remember: In Python, a variable is merely a name i.e., a label that can be attached to an object.”
  + Techs = ['MIT','Caltech']
  + Ivys = ['Harvard','Yale','Brown']
  + Univs = [Techs, Ivys]
  + Univs\_alt = [['MIT','Caltech'], ['Harvard','Yale','Brown']]
  + print(Univs == Univs\_alt)
  + # IT APPEARS AS IF UNIVS & UNIVS\_ALT are bound to the same value. They are not. We can verify this with id
  + print(id(Univs)) # 140649871405952
  + print(id(Univs\_alt)) # 140650126787136
  + # Elements of Univs are actually lists themselves.

**Aliasing**

* Why does this matter? Who cares?
  + Techs.append(‘RPI’) adds RPI to techs, and hence Univs. The append method has a side effect. Rather than creating a new list, it mutates the existing list.
  + This is called **aliasing.** there are two distinct paths to the same list object. One is through the variable Techs and the other through the first element of the list object to which Univs is bound. This can be both convenient and treacherous.

**Cloning**

* It’s usually prudent to avoid mutating a list over which one is iterating.
* Perform cloning with [:]
* So, list\_1[:] just creates a clone of list 1 which can be iterated on
* List(list\_1) returns a copy of the list too.

**List comprehension**

L = [x\*\*2 for x in range(1,7)]

Print(L)

Will print [1,4,9,16,25,36]

The **for** clause in a list comprehension can be followed by one or more if and for statements that are applied to the values produced by the **for** clause. These additional clauses modify the sequence of values generated by the first **for** clause and produce a new sequence of values, to which the operation associate dwith the comprehension is applied. So:

Mixed = [1,2,’a’,3,4.0]

Print([x\*\*2 for x in mixed if type(x) == int])

[1,4,9]

**Strings, Tuples, Ranges, and Lists (ch. 5.5)**

We’ve looked at four different sequence types: str, tuple, range and list. They are similar in that they can be operated on similarly.

* Seq[i] returns the ith element in the sequence.
* Len(seq) returns the length of the sequence
* Seq1 + seq2 concatenates both (not available for ranges)
* N\*seq returns a sequence that repeats seq n times (not available for ranges)
* Seq[start:end] returns a slice of seq.
* E in seq is TRUE if e is contained in the sequence and FALSE otherwise.
* E not in seq is opposite to above
* For e in seq iterates over the elements of the sequence.

Python users tend to use lists far more often than tuples. Since they are mutable, they can be constructed incrementally during computation. However, because tuples are immutable, aliasing is never a worry. They can also (due to their immutability) be used as keys in dictionaries.

**Dictionaries**

* Object of type dict are like lists except that we index them using keys. They are enclosed in curly brackets.
* Mutable, can add entry by writing: monthNumbers[‘June’] = 6 or monthNumbers[‘May] = ‘V’
* We can use for translation etc.
* The method *keys* returns an object of type dict\_keys, which is an example of a view object. A view object is dynamic in that if the object with which it is associated changes, the change is visible through the view object.
* Only immutable objects are hashable, so can be used as keys. This makes tuples brilliant for this. Mutable objects are NOT hashable and hence cannot be keys for dictionaries.
* Methods
  + Len(d) returns number of items in d
  + D.keys() returns a view of the keys in d.
  + D.values() returns a view of values in d
  + K in d returns TRUE if k is in d
  + D[k] returns the item in d with key k.
  + D.get(k, v) returns d[k] if k is in d, and v otherwise.
  + D[k] = v associates the value v with the key k in d. If there is already a value associated with k, it is replaced.
  + Del d[k] removes the key k for d.
  + For k in d iterates over the keys in d.

**Seminar: Week 2 – Working with Strings & Lists in Python (in Jupyter)**

Assignment 2 due on Monday 10th

**Lecture: Week 3 –**

**Overview**

* What is control flow?
  + Conditional statements
  + Iteration
  + List comprehensions
* Examples
  + Exhaustive enumeration
  + Bisection search
  + Newton-Raphson algorithm
* Iteration and efficiency
* Control flow is the order in which statements are executed or evaluated
* In Python, there are three main categories of control flow:
  + **Branches** (conditional statements) – execute only if some condition is met
  + **Loops** (iteration) – execute repeatedly
  + Function calls – execute a set of distant statements and return back to the control flow

x **=** input('How old are you? ')

**if** int(x) **>=** 25:

print("Ah, I see, you are a mature student.")

## Indentation in Python Code

* Indentation is semantically meaningful in Python
* You can use [tabs or spaces](https://www.youtube.com/watch?v=SsoOG6ZeyUI)
* Obviously(!), tabs are preferable
* However, it does not really matter in Jupyter as Jupyter converts tabs to spaces by default

Graphical user interface, text, application

Description automatically generated

Else refers to **all other cases.** If we want to break up ‘everything else,’ then use else if.

Graphical user interface, text, application

Description automatically generated

**Iteration**

Diagram

Description automatically generated

Text

Description automatically generated

**Iteration: *while* with decrementing function**

The decrementing function is a function that maps variables to an integer that is initially non-negative but that decreases with every pass through the loop; the loop ends when the integer is 0.

In [7]:

*# decrementing function: 5 - x*

x **=** 0

**while** x **<** 5:

x **+=** 1

print(x)

**Iteration: *for* with decrementing function**

Graphical user interface, text, application, email

Description automatically generated

**Range()**

* In-built function that produces an immutable ordered non-scalar object of type range
* Initiate as range([start], stop, [step]). If ommitted, start = 0 and step = 1.
* Function produces progression of integers [start, start + step, start + 2\*step, ..., start + i\*step]
  + If step > 0, start + i\*step < stop
  + If step < 0, start + i\*step > stop

**Iteration**: ***break*****and *continue***

* Use *break* to exit a loop
* Use *continue* to go directly to next iteration

Graphical user interface, text, application, email

Description automatically generated

**List Comprehensions**

L = [\*object, expression, or function\* for \*element\* in \*sequence\*]

L = [\*object, expression, or function\* for \*element\* in \*sequence\* if \*Boolean expression\*]

L = [\*object, expression, or function\* for \*element\* in \*sequence\* for \*element2\* in \*sequence2\*]

* Provide a concise way to create lists
* Faster because implemented in C
* Nested list comprehensions can be somewhat confusing
* Can also do dictionary & set comprehensions with {}

**Exhaustive Enumeration (Brute force/guess and check)**

* Systematically enumerate all possible solutions until you get the right answer or run out of possibilities
* Example of **brute-force search** (a type of **guess and check** strategy) — a general problem-solving technique in computer science
* Surprisingly useful as computers are quite fast these days!

**Bisection Search**

* Start in the middle of the array, eliminate the half in which the answer cannot lie, and continue the search in the other half until you get the right answer or run out of possibilities
* Example of **divide and conquer** strategy — an algorithm-design paradigm in computer science.
* Naturally implemented as a recursive procedure (covered next week.
* *In divide and conquer algorithms you divide the problem into smaller pieces until you can solve them, then reassemble the pieces to find the final solution.*

**Iteration & Efficiency**

* **Half a loop is better than one**: Use continue and break to shorten iteration
* **One loop is better than two**: Consolidate loops whenever possible
* **Two loops are better than nested loops**: Attempt to rewrite any nested loops

**Readings: Week 3 –**

* 1. **– Branching Programs**

Straight-line programs execute one statement after another in the order they appear, branching programs execute in orders specified by the program, not just linearly.

Like a conditional:

1. A test i.e. an expression that evaluates to True or False
2. A block of code that is executed if test is True
3. An optional block of code that is executed in the test evaluates to False.

**Indentation is semantically meaningful in python.** If a line of code is indented then it follows from the previously non-indented command.

When either the true or false block of a conditional contains another conditional, the statements are said to be nested. (elif is else if).

Text, letter

Description automatically generated

A program for which the maximum running time is bounded by the length of the program is said to run in constant time. This does not mean that each time it is run it executes the same number of steps. It means that there exists a constant, k, such that the program is guaranteed to take no more than k steps to run. With constant-time programs, the length of time taken to execute the program is dependent on the number of times it must run.

**Chapter 3 – Some Simple Numerical Programs**

**Text

Description automatically generated**

**Text, letter

Description automatically generated**

* 1. **For Loops**

Variable following for is bound to the first value in the sequence, and the code block is executed. The variable is then assigned the second value in the sequence, and the code block is executed again. The process continues until the sequence is exhausted or a break statement is executed within the code block.

The sequence of values bound to variable is most commonly generated using the function range(), which returns a series of integers.

A picture containing text, newspaper, screenshot, document

Description automatically generated

**Week 4: Lecture – Functions**

Diagram

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**A Function Call Always Returns a Value**

* The execution of a return statement terminates the function call
* The function call also terminates when there are no more statements to execute
* If no expression follows return or there is no return statement, the function returns None

Text

Description automatically generated

Text

Description automatically generated with medium confidence

**Week 4: Readings – Functions**

**Ch.4 Functions, Scoping, and Abstraction**

Parameters provide something called **lambda abstraction,** allowing us to write code that manipulates not specific objects, but whatever objects the caller chooses as the parameters of the function.

Keyword vs. Position for parameter specification (arguments). Choose one!

* At level of shell, **symbol table** keeps track of all names defined at that level and their bindings.
* When a function is called, a new symbol table (often called a **stack frame**) is created. This keeps track of all names defined within a function.

MUST use docstrings *“”” “”””* to provide specifications for functions. These can be accessed using the built-in function **help.**

User of function has contract containing 2 things with function:

* Assumptions: these describe conditions that must be met by the user.
* Guarantees: These describe conditions that must be met by the function, if it has been called in a way that satisfies the assumptions.

Decomposition

* Creates structure.
* Allows us to break a program into parts that are reasonably self-contained, and that may be reused in different settings.

Abstraction

* Hides detail. Allows us to use a piece of code as if it were a black box.
* We cannot see interior details and don’t need, or want, to see them.

**Recursion**

In general, a recursive definition is made of two parts.

* There is at least one base case that directly specifies the result for a special case.
* At least one recursive (inductive) case that defines the answer in terms of the answer to the question on some other input, typically a simpler version of the same problem.

Could add some code that counts the number of calls to understand efficiency. We can use **global variables** for this. Put **global** before variable to express that you want to take global value. Issue is that locality helps make code readable.

**Modules**

A Module is a .py file containing Python definitions and statements, including functions. Each module has its own private symbol table. Use dot notation to not get mixed up with two functions w/ competing names.

**Files**

Python achieves operating-system independence by accessing files through something called a **file handle.**

nameHandle = open(‘kids’, ‘w’) instructs operating system to create a file with the name *kids* and return a file handle for that file. The argument ‘w’ to open indicates the file is to be opened for writing.

Text, letter

Description automatically generated

Text, letter

Description automatically generated

**Chapter 8. Classes & Object-Oriented Programming**

The key to object-oriented programming is thinking about objects as collections of both data and the methods that operate on that data.

Programmers can define new types. An abstract data type is a set of objects and the operations of those objects. One can pass an object from one part of the program to another. The specifications of those operations define an **interface** between abstract data type and the rest of the program. Interface defines the behaviour of the operations. What they do, but not how they do it. The interface thus provides an **abstraction barrier** that isolates the rest of the program from the data structures, algorithms, and code involved in providing a realization of the type abstraction.

In Python, one implements data abstractions using **classes.**

Text, letter

Description automatically generated

When a function definition occurs within a class definition, the defined function is called a **method** and is associated with the class. These methods are sometimes referred to as **method attributes** of the class. If this seems confusing at the moment, don’t worry about it.

Classes support two kinds of operations:

* Instantiation: Is used to create instances of the class. E.g. the statement s = IntSet() creates a new object of type IntSet. This object is called an instance of IntSet.
* Attribute references: use dot notation to access attributes associated with the class. E.g. s.member refers to the method **member** associated with the instance s of type IntSet.

Python has a number of special method names that start and end with two underscores \_\_init\_\_. Wehenver a class is instantiated, a call is made to the \_\_init\_\_ method defined In that class.

When s = IntSet() is executed, the interpreter will:

* 1. Create a new instance of type IntSet,
  2. Call IntSet.\_\_init\_\_ with the newly created object as the actual parameter that is bound to the former parameter *self.*
  3. When invoked, IntSet.\_\_init\_\_ creates *vals,* an object of type **list,** which becomes part of the newly created instance of type IntSet. (The list is created using [] which is simply an abbreviation of list()).
  4. The list is called a **data attribute** of the instance of IntSet. Notice that each object of type IntSet will have a different *vals* list, as one would expect.

Try:

1. S = IntSet() – creates a new instance of IntSet

2. s.insert(3) – inserts the integer 3 into that IntSet

3. print(s.member(3)) – prints True.

But member has two formal parameters and we’re calling with one? Dot notation means that expression preceding the dot is implicitly passed as the first parameter of the method.

Attributes can be associated either with a class itself or with an instances of a class:

* Method attributes are defined in a class definition. E.g. IntSet.member is an attribute of the class IntSet. When class is instantiated with s = IntSet(), attributes such as s.member are created.
* Keep in mind that IntSet.member and s.member are different objects. While s.member is initially bound to the **member** method defined in the class IntSet, that binding changed during the computation.
* We call data attrbitures associated with a class **class variables.**
* When they are associated with an instance we call them **instance variables.** For example, *vals* is an instance variable because for each instance of class IntSet, *vals* is bound to a different list.

Data abstraction achieves representation-independence. Think of the implementation of an abstract type as having several components:

* Implementations of the method of the type,
* Data structures that together encode values of the type, and,
* Conventions about how the implementations of the methods are to use the data structures. A key convention is captured by the representation invariant.

The **representation invariant** defines which values of the data attributes correspond to valid representations of class instances. The representation invariant for intSet is that vals contains no duplicates. The implementation of \_\_init\_\_ is responsible for establishing the invariant (which holds on the empty list), and the other methods are responsible for maintaining that invariant. That is why **insert** appends e only if it is not already in self.vals.

The implementation of **remove** exploits the assumption that the representation invariant is satisfied when remove is entered. It calls list.remove only once since representation variant guarantees that there is only one occurrence (at most) of e in self.vals.

\_\_str\_\_ is a special method.

s = IntSet()

s.insert(3)

s.insert(4) print(s)

will print {3,4}

The \_\_str\_\_ method of a class is invoked when a program convrts an instance of that class to a string by calling **str.**

All user-defined classes are hashable and can be used as dictionary keys.

Abstract data types lead to a different way of thinking about organizing large programs. Data abstraction encourages program designers to focus on the centrality of data objects rather than functions. Thiknking about a program more as a collection of types than as a collection of functions leads to a profoundly different organizing principle.

**8.2 Inheritance**

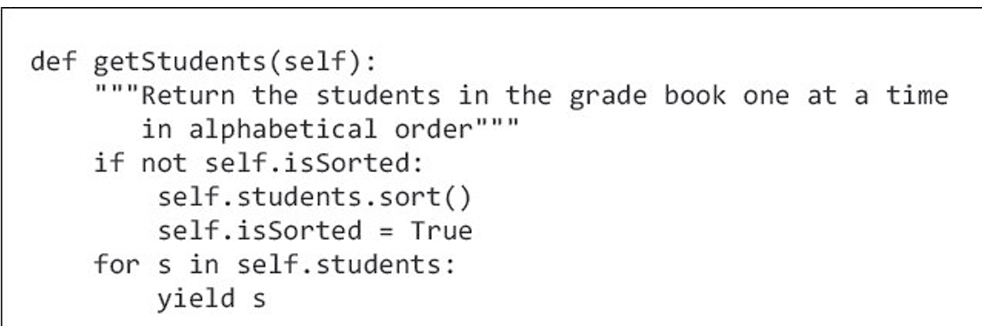
Many types have properties in common. Types list() and str() each have len() functions that mean the same thing. Within a class, following objects **inherit** from above. They inherit all attributes from parent class. In the jargon, the **subclasss** inherits the attributes of its **superclass.**

However, there can be multiple level of inheritance. By using the word **pass** in the body, we indicate that the class has no attributes other than those inherited from its superclass. Example on page 127.

Substitution principle – When subclassing is used to define a type hierarchy, the subclasses should be thought of as extending the behaviour of their superclasses. We do this by adding new attributes or overriding attributes inherited from a superclass.

**Generators**

Any function definition containing a **yield** statement is treated in a special way. The presence of **yield** tells the Python system that the function is a generator. Generators are typically used In conjunction with for statements, as in:



MORTAGES EXAMPLE AFTER

**Week 7: Readings – Testing & Debugging and Exceptions and Assertions**

**6.1 Testing**

Testing is used to show that bugs exist.

**Teste suite** is collection of inputs that has a high likelihood of revealing bugs but does not take too long tu run.

Heuristics based on exploring paths through the code fall into a class called **glass-box testing.**

Heuristics based on exploring paths through the specification fall into a call called **black-box testing.**

6.1.1 Black-box Testing

Black-box tests are constructed without looking at the code to be tested.

* An advantage of black-box testing is that it is robust wth respect to implementation changes. Since the test data is generated without knowledge of the implementation, the tests need not be changed when the implementation is changed.

6.1.2 Glass-box Testing

* By looking at code, you can identify special cases to be tested

A glass-box test suit is path-complete if it exercises every potential path through the program.

Best practice includes:

* Exercise both branches of all if statements
* Make sure that each except clause is executed.
* For each for loops, have test cases in which:
  + Loop is not entered, e.g. on empty list.
  + The body of loop is executed once.
  + The body of loop is executed more than once.
* For each while loop:
  + Look at the same kinds of cases as when dealing with for loops
  + Include test cases corresponding to all possible ways of exiting the loop
* For recursive functions, include test cases that cause the function to return with no recursive calls, exactly one recursive call, and more than one.

6.1.3 Conducting Tests

Two phases:

* Unit testing
  + Do individual units of code work?
* Integration testing
  + Does program as a whole behave correctly?

**6.2 Debugging**

Runtime bugs can be categorized across two dimensions:

* Overt -> Covert
  + Overt = program crashes or takes too long
  + Covert has no obvious manifestation.
* Persistent -> Intermittent
  + A persistent bug occurs every time the program is run with the same inputs.
  + An intermittent bug occurs only some of the time.

Good programmers try to write their programs in such a way that programming mistakes lead to bugs that are both overt and persistent. This is often called defensive programming.

6.2.1 Learning to Debug

Debugging is a learned skill.

Print() statement is most important debugging skill.

Need to be systematic.

1. Study test results
2. Form hypothesis
3. Design and repeat experiment

6.2.2 Designing the Experiment

Think of debugging as a search process and each experiment as an attempt to reduce the size of the search space.

Usual suspects for debugging found on 6.2.3

* Stop asking yourself why the program isn’t doing what you want it to. Instead, ask yourself why it is doing what it is.
* The bug is probably not where you think it is
* Try to explain the problem to somebody else
* Try again tomorrow

6.2.4 When You Have Found ‘The’ Bug

**Chapter 7 – Exceptions and Assertions**

Most of the built-in exceptions of Python deal with situations in which a program has attempted to execute a statement with no appropriate semantics.

7.1 Handling Exceptions

When an exception is raised, the program terminates (crashes might be a more appropriate word in this case). When an exception is raised that causes the program to terminate, we say that an unhandled exception has been raised. An exception does not need to lead to program termination. Exceptions, when raised, can and should be **handled** by the program. Sometimes an exception is raised because there is a bug in the program, but many times an exception is something the programmer can and should anticipate. In a well-written program, unhandled exceptions should be the exception.

**Text, letter

Description automatically generated**

**Text, letter

Description automatically generated**

**Exceptions should be treated as informative. They are a convenient flow-of-control mechanism that can be used to simplify programs.**

**7.2 Exceptions as a Control Flow Mechanism**

In many programming languages, the standard approach to dealing with errors is to have the function return a value. In Python, it is more usual to have a function raise an exception when it cannot produce a result that is consistent with the function’s specification.

The **raise** statement forces a specified exception to occur.

Raise exceptionName(arguments)

Raise ValueError

Can define new exceptions by creating a subclass of the built-in class Exception.

Text

Description automatically generated

7.3 Assertions

The python **assert** statement provides programmers with a simple way to confirm that the state of a computation is as expected. An assert statement can take one of two forms:

Assert Boolean expression

Or

Assert Boolean expression, argument

When an assert statement is encountered, the Boolean expression is evaluated. If it evaluates to True, execution proceeds on its merry way. If it evaluates to False, an AssertionError exception is raised.

Assertions are a useful defensive programming tool. They can be used to confirm that the arguments to a function are of appropriate types. They can also be a useful debugging tool. E.g. they can be used to confirm the intermediate values have the expected values or that a function returns an acceptable value.