CA117 Documentation

Release 2022

Darragh O'Brien B.Sc. Ph.D.

Sep 01, 2022

CONTENTS

CHAPTER ONE

LECTURE 1.1 : MODULE OVERVIEW

1.1 Introduction

- CA117 is a second course in computer programming using the Python language.
- It is assumed you are already familiar with the Python programming environment (running the Python interpreter and editing text files), basic Python data types (strings, lists, dictionaries, etc.) and are able to write short programs that make decisions with `if` statements, iterate with `while` and `for` loops and that use functions to decompose a problem.
- This course will enhance your programming skills so that you will be able to write more complex programs using more sophisticated techniques, in particular those of object-oriented programming.
- It is a practical programming course that requires a commitment to writing lots of programs.

1.2 Module homepage

- On the homepage you will find lecture notes, lab exercises, bucketlist assignments, announcements, links to videos, useful links, further reading, etc.

1.3 Einstein

- As with CA116, upload your programs to Einstein to have them verified.
- As with CA116, Einstein will track for you your progress (in terms of completed lab exercises) compared with other students in the class. (Student details apart from your own are anonymized.) Use this facility to track how you are doing compared to your classmates. Obviously, your aim should be to avoid falling too far behind.

1.4 Timetable (weeks 1-12)

- Tuesday:
  - 0900-1100 : lecture in QG13
  - 1400-1600 : labs in L101, L114, L125, L128
- Thursday:
  - 1100-1300 : lecture in QG13
  - 1400-1600 : labs in L101, L114, L125, L128
- Timetabling is not under my control so if you have any issues with the timetable raise them with the chair of your degree programme.

1.5 Lectures

- Normally each lecture will:
  1. Introduce some Python programming concept, and subsequently,
  2. Demonstrate the practical application of that concept in code.
- Normally a lecture involves me coding in a Python notebook. These notebooks are made available at the start of each week. I recommend you take a copy and follow along with me on your laptop during a lecture.
- Normally some time is allocated each week to reviewing previous lab exercises. During these sessions we will examine some common programming errors extracted from code uploaded by anonymized CA117 students.
- 2021's lectures were delivered on-line. These will be made available to you.
- Additional videos will also be posted covering lecture material, walkthroughs of solutions to selected lab exercises, programming tips, pointers to on-line resources, etc.

1.6 Continuous assessment

- Continuous assessment accounts for 50% of your overall mark in this module and is made up of four components:

CA component	Marks	Due
Labs	5%	Weekly
Bucketlist	10%	Weeks 9, 12
Lab exam 1	15%	Week 6 (Tuesday 15 February 1400-1550)
Lab exam 2	20%	Week 12 (Tuesday 29 March 1400-1550)

- The program below calculates your final CA mark (as an overall percentage) from your marks in each of the four CA elements. Substitute your own marks to work out your final CA mark.

```
#!/usr/bin/env python3

from decimal import Decimal, ROUND_HALF_UP

def main():

    # Sample marks in each CA component
    labs = 66
    bucket_list = 50
    labexam_01 = 33
    labexam_02 = 44

    ca = (1 * labs) + (2 * bucket_list) + (3 * labexam_01) + (4 *
↪labexam_02)
    ca = ca / 10

    # Round to nearest integer (with .5 always rounding up)
    ca = int(Decimal(ca).to_integral_value(ROUND_HALF_UP))

    print(f'Your overall CA mark is: {ca:d}%')

if __name__ == "__main__":
    main()
```

```
Your overall CA mark is: 44%
```

1.7 Labs (5%)

- You cannot learn how to program from lecture notes alone. Programming ability is a *skill* that is acquired through practice and through learning from mistakes.
- You must *at a minimum* complete all lab programming exercises. Do not expect to always complete each set of lab exercises during the scheduled lab time. You are expected to dedicate considerable additional private study time in order to master the course material.
- Similar to CA116, when on campus, lab tutors will be present to help you with programming exercises. Their role is to help you find a suitable solution and **not** to provide you with a solution.
- Similar to CA116, you are required to upload your code to Einstein. It will help you verify you have successfully solved lab exercises. (It will also help me identify common programming errors that can be addressed in class.)
- Lab exercises contribute 5% to your overall mark in this module. Each lab has an associated completion deadline. Credit is awarded only for successfully completed lab exercises that are submitted before the deadline. For example, if over the course of the semester you successfully complete 60 of 80 lab exercises on time then your lab mark would be 75%.
- To receive the marks available for a lab exercise it must pass all of the associated test cases on Einstein.
- Normally at least one of the test cases on Einstein for any particular lab exercise will be hidden. This means you will not be able to see the inputs used to test your program.
- Passing the public test cases but failing a hidden test indicates you have not sufficiently tested your code and you will need to devise your own additional test cases based on the problem description (coming up with test cases is part of the exercise and something real world programmers have to do).
- Selected solutions to previous lab exercises will appear here.
- All lab submissions must adhere to DCU's academic integrity policy. See *Academic integrity*.

1.8 Bucketlist (10%)

- A **bucketlist** of two programming assignments will be posted over the course of the semester. These programming assignments contribute 10% to your overall mark in this module.
- Bucketlist exercises must be uploaded to Einstein before their associated deadline.
- All bucketlist submissions must adhere to DCU's academic integrity policy. See *Academic integrity*.

1.9 Lab exams (35%)

- Two lab exams will take place over the course of the semester and contribute a total of 35% to your overall mark in this module.
- **Lab exams will take place during the Tuesday lab slots of weeks 6 and 12.**
- **Lab exams cannot be rescheduled.**
- There will be no lectures on the lab exam days.
- The first lab exam is worth 15% of your overall mark.
- The second lab exam is worth 20% of your overall mark.
- Lab exam exercises will be solvable using techniques we have covered up until the time of the lab exam.
- Each lab exam lasts 1 hour and 50 minutes.
- You must log in under Linux to take the lab exam.
- You must log in ten minutes before the start of the exam from your assigned lab.
- All phones must be turned off and placed on the computer case in view of invigilators.
- Your student ID card must be placed on your desk in view of invigilators.
- Apart from your ID card there must be nothing on your desk.
- Unless otherwise stated lab exams are posted at the bottom of the CA117 home page. Lab exams become accessible at the exam start time.
- Upload your solutions to Einstein. Login here with your DCU credentials. If you fail to login here with your DCU credentials you will receive a mark of zero.
- Only code uploaded to Einstein during the lab exam will be graded. All uploads are IP-stamped and time-stamped. The submission deadline will be strictly enforced.
- It is your responsibility to ensure your Python code is compatible with Einstein's Python version.
- All exercises must be completed (unless otherwise stated).
- Your programs may only import from the following Python modules: `sys`, `math`, `re`, `string`.
- Your programs are not permitted to use `eval`.
- During the exam you will have access to the CA117 web site.
- During the exam you will have access to model solutions.
- During the exam you will have access to TermCast.

- During the exam you will log in as normal and will have access to your home directory.
- During the exam you will **not** have general Internet access (apart from to those resources mentioned above).
- During the exam you will **not** have access to videos and notebooks on the CA117 Google Drive.
- All lab exam submissions must adhere to DCU's academic integrity policy. See *Academic integrity*.
- Sample input and output are provided for each exercise. These sample test cases are not exhaustive. Passing such sample test case(s) does not guarantee marks for that exercise. Your mark for each exercise is calculated based on additional test cases applied during the marking process. See *Marking of bucketlist, lab exam and final exam exercises*.
- It is your responsibility to save your work regularly.
- It is your responsibility to upload your work regularly.
- If you miss the exam due to illness, **you must submit a medical cert** to cover your absence.
- A sample lab exam will typically be supplied in advance of the real lab exam.
- A sample lab exam solution will typically be supplied in advance of the real lab exam.
- The sample lab exam and the real lab exam will **not** be the same.

1.10 Lab exam FAQ

- **Q.** I cannot attend the lab exam because I am playing sport for DCU/Dublin/Leinster/Ireland/Europe etc. on that day. What can be done?
- **A.** A supplementary exam will **not** be organised to facilitate students who do not attend a lab exam. Thus you need to confirm well in advance of the lab exam and with the **chair of your degree programme** how your situation is to be handled.
- **Q.** What if I have a question during the lab exam?
- **A.** It should be clear from the exercise description and the example input and output what is required. However if you are confused you can ask me or an invigilator.
- **Q.** I very nearly solved question X. Am I awarded attempt marks?
- **A.** See *Marking of bucketlist, lab exam and final exam exercises*.
- **Q.** My program passed the supplied test cases but did not receive full marks. Why not?
- **A.** See *Marking of bucketlist, lab exam and final exam exercises*.
- **Q.** Why is use of `eval` not permitted?

- **A.** It's bad programming.
- **Q.** My computer crashed/exploded/disappeared during the exam. Can I have an extension?
- **A.** No. An extension of 15 minutes for technical difficulties has been built into the exam duration.
- **Q.** I ran out of disk space during the exam. Can I have an extension?
- **A.** No. Managing your account's contents is your responsibility. To identify bloated directories run this command in your home directory: `du -h -d1`. It should be clear from the output where your big files are living.
- **Q.** I uploaded my code a mere X minutes after the deadline. Will it be marked?
- **A.** No. An extension of 15 minutes has been built into the exam duration.
- **Q.** My program works with the version of Python I have installed at home but does not work with Einstein's version. Can I have some marks?
- **A.** No. It is your responsibility to ensure your code is compatible with Einstein's Python version.
- **Q.** How should I test my program?
- **A.** In general, you need to read the question carefully and identify the range of possible inputs. Following that, identify any *boundary* or *edge* or *awkward* cases since it is often here where problems can arise. As a simple example, if the exercise states input is an integer in the range [-100, 100] you might devise the following test cases:
  - -100 (end of range),
  - 100 (end of range),
  - 0 (positive/negative boundary),
  - -1 (start negative range),
  - 1 (start positive range),
  - -50 (mid-range negative),
  - 50 (mid-range positive).
- **Q.** Have you any lab exam tips?
- **A.** Some general tips:
  - Read each question carefully. Study the example input and output.
  - Some questions are easier than others (typically the early ones) and I would do the easier ones first.
  - Try not to panic. If a particular lab exam goes poorly you have other opportunities to catch up: labs, bucketlist, the other lab exam, the final exam.

  - Your final upload for each question is the one that counts so ensure your final upload is your best attempt.
  - Know where to find things in the notes.

1.11 Catch-up lab exam

- A catch-up lab exam for those who missed a lab exam in week 6 or 12 (e.g. for COVID-related reasons) will take place on Monday 25 April 1000-1150 in L101.
- Only those students who missed one or both of the lab exams in weeks 6 or 12 are eligible to sit this lab exam.
- Only those students who informed me prior to the lab exams in weeks 6 or 12 of their forced absence are eligible to sit this lab exam.
- This lab exam will contain a mix of questions of the type included in lab exam 1 and lab exam 2.
- This lab exam compensates only for a missed lab exam in week 6 or 12.
- The normal lab exam rules apply.
- All exam submissions must adhere to DCU's academic integrity policy. See *Academic integrity*.

1.12 Final exam (50%)

- A final exam, worth 50% of your overall mark, takes place in April.
- The exam lasts 3 hours.
- The exam is lab-based and the normal lab exam rules apply.
- All exam submissions must adhere to DCU's academic integrity policy. See *Academic integrity*.

1.13 Marking of bucketlist, lab exam and final exam exercises

- Test cases, where supplied on Einstein, will be visible but will **not** be exhaustive. These *sample* test cases will serve as an aid to developing your solution.
- Devising your own additional test cases is considered part of the exercise (as it is part of being a programmer in the real world).

- Passing all tests cases on Einstein will thus not guarantee all marks for that exercise. Additional tests will be run on your code afterwards as part of the marking process.
- Marks may be awarded for reasonable attempts so ensure your final upload for each exercise is your best attempt.
- Marks may be deducted for code that names variables inappropriately, is incomprehensible, overly long, poorly structured, inefficient, incomplete, etc.

1.14 Academic integrity

- Ensure you read DCU's Academic Integrity and Plagiarism Policy.
- Sharing your work with anyone is a breach of the above policy.
- Copying work from anyone is a breach of the above policy.
- All Einstein submissions will be actively monitored for signs of collusion/copying.
- Any breach of the above policy is a serious offence that will result in penalties and/or the application of disciplinary procedures.

1.15 How to pass

- Your overall mark is calculated from two components: continuous assessment (worth 50%) and one written exam (worth 50%).
- To pass the module your overall mark must be 40+.

1.16 Resit information

- Only in the event that your *overall* mark is less than 40 must you resit some component(s):
  - If you failed the final exam you must resit it,
  - if you failed the CA you must resit it,
  - if you failed the final exam and the CA you must resit both.

1.17 Resit exam

- Friday 5 August 2022, 0930-1230, L101.
- The resit exam (worth 50% of your overall resit mark) consists of an on-campus lab exam. This exam will follow a format similar to the final exam. You must complete the exam on Linux.

1.18 Resit CA

- Thursday 11 August, 0930-1250, L101.
- Resit CA (worth 50% of your overall resit mark) consists of an on-campus quiz (0930-1030) followed by lab exam (1100-1250).
- A Loop-based quiz:
  - 0930-1030 (you should be seated in L101 by 0915).
  - Worth 20% of your overall resit mark.
  - You will have 60 minutes to complete this closed-book quiz (i.e. no access to notes or Python interpreter).
  - You must complete the quiz on Windows.
  - Sample quizzes are available on the CA117 Loop pages.
- A Linux lab exam:
  - 1100-1250 (you should be seated in L101 by 1045).
  - Worth 30% of your overall resit mark.
  - Follows a format similar to lab exams 1 and 2 from term time.
  - Must be completed on Linux and the normal lab exam rules apply.

1.19 Continuous assessment extensions and absences

- As per DCU regulations and in order to be fair to all students, any request by a student for an extension to any continuous assessment component(s) must be accompanied by a medical cert.
- Absence from a lab exam due to illness must also be covered by a medical cert.

1.20 Attendance

- Physical attendance at lectures and labs is recorded through Loop.
- Attendance numbers are tracked only to monitor overall student engagement and to plan module delivery.
- Attendance does not contribute to your final grade.

1.21 Resources

- All course notes and programming exercises will be made available on-line.
- Should you require further resources, there are numerous Python programming books available in the library (some are available as e-books) including the following:
  - *Starting out with Python* by Gaddis,
  - *Introducing Python* by Lubanovic,
  - *The Practice of Computing Using Python* by Punch and Enbody,
  - *Introduction to Programming in Python* by Sedgewick, Wayne and Dondero.
- There are also numerous Python programming tutorials available on-line including the following:
  - Think Python: How to Think Like a Computer Scientist,
  - A Byte of Python,
  - Learn Python the Hard Way,
  - The Python Tutorial.

1.22 Programming help desk in L114A

- Monday: 0900-1000, 1200-1300.
- Tuesday: 0900-1100.
- Wednesday: 0900-1100, 1200-1400.
- Thursday: 1100-1300, 1600-1800.
- Friday: 0900-1400 on Zoom.

1.23 Struggling with programming?

- Check out Computer Science Circles.
- Check out Khan Academy's Python YouTube Channel.
- Sign up for edX and check out their programming courses (to earn a certificate you have to pay but you can audit some courses for free).
- Sign up for Coursera and check out their programming courses (you can audit some courses for free).
- Take a look at Udemy for Python programming courses.
- Contact the School's Programming Help Desk.
- Check out the Python programming e-books available from DCU library.
- Study and attempt past exam papers.
- Use a visualizer to step through code so you can see how it works.
- You need to practise programming at home so get Python installed on your laptop or PC (see below). As you build experience code patterns will start to come to you automatically e.g. how to read in every line from `stdin`, splitting it and building a dictionary from it, etc.
- When writing a program, write it piece-by-piece and save and run it after each step to slowly build-up a solution.

1.24 Python programming off-campus

- You should install Python 3 on your laptop or PC so that you can practise programming while off-campus.
- Alternatively log into TermCast (if you use TermCast during labs I can drop in to your session to offer advice).
- Google Colab comes with a built-in Python interpreter so you could also use that.

1.25 Contacting me

- Talk to me during a lab or lecture.
- E-mail me from your DCU e-mail account with CA117 in the subject.
- An anonymous survey is normally distributed in week 5 and provides you with an opportunity to supply feedback on how the module is working for you and on how you think it might be improved.

1.26 Slack

- There is a CA117 Slack channel.
- Use it to ask questions and answer your classmates' questions.
- Go to the sign-up page and join the "#ca117-general" channel.

1.27 Python version

- We will be programming on Linux with Python 3 (your code must be compatible with Python version 3.9.2 running on Einstein). To invoke this version of the Python interpreter you must enter `python3` at the command prompt (and not `python` which runs Python version 2):

```
$ python3
Python 3.9.2 (default, Feb 28 2021, 17:03:44)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more␣
↪information.
>>> print('hello')
hello
```

- The Python 3 standard library documentation is a resource you should consult regularly.
- Your code should adhere to the Python Style Guide.

1.28 Module synopsis

- We will begin by reviewing, consolidating and, where applicable, extending the programming skills you acquired through completing CA116.
- We will then move on to study *object-oriented programming*.

CHAPTER

TWO

LECTURE 1.2 : STRINGS

2.1 Introduction

- A string is simply a *sequence* of characters. The string type is a *collection type* meaning it contains a number of objects (characters in this case) that can be treated as a single object. The string type is a particular type of collection type called a *sequence type*. This means each constituent object (here character) in the collection occupies a specific numbered location within it i.e. elements are *ordered*.
- Python strings are typically enclosed in single or double quotes. (Pick a style and be consistent throughout and across your programs.)

```
name1 = "Jimmy Murphy"
name2 = 'Mary Kelly'
print(name1)
print(name2)
```

```
Jimmy Murphy
Mary Kelly
```

- What if you want to include quotes in your string? You have a couple of options: You can enclose your string in the other kind of quote or you can *escape* the quote with a backslash (nullifying its role as a string delimiter).

```
name3 = "Nora O'Neill"
name4 = 'Sally O\'Brien'
print(name3)
print(name4)
```

```
Nora O'Neill
Sally O'Brien
```

- Python strings can contain *non-printing* characters (such as a `\n` which causes a new line to be emitted when printing the string).

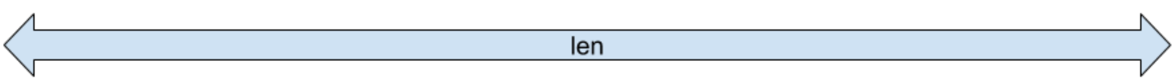
```python
rhyme = "Humpty Dumpty sat on a wall,\nHumpty Dumpty had a great fall."
print(rhyme)
```

```
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
```

2.2 String indexing

- As mentioned, a string is a *sequence* type. This means each member object (character in this case) occupies a numbered position in the collection and can be accessed by *indexing* the sequence at that index.
- For example, the characters of the string `'This is a sentence.'` reside at the indices indicated here:

-19	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
T	h	i	s		i	s		a		s	e	n	t	e	n	c	e	.
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



- The length of a string is the number of characters it contains. The `len()` function returns the length of a string.
- We can extract individual characters by *indexing* the string at a given location. The first character in the string is located at index zero. If the length of the string is N, the final character is located at index N-1. Indexing outside string boundaries gives rise to an error.

```python
s = 'This is a sentence.'
print(len(s))
print(s[0])
```

(continues on next page)

(continued from previous page)

```
print(s[1])
print(s[2])
print(s[18])
print(s[19])
```

```
19
T
h
i
.
```

```
---------------------------------------------------------------------
↪--------
IndexError                                Traceback (most recent␣
↪call last)
/tmp/ipykernel_6499/3585678721.py in <module>
      5 print(s[2])
      6 print(s[18])
----> 7 print(s[19])

IndexError: string index out of range
```

- In Python it is possible to index relative to the end of the string using negative indices: The last character is at index -1, the second last at index -2, the third last at index -3, etc.

```
print(s[-1])
print(s[-2])
print(s[-3])
print(s[-19])
print(s[-20])
```

```
.
e
c
T
```

```
---------------------------------------------------------------------
↪--------
IndexError                                Traceback (most recent␣
↪call last)
```

(continues on next page)

(continued from previous page)

```
/tmp/ipykernel_6499/79585671.py in <module>
      3 print(s[-3])
      4 print(s[-19])
----> 5 print(s[-20])

IndexError: string index out of range
```

2.3 String slicing

- We can extract more than a single character from a string. We can extract subsequences or *slices* by specifying a range of indices separated by a colon. Writing `s[start:end]` will return a new string composed of the characters `s[start]`, `s[start+1]`, `s[start+2]`, `...`, `s[end-3]`, `s[end-2]`, `s[end-1]`.
- Note that the character located at `s[end]` is *not* returned.

```
s = 'This is a sentence.'
print(s[0:4])
print(s[5:7])
```

```
This
is
```

- If either the starting or ending indices are omitted their values default to the beginning and end of the string.

```
s = 'This is a sentence.'
print(s[:4])
print(s[10:])
print(s[:])
```

```
This
sentence.
This is a sentence.
```

- As usual, negative indices can be used to specify locations relative to the end of the string.

```
s = 'This is a sentence.'
print(s[:-10])
```

```
This is a
```

- Slicing is more forgiving in terms of indices than pure indexing. A start or end index outside of a string's boundaries is treated as the string boundary.

```
s = 'This is a sentence.'
print(s[:100])
print(s[-100:])
print(s[-100:100])
```

```
This is a sentence.
This is a sentence.
This is a sentence.
```

2.4 Extended slicing

- It is possible to specify a third parameter when slicing sequences. It indicates the *step size* to take along the sequence when extracting its elements. Writing `s[start:end:step]` will return a new string composed of `s[start]`, `s[start+step]`, `s[start+2*step]`, `s[start+3*step]`, etc. Extraction continues for as long as `start+i*step < end` where i = 0, 1, 2, 3, etc.
- As usual, if the start or end of the slice are omitted they default to the beginning and end of the string respectively.

```
s = 'This is a sentence.'
print(s[::1])
print(s[::2])
print(s[:10:3])
print(s[10::3])
```

```
This is a sentence.
Ti sasnec.
Tss
stc
```

- A negative step size is interpreted as a step backwards through the sequence. This is handy for reversing a string.
- For a negative step size, if the start or end of the slice are omitted, their values default to the end and beginning of the string respectively.

```
s = 'This is a sentence.'
print(s[-1:-20:-1])
print(s[::-1])
print(s[::-2])
print(s[-18:-10:-1])
print(s[-10:-18:-1])
```

```
.ecnetnes a si sihT
.ecnetnes a si sihT
.censas iT

 a si si
```

2.5 String concatenation and replication

- We can use the + operator to concatenate strings.

```
line1 = 'Humpty Dumpty sat on a wall'
line2 = 'Humpty Dumpty had a great fall'
print(line1 + ',\n' + line2 + '.')
```

```
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
```

- We can use the * operator to replicate strings.

```
s = 'apple '
print(s * 3)
```

```
apple apple apple
```

2.6 String comparison

- Strings can be tested for equality with the == operator.

```
print('cat' == 'dog')
print('mouse' == 'mouse')
```

```
False
True
```

2.7 Strings are `True` or `False`

- The empty string `''` is interpreted as `False`.

```
s = ''
if s:
    print(True)
else:
    print(False)
```

```
False
```

- Any non-empty string is interpreted as `True`.

```
s = 'apple'
if s:
    print(True)
else:
    print(False)
```

```
True
```

2.8 Strings are iterable

- Because a string is an *iterable* sequence we can use a `for` loop to examine each of its characters in turn.

```
s = 'apple'
for c in s:
    print(c)
```

```
a
p
p
l
e
```

2.9 Strings are immutable

- Strings are *immutable*. This means they cannot be modified. If we try to modify a string we get an error.

```
s = 'apple'
s[0] = 'A'
print(s)
```

```
---------------------------------------------------------------------------
↪--------
TypeError                                 Traceback (most recent␣
↪call last)
/tmp/ipykernel_6499/2115023848.py in <module>
      1 s = 'apple'
----> 2 s[0] = 'A'
      3 print(s)

TypeError: 'str' object does not support item assignment
```

- If we want a "modify" a string we have to create a new one from the original.

```
s = 'apple'
s = 'A' + s[1:]
print(s)
```

```
Apple
```

2.10 Processing lines of text

- A common operation is to read in a line of text from a file, strip any surrounding whitespace and split the line into its constituent tokens.

```
line = 'This is a line of text'
tokens = line.strip().split()
print(tokens)
```

```
['This', 'is', 'a', 'line', 'of', 'text']
```

- By default, the `split()` method splits strings on whitespace but we can ask it to split on other characters.

```
line = 'This,is,a,line,of,text'
tokens = line.strip().split(',')
print(tokens)
```

```
['This', 'is', 'a', 'line', 'of', 'text']
```

- We can use the `join()` method to glue lists of words back together into a single string. We need to tell `join()` which glue character to use.

```
line = 'This,is,a,line,of,text'
tokens = line.strip().split(',')
print(tokens)
newline = ' '.join(tokens)
```

(continues on next page)

(continued from previous page)

```
print(newline)
newline = '-'.join(tokens)
print(newline)
```

```
['This', 'is', 'a', 'line', 'of', 'text']
This is a line of text
This-is-a-line-of-text
```

2.11 String membership

- We can use the `in` operator to check whether one string is a substring of another.

```
s = 'This is a sentence.'
print('This' in s)
print('x' in s)
print('.' in s)
```

```
True
False
True
```

2.12 String methods

- Python comes with built-in support for a large set of common string operations. These operations are called `methods` and they define the things we can do with strings. We have looked only at a small number of Python's string methods.
- Calling `help(str)` in the Python shell or `pydoc str` in a Linux terminal outputs a list of these methods. We see the methods we can invoke on a string `s` include `capitalize()` (returns a capitalized version of `s`), `isdecimal()` (returns `True` if `s` contains only decimal characters), `lower()` (returns a new copy of `s` with all characters converted to lowercase), etc.
- Many editors and IDEs support autocompletion, whereby, if `s` is a string, placing your cursor after the dot in `s.` and hitting tab will provide you with a list of the string methods that can be invoked on `s`. Then simply select the one you want.
- Note that, because strings are immutable, calling a method on a string will *not* alter the string itself.

```
s = 'apple'
print(s.capitalize())
print(s)
```

```
Apple
apple
```

- Whenever you find it necessary to carry out some string processing, first look up the list of built-in string methods. There may be one that will help you with your task. There is no point writing code that duplicates what a built-in string method can do for you already.

```
help(str)
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding␣
↪or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error␣
↪handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
```

(continues on next page)

(continued from previous page)

```
 |      Return a formatted version of the string as described by␣
↪format_spec.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(...)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rmod__(self, value, /)
```

(continues on next page)

(continued from previous page)

```
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __sizeof__(self, /)
|      Return the size of the string in memory, in bytes.
|
|  __str__(self, /)
|      Return str(self).
|
|  capitalize(self, /)
|      Return a capitalized version of the string.
|
|      More specifically, make the first character have upper␣
↪case and the rest lower
|      case.
|
|  casefold(self, /)
|      Return a version of the string suitable for caseless␣
↪comparisons.
|
|  center(self, width, fillchar=' ', /)
|      Return a centered string of length width.
|
|      Padding is done using the specified fill character␣
↪(default is a space).
|
|  count(...)
|      S.count(sub[, start[, end]]) -> int
|
|      Return the number of non-overlapping occurrences of␣
↪substring sub in
|      string S[start:end].  Optional arguments start and end are
|      interpreted as in slice notation.
|
|  encode(self, /, encoding='utf-8', errors='strict')
|      Encode the string using the codec registered for encoding.
|
|      encoding
|        The encoding in which to encode the string.
|      errors
|        The error handling scheme to use for encoding errors.
|        The default is 'strict' meaning that encoding errors␣
↪raise a
```

(continues on next page)

(continued from previous page)

```
 |        UnicodeEncodeError.  Other possible values are 'ignore',
↪'replace' and
 |        'xmlcharrefreplace' as well as any other name registered␣
↪with
 |        codecs.register_error that can handle␣
↪UnicodeEncodeErrors.
 |
 |  endswith(...)
 |      S.endswith(suffix[, start[, end]]) -> bool
 |
 |      Return True if S ends with the specified suffix, False␣
↪otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      suffix can also be a tuple of strings to try.
 |
 |  expandtabs(self, /, tabsize=8)
 |      Return a copy where all tab characters are expanded using␣
↪spaces.
 |
 |      If tabsize is not given, a tab size of 8 characters is␣
↪assumed.
 |
 |  find(...)
 |      S.find(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice␣
↪notation.
 |
 |      Return -1 on failure.
 |
 |  format(...)
 |      S.format(*args, **kwargs) -> str
 |
 |      Return a formatted version of S, using substitutions from␣
↪args and kwargs.
 |      The substitutions are identified by braces ('{' and '}').
 |
 |  format_map(...)
 |      S.format_map(mapping) -> str
 |
 |      Return a formatted version of S, using substitutions from␣
↪mapping.
```

(continues on next page)

(continued from previous page)

```
|       The substitutions are identified by braces ('{' and '}').
|
|   index(...)
|       S.index(sub[, start[, end]]) -> int
|
|       Return the lowest index in S where substring sub is found,
|       such that sub is contained within S[start:end].  Optional
|       arguments start and end are interpreted as in slice␣
↪notation.
|
|       Raises ValueError when the substring is not found.
|
|   isalnum(self, /)
|       Return True if the string is an alpha-numeric string,␣
↪False otherwise.
|
|       A string is alpha-numeric if all characters in the string␣
↪are alpha-numeric and
|       there is at least one character in the string.
|
|   isalpha(self, /)
|       Return True if the string is an alphabetic string, False␣
↪otherwise.
|
|       A string is alphabetic if all characters in the string are␣
↪alphabetic and there
|       is at least one character in the string.
|
|   isascii(self, /)
|       Return True if all characters in the string are ASCII,␣
↪False otherwise.
|
|       ASCII characters have code points in the range U+0000-
↪U+007F.
|       Empty string is ASCII too.
|
|   isdecimal(self, /)
|       Return True if the string is a decimal string, False␣
↪otherwise.
|
|       A string is a decimal string if all characters in the␣
↪string are decimal and
|       there is at least one character in the string.
|
|   isdigit(self, /)
```

(continues on next page)

(continued from previous page)

```
 |      Return True if the string is a digit string, False␣
↪otherwise.
 |
 |      A string is a digit string if all characters in the string␣
↪are digits and there
 |      is at least one character in the string.
 |
 |  isidentifier(self, /)
 |      Return True if the string is a valid Python identifier,␣
↪False otherwise.
 |
 |      Call keyword.iskeyword(s) to test whether string s is a␣
↪reserved identifier,
 |      such as "def" or "class".
 |
 |  islower(self, /)
 |      Return True if the string is a lowercase string, False␣
↪otherwise.
 |
 |      A string is lowercase if all cased characters in the␣
↪string are lowercase and
 |      there is at least one cased character in the string.
 |
 |  isnumeric(self, /)
 |      Return True if the string is a numeric string, False␣
↪otherwise.
 |
 |      A string is numeric if all characters in the string are␣
↪numeric and there is at
 |      least one character in the string.
 |
 |  isprintable(self, /)
 |      Return True if the string is printable, False otherwise.
 |
 |      A string is printable if all of its characters are␣
↪considered printable in
 |      repr() or if it is empty.
 |
 |  isspace(self, /)
 |      Return True if the string is a whitespace string, False␣
↪otherwise.
 |
 |      A string is whitespace if all characters in the string are␣
↪whitespace and there
 |      is at least one character in the string.
```

(continues on next page)

(continued from previous page)

```
 |
 |  istitle(self, /)
 |      Return True if the string is a title-cased string, False␣
↪otherwise.
 |
 |      In a title-cased string, upper- and title-case characters␣
↪may only
 |      follow uncased characters and lowercase characters only␣
↪cased ones.
 |
 |  isupper(self, /)
 |      Return True if the string is an uppercase string, False␣
↪otherwise.
 |
 |      A string is uppercase if all cased characters in the␣
↪string are uppercase and
 |      there is at least one cased character in the string.
 |
 |  join(self, iterable, /)
 |      Concatenate any number of strings.
 |
 |      The string whose method is called is inserted in between␣
↪each given string.
 |      The result is returned as a new string.
 |
 |      Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
 |
 |  ljust(self, width, fillchar=' ', /)
 |      Return a left-justified string of length width.
 |
 |      Padding is done using the specified fill character␣
↪(default is a space).
 |
 |  lower(self, /)
 |      Return a copy of the string converted to lowercase.
 |
 |  lstrip(self, chars=None, /)
 |      Return a copy of the string with leading whitespace␣
↪removed.
 |
 |      If chars is given and not None, remove characters in chars␣
↪instead.
 |
 |  partition(self, sep, /)
 |      Partition the string into three parts using the given␣
↪separator.
```

(continues on next page)

(continued from previous page)

```
 |
 |      This will search for the separator in the string.  If the␣
↪separator is found,
 |      returns a 3-tuple containing the part before the separator,
↪ the separator
 |      itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple␣
↪containing the original string
 |      and two empty strings.
 |
 |  removeprefix(self, prefix, /)
 |      Return a str with the given prefix string removed if␣
↪present.
 |
 |      If the string starts with the prefix string, return␣
↪string[len(prefix):].
 |      Otherwise, return a copy of the original string.
 |
 |  removesuffix(self, suffix, /)
 |      Return a str with the given suffix string removed if␣
↪present.
 |
 |      If the string ends with the suffix string and that suffix␣
↪is not empty,
 |      return string[:-len(suffix)]. Otherwise, return a copy of␣
↪the original
 |      string.
 |
 |  replace(self, old, new, count=-1, /)
 |      Return a copy with all occurrences of substring old␣
↪replaced by new.
 |
 |        count
 |          Maximum number of occurrences to replace.
 |          -1 (the default value) means replace all occurrences.
 |
 |      If the optional argument count is given, only the first␣
↪count occurrences are
 |      replaced.
 |
 |  rfind(...)
 |      S.rfind(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found,
```

(continues on next page)

(continued from previous page)

```
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice␣
↪notation.
 |
 |      Return -1 on failure.
 |
 |  rindex(...)
 |      S.rindex(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice␣
↪notation.
 |
 |      Raises ValueError when the substring is not found.
 |
 |  rjust(self, width, fillchar=' ', /)
 |      Return a right-justified string of length width.
 |
 |      Padding is done using the specified fill character␣
↪(default is a space).
 |
 |  rpartition(self, sep, /)
 |      Partition the string into three parts using the given␣
↪separator.
 |
 |      This will search for the separator in the string, starting␣
↪at the end. If
 |      the separator is found, returns a 3-tuple containing the␣
↪part before the
 |      separator, the separator itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple␣
↪containing two empty strings
 |      and the original string.
 |
 |  rsplit(self, /, sep=None, maxsplit=-1)
 |      Return a list of the words in the string, using sep as the␣
↪delimiter string.
 |
 |        sep
 |          The delimiter according which to split the string.
 |          None (the default value) means split according to any␣
↪whitespace,
 |          and discard empty strings from the result.
```

(continues on next page)

(continued from previous page)

```
|         maxsplit
|           Maximum number of splits to do.
|           -1 (the default value) means no limit.
|
|       Splits are done starting at the end of the string and␣
↪working to the front.
|
|   rstrip(self, chars=None, /)
|       Return a copy of the string with trailing whitespace␣
↪removed.
|
|       If chars is given and not None, remove characters in chars␣
↪instead.
|
|   split(self, /, sep=None, maxsplit=-1)
|       Return a list of the words in the string, using sep as the␣
↪delimiter string.
|
|       sep
|         The delimiter according which to split the string.
|         None (the default value) means split according to any␣
↪whitespace,
|         and discard empty strings from the result.
|       maxsplit
|         Maximum number of splits to do.
|         -1 (the default value) means no limit.
|
|   splitlines(self, /, keepends=False)
|       Return a list of the lines in the string, breaking at line␣
↪boundaries.
|
|       Line breaks are not included in the resulting list unless␣
↪keepends is given and
|       true.
|
|   startswith(...)
|       S.startswith(prefix[, start[, end]]) -> bool
|
|       Return True if S starts with the specified prefix, False␣
↪otherwise.
|       With optional start, test S beginning at that position.
|       With optional end, stop comparing S at that position.
|       prefix can also be a tuple of strings to try.
|
|   strip(self, chars=None, /)
```

(continues on next page)

(continued from previous page)

```
|      Return a copy of the string with leading and trailing␣
↪whitespace removed.
|
|      If chars is given and not None, remove characters in chars␣
↪instead.
|
|  swapcase(self, /)
|      Convert uppercase characters to lowercase and lowercase␣
↪characters to uppercase.
|
|  title(self, /)
|      Return a version of the string where each word is␣
↪titlecased.
|
|      More specifically, words start with uppercased characters␣
↪and all remaining
|      cased characters have lower case.
|
|  translate(self, table, /)
|      Replace each character in the string using the given␣
↪translation table.
|
|        table
|          Translation table, which must be a mapping of Unicode␣
↪ordinals to
|          Unicode ordinals, strings, or None.
|
|      The table must implement lookup/indexing via __getitem__,␣
↪for instance a
|      dictionary or list.  If this operation raises LookupError,␣
↪the character is
|      left untouched.  Characters mapped to None are deleted.
|
|  upper(self, /)
|      Return a copy of the string converted to uppercase.
|
|  zfill(self, width, /)
|      Pad a numeric string with zeros on the left, to fill a␣
↪field of the given width.
|
|      The string is never truncated.
|
|  ---------------------------------------------------------------
↪-------
|  Static methods defined here:
```

(continues on next page)

(continued from previous page)

```
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for␣
↪accurate signature.
 |
 |  maketrans(...)
 |      Return a translation table usable for str.translate().
 |
 |      If there is only one argument, it must be a dictionary␣
↪mapping Unicode
 |      ordinals (integers) or characters to Unicode ordinals,␣
↪strings or None.
 |      Character keys will be then converted to ordinals.
 |      If there are two arguments, they must be strings of equal␣
↪length, and
 |      in the resulting dictionary, each character in x will be␣
↪mapped to the
 |      character at the same position in y. If there is a third␣
↪argument, it
 |      must be a string, whose characters will be mapped to None␣
↪in the result.
```

CHAPTER

THREE

LAB 1.1 (DEADLINE FRIDAY 21 JANUARY 23:59)

- Upload your code to Einstein to have it verified.

3.1 Chop

- Write a Python program called `chop_011.py` that reads lines of text from `stdin`. Each line consists of a single string. The program should print out the string minus its first and last characters. If there is nothing left after removing the first and last characters the program should not print anything. For example:

```
$ cat chop_stdin_00_011.txt
Jimmy
ran
to
a
standstill
```

```
$ python3 chop_011.py < chop_stdin_00_011.txt
imm
a
tandstil
```

3.2 Capitals

- Write a program called `capitals_011.py` that reads lines of text from `stdin`. Each line consists of a single string. The program should capitalise the first and last characters of the string and print the result. If the string has fewer than two characters the program should print nothing. For example:

```
$ cat capitals_stdin_00_011.txt
mittens
mit
m
pi
pittance
```

```
$ python3 capitals_011.py < capitals_stdin_00_011.txt
MittenS
MiT
PI
PittancE
```

3.3 Middle

- Write a program called `middle_011.py` that reads lines of text from `stdin`. Each line consists of a single string. The program should print out the middle character of each string. If the string does not have a middle character the program should print "No middle character!". For example:

```
$ cat middle_stdin_00_011.txt
marshmallow
pip
p
pips
zingy
```

```
$ python3 middle_011.py < middle_stdin_00_011.txt
m
i
p
No middle character!
n
```

3.4 Substring

- Write a program called `substring_011.py` that reads lines of text from `stdin`. Each line consists of two strings. The program should print `True` if the first string is a substring of the second (and `False` otherwise). Note that differences in case can be ignored. For example:

```
$ cat substring_stdin_00_011.txt
Rump Rumpelstiltskin
rump Rumpelstiltskin
stilt Rumpelstiltskin
stiLT Rumpelstiltskin
pest Rumpelstiltskin
up Rumpelstiltskin
```

```
$ python3 substring_011.py < substring_stdin_00_011.txt
True
True
True
True
False
False
```

3.5 Contains

- Write a program called `contains_011.py` that reads lines of text from `stdin`. Each line consists of two strings. The program should print `True` if each of the characters in the first string is also in the second string (and `False` otherwise). Note that once a character has been matched in the second string it cannot be matched again. Also note that differences in case can be ignored. For example:

```
$ cat contains_stdin_00_011.txt
c cat
AC cat
tac caT
ttac cat
```

```
$ python3 contains_011.py < contains_stdin_00_011.txt
True
True
True
False
```

- Hint: You may find the `str.replace()` method useful. Use `help` or `pydoc` to look it up.

3.6 Emails

- Write a program called `emails_011.py` that reads DCU student email addresses from `stdin`. For each email address the program should print out the corresponding student's name. For example:

```
$ cat emails_stdin_00_011.txt
valerie.maguire2@mail.dcu.ie
fred.quinn33@mail.dcu.ie
jimmy.clancy5@mail.dcu.ie
```

```
$ python3 emails_011.py < emails_stdin_00_011.txt
Valerie Maguire
Fred Quinn
Jimmy Clancy
```

3.7 First M

- Write a program called `firstm_011.py` that reads lines of text from `stdin`. The program should print each line with the first word that begins with a lower case `m` now capitalized. For example:

```
$ cat firstm_stdin_00_011.txt
Mickey Mouse was a kind of mouse.
Mickey Mouse was a kind of mouse. A mouse with a sense of humour.
```

```
$ python3 firstm_011.py < firstm_stdin_00_011.txt
Mickey Mouse was a kind of Mouse.
Mickey Mouse was a kind of Mouse. A mouse with a sense of humour.
```

3.8 Water

- You have some water and some buckets to fill.
- Write a program called `water_011.py` that reads two lines of text from `stdin`.
- Line 1 contains a single integer, N, the number of litres of water available. N is in the range 0-1000.
- Line 2 lists the capacity in litres of one or more buckets. The capacity of each bucket is specified by a positive integer.
- Buckets must be filled in the order specified on line 2.

- Your program should output the number of buckets that can be completely filled before you run out of water.
- In this example we have 10 litres of water. We fill the first bucket (taking 6 litres), we fill the second bucket (taking another 2 litres) but we run out of water before we have completely filled the third bucket (it requires 5 litres). We output 2 (the number of buckets completely filled):

```
$ cat water_stdin_00_011.txt
10
6 2 5 1 1
```

```
$ python3 water_011.py < water_stdin_00_011.txt
2
```

CHAPTER

FOUR

LECTURE 1.3 : FORMATTED STRING OUTPUT

4.1 Introduction

- So far we have been using `print()` to send output to `stdout`.

```
x = 1/3
print(x)

from math import pi
print(pi)
```

```
0.3333333333333333
3.141592653589793
```

- Unfortunately `print()` alone affords us little control over the *format* of the printed string. For instance, what if we want to display a floating point number to some specific number of decimal places? If we calculate an average price, for example, it would not make sense to go beyond two decimal places when displaying it. The `print()` function alone will not allow us to do that. We need a more sophisticated approach.

4.2 f-strings

- We can use *f-strings* to format a string prior to printing.
- An f-string is similar to a normal string except it starts with an f (surprise!) and contains *placeholders* for the data we wish to be specially formatted.
- Inside the placeholders we specify *which* data we want displayed and *how* we want it to be displayed.

```
print(f'{x}')
print(f'{pi}')
```

```
0.3333333333333333
3.141592653589793
```

- Each f-string above contains a single placeholder (a placeholder is delimited by curly brackets). Into each placeholder we insert the name of a variable whose value we wish to print. Above we are printing `x` and `pi`.
- So far our f-strings have not yielded results different to those produced using the `print()` function. The power of f-strings however comes from the *format commands* that can be placed inside the placeholders. They control *how* the data inserted into those placeholders is displayed.
- The general structure of a *format command* is `{[:[align] [minimum_width] [.precision] [type]]}`. Square brackets indicate optional arguments.
- The `align` field is used to control whether the printed value is centred, left justified or right justified. Values include ^ for centred, < for left justified and > for right justified.
- The `type` field specifies the type of value we are printing. Commonly used types include `s` (for string), `d` (for integer) and `f` (for floating point).
- The `minimum_width` field specifies the desired *overall* minimum width for this value once printed.
- The `.precision` field specifies the number of digits to follow the decimal point when printing a floating point type.
- This all sounds more complicated than it is. Let's look at some examples in order to make things clearer.
- To print `x` to two decimal places and `pi` to five decimal places we would write:

```
print(f'{x:.2f}')
print(f'{pi:.5f}')
```

```
0.33
3.14159
```

- By specifying a minimum width we can cause leading spaces to be inserted in order to pad the number out to an overall width that equals the minimum width (there will be no padding if the width of the number already equals or exceeds the minimum width):

```
print(f'{x:8.2f}')
print(f'{pi:1.5f}')
```

```
    0.33
3.14159
```

- Sometimes we want to pad with zeros rather than spaces.

```
hour, min, sec = 3, 4, 5
print(f'The current time is {hour:02d}:{min:02d}:{sec:02d}')

hour, min, sec = 13, 14, 15
print(f'The current time is {hour:02d}:{min:02d}:{sec:02d}')
```

```
The current time is 03:04:05
The current time is 13:14:15
```

- We can include as many placeholders in the f-string as we wish.

```
print(f'x has the value {x:.2f} and pi has the value {pi:.5f}')
```

```
x has the value 0.33 and pi has the value 3.14159
```

- Suppose we want to print our times 12 multiplication table. We could do it like this but the output is not nicely aligned (which we find upsetting):

```
for i in range(1, 13):
   print(str(i) + ' * 12 = ' + str(i*12))
```

```
1 * 12 = 12
2 * 12 = 24
3 * 12 = 36
4 * 12 = 48
5 * 12 = 60
6 * 12 = 72
7 * 12 = 84
8 * 12 = 96
9 * 12 = 108
10 * 12 = 120
11 * 12 = 132
12 * 12 = 144
```

- Specifying minimum widths is useful when we want to align output. If we write it like this then the output is neatly aligned (because all numbers are right-justified and padded out to the specified minimum width):

```
for i in range(1, 13):
   print(f'{i:2d} * 12 = {i*12:3d}')
```

```
 1 * 12 =  12
 2 * 12 =  24
 3 * 12 =  36
 4 * 12 =  48
 5 * 12 =  60
 6 * 12 =  72
 7 * 12 =  84
 8 * 12 =  96
 9 * 12 = 108
10 * 12 = 120
11 * 12 = 132
12 * 12 = 144
```

4.3 Nested placeholders

- Suppose we want to print `pi` to some user-defined number of decimal places. Can we do that? Yes, with nested placeholders. Below we first print `pi` to 3 decimal places. Next we replace the 3 in the f-string with a placeholder. This allows us to insert at runtime an arbitrary value for the precision.

```
print(f'{pi:.3f}')
N = 5
print(f'{pi:.{N}f}')
```

```
3.142
3.14159
```

- Below we use this technique to display `pi` to various decimal places inside a `for` loop.

```
for i in range(10):
    print(f'{pi:.{i}f}')
```

```
3
3.1
3.14
3.142
3.1416
3.14159
3.141593
3.1415927
3.14159265
3.141592654
```

CHAPTER

FIVE

LAB 1.2 (DEADLINE FRIDAY 21 JANUARY 23:59)

- Upload your code to Einstein to have it verified.

5.1 Password security

- Password security is a problem when users choose passwords that can be easily guessed. Write a Python program that assesses the security of a password by counting the number of character classes it contains. For our purposes there are four character classes: digits, lower case characters, upper case characters and special characters (i.e. everything else).
- Write a program called `password_012.py` that reads passwords from `stdin`. For each password read the program should print out the number of character classes it contains. For example:

```
$ cat password_stdin_00_012.txt
256
abc
aBc
1aBc2
^@())($$$
^@a1())B($43$$
```

```
$ python3 password_012.py < password_stdin_00_012.txt
1
1
2
3
1
4
```

- Hint: Use `pydoc` or `help` to have a look at the `str` class documentation. You will find described therein various methods that will be useful in determining the class of each character in the string.

5.2 Plural

- Write a program called `plural_012.py` that reads nouns from `stdin`. For each noun read the program should print its plural according to the following rules:
  - Add *es* if the noun ends in *ch*, *sh*, *x*, *s* or *z*.
  - If a noun ends in a consonant + *y* drop the *y* and add *ies*.
  - If a noun ends in *f* (or *fe*) drop the *f* (or *fe*) and add *ves*.
  - If a noun ends in *o* add *es*.
  - Otherwise add *s*.

  For example:

```
$ cat plural_stdin_00_012.txt
peach
wife
bay
dish
box
fuss
fuzz
banjo
dainty
toy
self
nut
```

```
$ python3 plural_012.py < plural_stdin_00_012.txt
peaches
wives
bays
dishes
boxes
fusses
fuzzes
banjoes
dainties
toys
selves
nuts
```

5.3 Poetry

- Write a program called `poetry_012.py` that reads in the lines of a poem from `stdin` and uses f-strings to output a centred version. For example:

```
$ cat poetry_stdin_00_012.txt
Sonnet 98
by William Shakespeare

From you have I been absent in the spring,
When proud-pied April dress'd in all his trim
Hath put a spirit of youth in every thing,
That heavy Saturn laugh'd and leap'd with him.
Yet nor the lays of birds nor the sweet smell
Of different flowers in odour and in hue
Could make me any summer's story tell,
Or from their proud lap pluck them where they grew;
Nor did I wonder at the lily's white,
Nor praise the deep vermilion in the rose;
They were but sweet, but figures of delight,
Drawn after you, you pattern of all those.
Yet seem'd it winter still, and, you away,
As with your shadow I with these did play.
```

```
$ python3 poetry_012.py < poetry_stdin_00_012.txt
                     Sonnet 98
              by William Shakespeare

    From you have I been absent in the spring,
   When proud-pied April dress'd in all his trim
    Hath put a spirit of youth in every thing,
  That heavy Saturn laugh'd and leap'd with him.
   Yet nor the lays of birds nor the sweet smell
     Of different flowers in odour and in hue
      Could make me any summer's story tell,
Or from their proud lap pluck them where they grew;
       Nor did I wonder at the lily's white,
    Nor praise the deep vermilion in the rose;
   They were but sweet, but figures of delight,
    Drawn after you, you pattern of all those.
    Yet seem'd it winter still, and, you away,
    As with your shadow I with these did play.
```

5.4 Pi

- Write a program called `pi_012.py` that reads integers from `stdin` and, for each integer read, uses an f-string to print pi to that number of decimal places.
- Note you will have to import the `math` module to get access to the `math.pi` constant. For example:

```
$ cat pi_stdin_00_012.txt
1
2
3
4
10
```

```
$ python3 pi_012.py < pi_stdin_00_012.txt
3.1
3.14
3.142
3.1416
3.1415926536
```

5.5 League table

- Write a program called `league_012.py` that reads in the lines of a league table from `stdin` and uses f-strings to display them in neatly tabulated columns. For example:

```
$ cat league_stdin_00_012.txt
1 Spurs 11 7 3 1 23 9 14 24
2 Liverpool 11 7 3 1 26 17 9 24
3 Chelsea 11 6 4 1 25 11 14 22
4 Leicester 11 7 0 4 21 15 6 21
5 Man Utd 10 6 1 3 19 17 2 19
6 Man City 10 5 3 2 17 11 6 18
7 West Ham 11 5 2 4 18 14 4 17
8 Southampton 10 5 2 3 19 16 3 17
9 Everton 11 5 2 4 20 18 2 17
10 Wolves 11 5 2 4 11 15 -4 17
11 C Palace 11 5 1 5 17 16 1 16
12 Aston Villa 9 5 0 4 20 13 7 15
13 Newcastle 10 4 2 4 12 15 -3 14
14 Leeds 11 4 2 5 16 20 -4 14
15 Arsenal 11 4 1 6 10 14 -4 13
16 Brighton 10 2 4 4 14 16 -2 10
```

(continues on next page)

(continued from previous page)

```
17 Fulham 11 2 1 8 11 21 -10 7
18 Burnley 10 1 3 6 5 18 -13 6
19 West Brom 11 1 3 7 8 23 -15 6
20 Sheff Utd 11 0 1 10 5 18 -13 1
```

```
$ python3 league_012.py < league_stdin_00_012.txt
POS CLUB         P   W   D   L  GF  GA  GD PTS
  1 Spurs       11   7   3   1  23   9  14  24
  2 Liverpool   11   7   3   1  26  17   9  24
  3 Chelsea     11   6   4   1  25  11  14  22
  4 Leicester   11   7   0   4  21  15   6  21
  5 Man Utd     10   6   1   3  19  17   2  19
  6 Man City    10   5   3   2  17  11   6  18
  7 West Ham    11   5   2   4  18  14   4  17
  8 Southampton 10   5   2   3  19  16   3  17
  9 Everton     11   5   2   4  20  18   2  17
 10 Wolves      11   5   2   4  11  15  -4  17
 11 C Palace    11   5   1   5  17  16   1  16
 12 Aston Villa  9   5   0   4  20  13   7  15
 13 Newcastle   10   4   2   4  12  15  -3  14
 14 Leeds       11   4   2   5  16  20  -4  14
 15 Arsenal     11   4   1   6  10  14  -4  13
 16 Brighton    10   2   4   4  14  16  -2  10
 17 Fulham      11   2   1   8  11  21 -10   7
 18 Burnley     10   1   3   6   5  18 -13   6
 19 West Brom   11   1   3   7   8  23 -15   6
 20 Sheff Utd   11   0   1  10   5  18 -13   1
```

- There is more work in this than might first appear:
  1. The width of the CLUB field depends on the list of clubs supplied to the program: it should be just wide enough to accommodate the longest club name (above that would be Southampton). This means you will have to do some preprocessing on the list read from `stdin` before you can print it.
  2. The P field is 2 characters wide. All other fields are 3 characters wide. (Not including the space between fields.)
  3. Extracting the club name will require some ingenuity. Hint: if each line is converted to a list of tokens then the first word in the club name is at a fixed index from the left while the last word in the club name is at a fixed index from the right. Thus a specially crafted slice over the tokens in each line should grab the club name.

CHAPTER

SIX

LECTURE 2.1 LISTS

6.1 Introduction

- Python's built-in list type is a *collection type* (meaning it contains a number of objects that can be treated as one object). It is also a *sequence type* meaning each object in the collection occupies a specific numbered location within it i.e. elements are *ordered*. The list is an *iterable type* meaning we can use a loop to inspect each of its elements in turn.
- So far a list seems similar to a string. However a list differs in two significant ways:
  1. A list can contain objects of *differing* and *arbitrary* type. (A string is made up entirely of objects of the same type, namely, characters.)
  2. A list is a *mutable* type. This means it can be modified after initialisation. (A string is an *immutable* type. It cannot be modified after creation.)

6.2 Indexing and slicing lists

- As with strings, to select a particular element in a list we index into it using square brackets. The first element of the list is located at index zero. The last element is at index N-1 in a list of length N.
- Slicing and extended slicing work exactly as they do for strings. (This makes sense as both lists and strings are sequence types.)

6.3 Lists in action

- Below we explore some list properties and demonstrate associated methods:

```
# Create a list with []
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

```
# Reverse a list with extended slicing
print(days[::-1])
```

```
['Friday', 'Thursday', 'Wednesday', 'Tuesday', 'Monday']
```

```
# Add to a list
days.append('Saturday')
days.append('Sunday')
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday
↪', 'Sunday']
```

```
# Check list membership
print('Friday' in days)
print('April' in days)
```

```
True
False
```

```
# Find the index of a list element
print(days.index('Tuesday'))
```

```
1
```

```
# Remove and return an item by index
i = days.index('Tuesday')
print(days.pop(i))
print(days)
```

```
Tuesday
['Monday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

```
# Remove first occurrence of an item by value (without returning␣
↪it)
days.remove('Wednesday')
```

(continues on next page)

(continued from previous page)

```
print(days)
```

```
['Monday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

```
# Insert an item at a particular index (bumping elements to the␣
↪right)
days.insert(1, 'Tuesday')
days.insert(2, 'Wednesday')
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday
↪', 'Sunday']
```

```
# Remove and return the last item in the list
print(days.pop())
print(days)
```

```
Sunday
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday
↪']
```

```
# Delete an item by index from the list (without returning it)
del(days[-1])
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

```
# Combine two lists with extend
days.extend(['Saturday', 'Sunday'])
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday
↪', 'Sunday']
```

```
# Combine a list of strings into a single string
print(' '.join(days))
```

```
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

```
# Update each element (proving lists are mutable)
i = 0
N = len(days)
```

(continues on next page)

(continued from previous page)

```
while i < N:
    days[i] = days[i].lower()
    i += 1
print(days)
```

```
['monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday
↪', 'sunday']
```

```
# Iterate over each element with a for loop to build a new list
capdays = list()
for day in days:
    capdays.append(day.capitalize())
print(capdays)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday
↪', 'Sunday']
```

```
# Sort a list in place
days.sort()
print(days)
```

```
['friday', 'monday', 'saturday', 'sunday', 'thursday', 'tuesday',
↪'wednesday']
```

6.4 Lists are `True` or `False`

- The empty list `[]` is interpreted as `False`.
- Any non-empty list is `True`.
- Because lists have truth values we can write code like this:

```
while (capdays):
    print(capdays.pop())
```

```
Sunday
Saturday
Friday
Thursday
Wednesday
Tuesday
Monday
```

6.5 List concatenation, replication, copying, comparison

```
# Create two lists
alist = [1, 2, 3]
blist = [4, 5, 6]

# Extend alist (with the contents of blist)
alist += blist
print(alist)

# Replicate blist
blist *= 3
print(blist)

# Make clist a copy of alist and compare
clist = alist[:]
print(clist == alist)
```

```
[1, 2, 3, 4, 5, 6]
[4, 5, 6, 4, 5, 6, 4, 5, 6]
True
```

6.6 Lists of lists

- A list can contain objects of any type, even other lists. Lists of lists are useful for representing many data types e.g. spreadsheets, matrices, images, etc.
- To select a particular element in a nested (i.e. embedded) list we first select the embedded list and then select the element. Each of these selection operations requires the use of square brackets:

```
lol = [ [1, 2, 3], ['a', 'b', 'c'] ]
print(lol[0])
print(lol[1])
print(lol[0][-1])
print(lol[1][1])
```

```
[1, 2, 3]
['a', 'b', 'c']
3
b
```

6.7 From strings to lists and back again

- Suppose every student's list of marks is available as a string e.g. `"Mary Rose O'Reilly-McCann 40 45 60 70 55"`
- We want to replace each student's set of marks with their min, max and average mark. How might we go about it?
- The length of each student's name is variable but the number of marks is fixed. We can take advantage of that.

```
line_in = "Mary Rose O'Reilly-McCann 40 45 60 70 55"
NUM_MARKS = 5

# Split the line into its constituent tokens
tokens = line_in.strip().split()
print(tokens)

# Extract the list of marks
marks_as_strings = tokens[-NUM_MARKS:]
print(marks_as_strings)

# Convert marks to integers
marks_as_ints = list()
for mark in marks_as_strings:
   marks_as_ints.append(int(mark))
print(marks_as_ints)

# Extract min, max and calculate average
min_mark = min(marks_as_ints)
max_mark = max(marks_as_ints)
avg_mark = sum(marks_as_ints) // NUM_MARKS

# Convert new marks to strings and put in a list
marks_as_strings = [str(min_mark), str(max_mark), str(avg_mark)]
print(marks_as_strings)

# Combine our two lists to rebuild tokens
tokens = tokens[:-NUM_MARKS] + marks_as_strings
print(tokens)

# Turn our list of strings into one string
print(' '.join(tokens))

# Phew!
```

```
['Mary', 'Rose', "O'Reilly-McCann", '40', '45', '60', '70', '55']
['40', '45', '60', '70', '55']
[40, 45, 60, 70, 55]
['40', '70', '54']
['Mary', 'Rose', "O'Reilly-McCann", '40', '70', '54']
Mary Rose O'Reilly-McCann 40 70 54
```

6.8 Multiple assignment

- Lists provide us with the opportunity to highlight a handy python feature called multiple assignment.

```python
# Here is a list with three elements
print(marks_as_strings)

# We can assign a name to each list element in a single line of␣
↪code
[min_mark, max_mark, avg_mark] = marks_as_strings

# Check it
print(f'min_mark: {min_mark}')
print(f'max_mark: {max_mark}')
print(f'avg_mark: {avg_mark}')
```

```
['40', '70', '54']
min_mark: 40
max_mark: 70
avg_mark: 54
```

- This is also sometimes referred to as list unpacking.

6.9 List methods

- Python comes with built-in support for a set of common list operations. These operations are called `methods` and they define the things we can do with lists. Calling `help(list)` or `pydoc list` outputs a list of these methods.
- Note that because lists are *mutable* calling a method on a list may alter the list itself. (Contrast this behaviour with that of string methods.)
- Whenever you find it necessary to carry out some list processing first look up the available built-in list methods. There may be one that will help you with your task. There is no point writing your own code that duplicates what a built-in list method can do for you already.

```
help(list)
```

```
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty␣
↪list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
|      Initialize self.  See help(type(self)) for accurate␣
↪signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __reversed__(self, /)
|      Return a reverse iterator over the list.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(self, /)
|      Return the size of the list in memory, in bytes.
|
|  append(self, object, /)
|      Append object to the end of the list.
|
|  clear(self, /)
|      Remove all items from list.
|
|  copy(self, /)
|      Return a shallow copy of the list.
|
```

(continues on next page)

(continued from previous page)

```
|  count(self, value, /)
|      Return number of occurrences of value.
|
|  extend(self, iterable, /)
|      Extend list by appending elements from the iterable.
|
|  index(self, value, start=0, stop=9223372036854775807, /)
|      Return first index of value.
|
|      Raises ValueError if the value is not present.
|
|  insert(self, index, object, /)
|      Insert object before index.
|
|  pop(self, index=-1, /)
|      Remove and return item at index (default last).
|
|      Raises IndexError if list is empty or index is out of␣
↪range.
|
|  remove(self, value, /)
|      Remove first occurrence of value.
|
|      Raises ValueError if the value is not present.
|
|  reverse(self, /)
|      Reverse *IN PLACE*.
|
|  sort(self, /, *, key=None, reverse=False)
|      Sort the list in ascending order and return None.
|
|      The sort is in-place (i.e. the list itself is modified)␣
↪and stable (i.e. the
|      order of two equal elements is maintained).
|
|      If a key function is given, apply it once to each list␣
↪item and sort them,
|      ascending or descending, according to their function␣
↪values.
|
|      The reverse flag can be set to sort in descending order.
|
|  ---------------------------------------------------------------
↪-------
|  Class methods defined here:
```

(continues on next page)

(continued from previous page)

```
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  ---------------------------------------------------------------
↪-------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for␣
↪accurate signature.
 |
 |  ---------------------------------------------------------------
↪-------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

CHAPTER

SEVEN

LECTURE 2.2 : TUPLES

7.1 Introduction

- A *tuple* is essentially an *immutable* list.
- Like a list, a tuple is a sequenced collection type. Like a list it can accommodate a collection of arbitrary types.
- Like a string, a tuple is immutable.

7.2 Tuple creation

- We create a tuple using the comma operator and, though not strictly necessary, we typically surround the tuple with round brackets to make it obvious it's a tuple.

```
t = (4, 5, 6)
print(t)
```

```
(4, 5, 6)
```

7.3 Immutability

- What does immutable mean in the context of a tuple? It means that once constructed the *top-level contents* of a tuple cannot be modified.

```
print(t[0])
# We cannot change the top-level contents of a tuple
t[0] += 1
```

```
4
```

```
---------------------------------------------------------------------------
↪--------
TypeError                                 Traceback (most recent␣
↪call last)
/tmp/ipykernel_6553/2919204386.py in <module>
      1 print(t[0])
      2 # We cannot change the top-level contents of a tuple
----> 3 t[0] += 1

TypeError: 'tuple' object does not support item assignment
```

- Is this changing the contents of tuple? No. It is creating a *new* tuple from the contents of two existing tuples (it just so happens we assign the name of an existing tuple to the new one):

```
t = ('a', 'b', 'c')
t += ('d', 'e', 'f')
print(t)
```

```
('a', 'b', 'c', 'd', 'e', 'f')
```

- Why do we say that the *top-level* contents of a tuple cannot be changed rather than simply saying that the contents of a tuple cannot be changed? We refer specifically to the top-level contents in order to make clear that although the contents are immutable, the contents of the contents of a tuple are not necessarily immutable.

```
t = (['a', 'b', 'c'], ['cat', 'dog'])
t[1].append('fish')
print(t)
```

```
(['a', 'b', 'c'], ['cat', 'dog', 'fish'])
```

7.4 Named tuples

- A special case of a tuple is a *named tuple*. Related data can be grouped together as a set of attribute-value pairs to form a named tuple
- Suppose for example that we wish to model a car. A car has several attributes including a make, model and age. We can use a named tuple as follows to represent a single `Car` data type that has each of these attributes:

```
from collections import namedtuple

# Create a new data type that is named tuple
Car = namedtuple('Car', ['make', 'model', 'age'])
car1 = Car('Opel', 'Astra', 3)
car2 = Car('Mazda', 'MX5', 7)
print(f'{car1.make} {car1.model} {car1.age}')
print(f'{car2.make} {car2.model} {car2.age}')
```

```
Opel Astra 3
Mazda MX5 7
```

7.5 Uses of tuples

- Because they are immutable we often use tuples to store constants or values that we do not want our program to ever change.
- As we'll see later, when a function has multiple values to return to its caller, it will typically return them in a tuple
- Only immutables can be serve as dictionary keys and, being immutable, tuples can be used in this context.
- We can take advantage of tuples and multiple assignment to swap two values without using a temporary variable:

```
a = 3
b = 7
print(f'a={a}, b={b}')
(b, a) = (a, b)
print(f'a={a}, b={b}')
```

```
a=3, b=7
a=7, b=3
```

7.6 Tuple methods

- Apart from in those respects listed above, tuples behave similarly to lists and support the same indexing, slicing, concatenation, iteration etc. operations.

```
help(tuple)
```

```
Help on class tuple in module builtins:

class tuple(object)
 |  tuple(iterable=(), /)
 |
 |  Built-in immutable sequence.
 |
 |  If no argument is given, the constructor returns an empty␣
↪tuple.
 |  If iterable is specified the tuple is initialized from iterable
↪'s items.
 |
 |  If the argument is a tuple, the return value is the same␣
↪object.
 |
 |  Built-in subclasses:
 |      asyncgen_hooks
 |      UnraisableHookArgs
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
```

(continues on next page)

(continued from previous page)

```
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------------
↪-------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
```

(continues on next page)

(continued from previous page)

```
|      See PEP 585
|
|  ----------------------------------------------------------------
↪-------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for␣
↪accurate signature.
```

CHAPTER

EIGHT

LAB 2.1 (DEADLINE FRIDAY 28 JANUARY 23:59)

- Upload your code to Einstein to have it verified.

8.1 Anagrams

- Two words are anagrams if the letters of one word can be rearranged to form the other word. For example *angel* and *glean* are anagrams. Write a Python program called `anagram_021.py` that reads in pairs of words (one pair per line) from `stdin` and prints `True` if the pair are anagrams and `False` otherwise. For example:

```
$ cat anagram_stdin_00_021.txt
cinema iceman
dog god
house car
stub buts
angel glean
aangl angel
a aardvark
aardvark a
```

```
$ python3 anagram_021.py < anagram_stdin_00_021.txt
True
True
False
True
True
False
False
False
```

8.2 Palindromes

- A palindrome is a word, phrase, number or other sequence of characters which reads the same backwards as forwards. Allowances are made for capital letters, punctuation and white space (word dividers). Write a program called `palindrome_021.py` that reads lines of text from `stdin` and prints `True` if the line is a palindrome and `False` otherwise. For example:

```
$ cat palindrome_stdin_00_021.txt
Step on no pets.
Step on no cats.
Able was I ere I saw Elba.
Doc, note: I dissent. A fast never prevents a fatness. I diet on␣
↪cod.
Bananas
Abel was I ere
A
Aa
ABa
123.
1221.
4444444.
This is not a palindrome.
```

```
$ python3 palindrome_021.py < palindrome_stdin_00_021.txt
True
False
True
True
False
False
True
True
True
False
True
True
False
```

- Hints:
  1. Convert the string to lowercase first. Use `pydoc` to check out the `str` class documentation. You need to find a method that will do the conversion for you.
  2. You need to strip out any characters which are not alphanumeric. Again, use `pydoc` to check out the `str` class documentation to find methods that will help you.

3. Once you have the string in canonical form (i.e. in lowercase with all non-alphanumeric characters removed) then simply check whether it is the same sequence backwards as forwards.

8.3 Unique word count

- Write a program called `unique_021.py` that counts the total number of unique words in lines of text read from `stdin`. Running the program against gettysburg.txt should produce the following output:

```
$ python3 unique_021.py < gettysburg.txt
143
```

- Hints:
  1. You will have to remove *surrounding* punctuation in the text. For example *house* and *house.* should not be counted as separate unique words. For this task you may find `string.punctuation` useful.
  2. You will have to cater for upper and lower case versions of words. For example *Four* and *four* should not be counted as separate unique words.
  3. Only alphanumeric tokens are to be counted as words. For example *November* and *19* are words but – is not.
  4. Here is the sorted list of what my code considers a unique word:

```
['1863', '19', 'a', 'above', 'abraham', 'add', 'advanced', 'ago
↪', 'all', 'altogether', 'and', 'any', 'are', 'as', 'battle-
↪field', 'be', 'before', 'birth', 'brave', 'brought', 'but',
↪'by', 'can', 'cause', 'civil', 'come', 'conceived',
↪'consecrate', 'consecrated', 'continent', 'created', 'dead',
↪'dedicate', 'dedicated', 'detract', 'devotion', 'did', 'died
↪', 'do', 'earth', 'endure', 'engaged', 'equal', 'far',
↪'fathers', 'field', 'final', 'fitting', 'for', 'forget',
↪'forth', 'fought', 'four', 'freedom', 'from', 'full', 'gave',
↪ 'god', 'government', 'great', 'ground', 'hallow', 'have',
↪'here', 'highly', 'honored', 'in', 'increased', 'is', 'it',
↪'larger', 'last', 'liberty', 'lincoln', 'little', 'live',
↪'lives', 'living', 'long', 'measure', 'men', 'met', 'might',
↪'nation', 'never', 'new', 'nobly', 'nor', 'not', 'note',
↪'november', 'now', 'of', 'on', 'or', 'our', 'people', 'perish
↪', 'place', 'poor', 'portion', 'power', 'proper',
↪'proposition', 'rather', 'remaining', 'remember', 'resolve',
↪'resting', 'say', 'score', 'sense', 'seven', 'shall', 'should
↪', 'so', 'struggled', 'take', 'task', 'testing', 'that', 'the
↪', 'their', 'these', 'they', 'this', 'those', 'thus', 'to',
↪'under', 'unfinished', 'us', 'vain', 'war', 'we', 'what',
↪'whether', 'which', 'who', 'will', 'work', 'world', 'years']
```

(continues on next page)

(continued from previous page)

8.4 Birthday

- Write a program called `birthday_021.py` that reads lines of text from `stdin` where each line consists of a person's date of birth. A date of birth is specified by three integers: a day, a month and a year. The program should determine on which day of the week each person was born and print the corresponding line from the poem:

  Monday's child is fair of face
  Tuesday's child is full of grace
  Wednesday's child is full of woe
  Thursday's child has far to go
  Friday's child is loving and giving
  Saturday's child works hard for a living
  Sunday's child is fair and wise and good in every way

  For example:

```
$ cat birthday_stdin_00_021.txt
1 3 1990
12 10 1992
9 5 1995
```

```
$ python3 birthday_021.py < birthday_stdin_00_021.txt
You were born on a Thursday and Thursday's child has far to go.
You were born on a Monday and Monday's child is fair of face.
You were born on a Tuesday and Tuesday's child is full of grace.
```

- Hint: Import the `calendar` module. Use `pydoc` to look up the `calendar.weekday()` function. It will do the hard work for you.

8.5 ABC

- Write a program called `abc_021.py` that reads two lines of text from `stdin`.
- The first line consists of three numbers: A, B, C. The numbers can be in any order but we know that A < B < C.
- The second line specifies the order that your program should output the numbers. For example:

```
$ cat abc_stdin_00_021.txt
6 4 2
CAB
```

```
$ python3 abc_021.py < abc_stdin_00_021.txt
6 2 4
```

- Here is another example:

```
$ cat abc_stdin_01_021.txt
1 5 3
ABC
```

```
$ python3 abc_021.py < abc_stdin_01_021.txt
1 3 5
```

CHAPTER

NINE

LECTURE 2.3 : TEXT FILES

9.1 Introduction

- Our Python programs must be able to save their data to the hard disk. This is *persistent* storage i.e. data saved to the hard disk survives a reboot (unlike data stored in RAM).
- Our Python programs also need to be able to retrieve data from files on the hard disk.
- Our programs will, for now, save their data in and retrieve their data from *text files*. (While text files are human-readable, *binary files* are not. We may cover them later.)
- File processing entails the following steps:
    1. **Open the file:** This step initialises a *file object* that acts as a link between the program and the file on the disk. All subsequent file operations are invoked on the file object. (A file object is sometimes referred to as a *file descriptor* or *stream*.)
    2. **Read and/or write the file:** This is where the work is done. Through the file object the on-disk contents of the file will be read and/or written.
    3. **Close the file:** This step finalises the file on the disk and unlinks the file object from the program.

9.2 Reading from `stdin` versus reading from a file

- Up until now we have been reading from `stdin`.
- You can think of `stdin` as a file **that is automatically opened for you** i.e. you do not have to open it in order to read from it.
- When you see a program invoked as follows the program is reading from `stdin` and you will not have to open any files.

```
$ python3 program.py < input.txt
```

- However, when you see a program invoked as shown below the program will have to open the file whose name is supplied in `argv[1]`.

```
$ python3 program.py input.txt
```

9.3 Our sample text file

- The file whose contents we will process is called `results.txt`.

```
$ cat results.txt
Mary Connolly 76
John Paul Jones 44
Fred Higgins 30
Laura Timmons 57
Fernandinho 22
```

9.4 Opening and closing a file

- Below we open a file for reading.
- Other modes in which a file can be opened include `w` for writing (warning: if the file already exists when opened for writing it will be truncated i.e. its contents deleted) and `a` for appending (additions to the file will follow any existing contents).
- When we specify a file name in the call to `open()` Python will look in the same directory as the program to find the file.
- If we wish to open a file that is not in the same directory as our program we need to supply a path to the file e.g. `f = open(r'/tmp/results.txt', 'r')`. (The `r` indicates this is a *raw string* and prevents characters such as / from taking on any special meaning the Python interpreter might ordinarily assign them.)

```
#!/usr/bin/env python3

import sys

def main():

    # Open a file for reading only. If the open succeeds (why␣
↪might it fail?)
```

(continues on next page)

(continued from previous page)

```
    # it returns a file object (that we assign to f).
    f = open(sys.argv[1], 'r')

    # Read in the entire file contents. Reading in the entire␣
↪contents might
    # not be a good idea. Why not?
    contents = f.read()

    # Display the contents
    print(contents)

    # Close the file
    f.close()

if __name__ == '__main__':
    main()
```

```
$ python3 file_v01.py results.txt
Mary Connolly 76
John Paul Jones 44
Fred Higgins 30
Laura Timmons 57
Fernandinho 22
```

9.5 Reading a file

- There are several methods available to a Python programmer for accessing the contents of a file. The most basic is `read()` which we saw above.
- A variant on `read()` is `readlines()`. While `read()` causes the entire contents of the file to be read, `readlines()` returns a list of strings, with each element of the list being a line from the file:
- A potential drawback to `read()` and `readlines()` is that they read in *the entire file contents* and store them in memory. If the file is large this might not be the most efficient use of resources.

```
#!/usr/bin/env python3

import sys

def main():
```

(continues on next page)

(continued from previous page)

```
    f = open(sys.argv[1], 'r')

    contents = f.readlines()

    print(contents)

    f.close()

if __name__ == '__main__':
    main()
```

```
$ python3 file_v02.py results.txt
['Mary Connolly 76\n', 'John Paul Jones 44\n', 'Fred Higgins 30\n',
↪ 'Laura Timmons 57\n', 'Fernandinho 22\n']
```

- A sometimes better alternative it to process the file line-by-line. We can read in the contents of a file one line at a time with `readline()` (when we reach the end of the file `readline()` sets `line` to the empty string).

```
#!/usr/bin/env python3

import sys

def main():

    f = open(sys.argv[1], 'r')

    line = f.readline()

    # Repeat until there is nothing left to read
    while line:
        print(line.strip())
        line = f.readline()

    f.close()

if __name__ == '__main__':
    main()
```

```
$ python3 file_v03.py results.txt
Mary Connolly 76
John Paul Jones 44
Fred Higgins 30
Laura Timmons 57
Fernandinho 22
```

- Often the most convenient way, however, to read a file line-by-line is to use an *iterator*. This approach is similar to using `readline()` but requires less code as an explicit check for the end of the file is not required (the iterator handles that). This is the least error-prone approach (and therefore the one you should prefer whenever appropriate):

```
#!/usr/bin/env python3

import sys

def main():

    f = open(sys.argv[1], 'r')

    for line in f:
        print(line.strip())

    f.close()

if __name__ == '__main__':
    main()
```

```
$ python3 file_v04.py results.txt
Mary Connolly 76
John Paul Jones 44
Fred Higgins 30
Laura Timmons 57
Fernandinho 22
```

9.6 File processing

- Each line of `results.txt` consists of a student name and mark. Let's write a program that reads each line from `results.txt` and prints out whether the student in question has passed (or not).
- We want to read in each line, extract the mark and student name, and print `passed` if the mark is 40+ and `failed` otherwise.
- The only difficulty is in extracting the name and exam mark from the line. Although a student's name may consist of a variable number of tokens we can take advantage of the fact that there is a single mark at the end of each line.

```
#!/usr/bin/env python3

import sys
```

(continues on next page)

(continued from previous page)

```python

def main():

    f = open(sys.argv[1], 'r')

    for line in f:
        tokens = line.strip().split()
        mark = int(tokens[-1])
        name = ' '.join(tokens[:-1])

        if mark >= 40:
            result = 'passed'
        else:
            result = 'failed'

        print(f'{name} {result} with a mark of {mark}')

    f.close()

if __name__ == '__main__':
    main()
```

```
$ python3 file_v05.py results.txt
Mary Connolly passed with a mark of 76
John Paul Jones passed with a mark of 44
Fred Higgins failed with a mark of 30
Laura Timmons passed with a mark of 57
Fernandinho failed with a mark of 22
```

CHAPTER

TEN

LECTURE 2.4 : EXCEPTION HANDLING

10.1 Introduction

- Our programs will encounter errors. When something goes wrong we do not want our programs to simply fall over. We want them to be robust to all circumstances that may arise at runtime. How can our programs cope with runtime errors?
- When something goes wrong at runtime the Python interpreter will *raise an exception*. To be robust to runtime errors our code must accept that they will arise from time to time and *handle* resultant exceptions when they are raised. Here are examples of some runtime errors:

```
# Try to convert 'cat' to an integer
int('cat')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_6590/2913609348.py in <module>
      1 # Try to convert 'cat' to an integer
----> 2 int('cat')

ValueError: invalid literal for int() with base 10: 'cat'
```

```
# Try to divide by zero
3/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
/tmp/ipykernel_6590/2984019755.py in <module>
      1 # Try to divide by zero
----> 2 3/0

ZeroDivisionError: division by zero
```

```
# Try to open a file that does not exist
open('no-such-file.txt', 'r')
```

```
---------------------------------------------------------------------------
↪--------
FileNotFoundError                         Traceback (most recent␣
↪call last)
/tmp/ipykernel_6590/3427071812.py in <module>
      1 # Try to open a file that does not exist
----> 2 open('no-such-file.txt', 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'no-such-
↪file.txt'
```

10.2 File processing with no exception handling

- Let's look at what happens when a program that processes student results is passed the name of a file that does not exist.

```
#!/usr/bin/env python3

import sys

def main():

    f = open(sys.argv[1], 'r')

    for line in f:
        tokens = line.strip().split()
        mark = int(tokens[-1])
        name = ' '.join(tokens[:-1])

        if mark >= 40:
            result = 'passed'
        else:
            result = 'failed'
```

(continues on next page)

(continued from previous page)

```
        print(f'{name} {result} with a mark of {mark}')

    f.close()

if __name__ == '__main__':
    main()
```

```
$ python3 procfile_v01.py no-such-file.txt
Traceback (most recent call last):
  File "/home/dobrien/darragh/vinson/ca117/sphinx/week02/lecture04/
↪procfile_v01.py", line 24, in <module>
    main()
  File "/home/dobrien/darragh/vinson/ca117/sphinx/week02/lecture04/
↪procfile_v01.py", line 7, in main
    f = open(sys.argv[1], 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'no-such-
↪file.txt'
```

10.3 Handling the FileNotFoundError exception

- Rather than have a program abruptly exit on encountering such an error (the default behaviour) Python allows programmers to handle such scenarios gracefully with a `try-except` construct.
- In the `try` block of code we place the instructions that may fail due to a runtime error.
- In the `except` block of code we place the instructions to be carried out in the event of the `try` block failing due to a runtime error.
- If no error arises in the `try` block then execution continues at the instruction following the `try-except` (the contents of the `except` block are ignored).
- If an error occurs within the `try` block, execution stops at that point and the rest of the `try` block is ignored. An exception corresponding to the specific error that has arisen is *raised*. Python then searches for an `except` block that can handle the exception.
- If a suitable `except` block is found it is executed and execution continues from the point following the `try-except`.
- We have updated our program below to handle the file not found error gracefully.

```
#!/usr/bin/env python3

import sys
```

(continues on next page)

(continued from previous page)

```
def main():

    try:
        f = open(sys.argv[1], 'r')

        for line in f:
            tokens = line.strip().split()
            mark = int(tokens[-1])
            name = ' '.join(tokens[:-1])

            if mark >= 40:
                result = 'passed'
            else:
                result = 'failed'

            print(f'{name} {result} with a mark of {mark}')

        f.close()

    except FileNotFoundError:
        print(f'The file {sys.argv[1]} does not exist.')

if __name__ == '__main__':
    main()
```

```
$ python3 procfile_v02.py no-such-file.txt
The file no-such-file.txt does not exist.
```

10.4 What about other exceptions?

- Our code is not yet robust to all runtime errors however. For example, let's see what happens if the file we are processing is incorrectly formatted e.g. due to a typographic error it does not contain an integer mark. If no suitable `except` block is found then the default behaviour applies and program execution is halted.

```
$ cat errors.txt
Mary Connolly 76
John Paul Jones 44
Fred Higgins e0
Laura Timmons 57
Fernandinho 22
```

```
$ python3 procfile_v02.py errors.txt
Mary Connolly passed with a mark of 76
John Paul Jones passed with a mark of 44
Traceback (most recent call last):
  File "/home/dobrien/darragh/vinson/ca117/sphinx/week02/lecture04/
↪procfile_v02.py", line 28, in <module>
    main()
  File "/home/dobrien/darragh/vinson/ca117/sphinx/week02/lecture04/
↪procfile_v02.py", line 12, in main
    mark = int(tokens[-1])
ValueError: invalid literal for int() with base 10: 'e0'
```

- Hmm. We have a couple of problems here. Firstly our program is crashing on encountering a illegal mark. Secondly, because the exception causes our program to exit immediately it does so *without closing the file*. That's bad practice.
- Let's handle the second problem first. Can we fix it so that the file is *always* closed i.e. it is closed when the program runs correctly *and* it is closed in the event of an (unhandled) exception?
- Yes. The use of the `with` statement means the file is always closed cleanly irrespective of whether or not an exception is raised. Let's modify our program to use such a `with` statement:

```python
#!/usr/bin/env python3

import sys

def main():

    try:
        with open(sys.argv[1], 'r') as f:

            for line in f:
                tokens = line.strip().split()
                mark = int(tokens[-1])
                name = ' '.join(tokens[:-1])

                if mark >= 40:
                    result = 'passed'
                else:
                    result = 'failed'

                print(f'{name} {result} with a mark of {mark}')

    except FileNotFoundError:
        print(f'The file {sys.argv[1]} does not exist.')
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

10.5 Handling illegal marks

- To gracefully handle the `ValueError` exception caused by the presence of an illegal mark in our input file we need a new `except` block. Where should we place it?
- Well that depends on the kind of behaviour we want. If an error occurs do we want to continue processing the remainder of the file following the error or do we want to give up immediately and ignore the rest of the file?
- If we want to give up immediately and discontinue file processing then we need to place our `except` block *outside* the for loop (so we exit the loop when the exception is handled) and we would do something like this:

```
#!/usr/bin/env python3

import sys

def main():

    try:
        with open(sys.argv[1], 'r') as f:

            for line in f:
                tokens = line.strip().split()
                mark = int(tokens[-1])
                name = ' '.join(tokens[:-1])

                if mark >= 40:
                    result = 'passed'
                else:
                    result = 'failed'

                print(f'{name} {result} with a mark of {mark}')

    except ValueError:
        print(f'Illegal mark encountered: {tokens[-1]}')

    except FileNotFoundError:
        print(f'The file {sys.argv[1]} does not exist.')
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

```
$ python3 procfile_v04.py errors.txt
Mary Connolly passed with a mark of 76
John Paul Jones passed with a mark of 44
Illegal mark encountered: e0
```

- If we want to report the illegal mark but process the remainder of the file then we need to place a new `try-except` construct *inside* the for loop.
- Staying inside the for loop on encountering an error means we will go on and process the remainder of the file:

```
#!/usr/bin/env python3

import sys

def main():

    try:
        with open(sys.argv[1], 'r') as f:

            for line in f:

                try:
                    tokens = line.strip().split()
                    mark = int(tokens[-1])
                    name = ' '.join(tokens[:-1])

                    if mark >= 40:
                        result = 'passed'
                    else:
                        result = 'failed'

                    print(f'{name} {result} with a mark of {mark}')

                except ValueError:
                    print(f'Illegal mark encountered: {tokens[-1]}
↪')

    except FileNotFoundError:
        print(f'The file {sys.argv[1]} does not exist.')
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

```
$ python3 procfile_v05.py errors.txt
Mary Connolly passed with a mark of 76
John Paul Jones passed with a mark of 44
Illegal mark encountered: e0
Laura Timmons passed with a mark of 57
Fernandinho failed with a mark of 22
```

10.6 The `else` block

- If execution leaves the `try` block *normally* i.e. **not** as a result of an exception and not as a result of `break`, `continue` or `return` then the `else` block (if present) is executed. The `else` block must be placed after all `except` blocks.

```
#!/usr/bin/env python3

import sys

def main():
    try:
        with open(sys.argv[1], 'r') as f:
            for line in f:
                tokens = line.strip().split()
                mark = int(tokens[-1])
                name = ' '.join(tokens[:-1])

                if mark >= 40:
                    result = 'passed'
                else:
                    result = 'failed'

                print(f'{name} {result} with a mark of {mark}')

    except ValueError:
        print(f'Illegal mark encountered: {tokens[-1]}')

    except FileNotFoundError:
        print(f'The file {sys.argv[1]} does not exist.')

    else:
        print('Reached end of file')
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

```
$ python3 procfile_v06.py results.txt
Mary Connolly passed with a mark of 76
John Paul Jones passed with a mark of 44
Fred Higgins failed with a mark of 30
Laura Timmons passed with a mark of 57
Fernandinho failed with a mark of 22
Reached end of file
```

```
$ python3 procfile_v06.py errors.txt
Mary Connolly passed with a mark of 76
John Paul Jones passed with a mark of 44
Illegal mark encountered: e0
```

10.7 The `finally` block

- A `finally` block is often used in conjunction with `try` and `except` blocks. In the `finally` block we place code that we *always* want executed, irrespective of whether an exception occurs or not. Below we augment our program with a `finally` block that prints a summary of all successfully processed lines before exiting:

```
#!/usr/bin/env python3

import sys

def main():
    lines = 0
    try:
        with open(sys.argv[1], 'r') as f:
            for line in f:

                try:
                    tokens = line.strip().split()
                    mark = int(tokens[-1])
                    name = ' '.join(tokens[:-1])

                    if mark >= 40:
                        result = 'passed'
                    else:
```

(continues on next page)

(continued from previous page)

```
                        result = 'failed'

                    print(f'{name} {result} with a mark of {mark}')
                    lines += 1

                except ValueError:
                    print(f'Illegal mark encountered: {tokens[-1]}
↪')

    except FileNotFoundError:
        print(f'The file {sys.argv[1]} does not exist.')

    else:
        print('Reached end of file')

    finally:
        print(f'Lines processed: {lines}')

if __name__ == '__main__':
    main()
```

```
$ python3 procfile_v07.py errors.txt
Mary Connolly passed with a mark of 76
John Paul Jones passed with a mark of 44
Illegal mark encountered: e0
Laura Timmons passed with a mark of 57
Fernandinho failed with a mark of 22
Reached end of file
Lines processed: 4
```

10.8 Writing a file

- Let's complete our program such that it writes its output to a file rather than to the screen. Note how we have enhanced the `with` statement to deal with two files:

```
#!/usr/bin/env python3

import sys

def main():
    lines = 0
    try:
        with open(sys.argv[1], 'r') as fin, open(sys.argv[2], 'w')␣
↪as fout:
```

(continues on next page)

(continued from previous page)

```
            for line in fin:

                try:
                    tokens = line.strip().split()
                    mark = int(tokens[-1])
                    name = ' '.join(tokens[:-1])

                    if mark >= 40:
                        result = 'passed'
                    else:
                        result = 'failed'

                    fout.write(f'{name} {result} with a mark of
↪{mark}\n')
                    lines += 1

                except ValueError:
                    print(f'Illegal mark encountered: {tokens[-1]}
↪')

    except FileNotFoundError:
        print(f'The file {sys.argv[1]} does not exist.')

    else:
        print('Reached end of file')

    finally:
        print(f'Lines processed: {lines}')

if __name__ == '__main__':
    main()
```

```
$ python3 procfile_v08.py errors.txt processed.txt
Illegal mark encountered: e0
Reached end of file
Lines processed: 4
```

```
$ cat processed.txt
Mary Connolly passed with a mark of 76
John Paul Jones passed with a mark of 44
Laura Timmons passed with a mark of 57
Fernandinho failed with a mark of 22
```

CHAPTER

ELEVEN

LAB 2.2 (DEADLINE FRIDAY 28 JANUARY 23:59)

- Upload your code to Einstein to have it verified.

11.1 Best student version 1

- Write a program called `best_v1_022.py` that takes a filename as a single command line argument. The file contains a list of student marks where each line consists of a mark followed by a student name.
- The program should print out the name of the student who achieved the highest mark as well as the mark itself. Where there is a joint top mark the program should print the name associated with the first one encountered, for example:

```
$ cat best_v1_input_00_022.txt
64 Mary Ryan
89 Michael Murphy
22 Pepe
78 Jenny Smith
57 Patrick James McMahon
89 John Kelly
22 Pepe
74 John C. Reilly
```

```
$ python3 best_v1_022.py best_v1_input_00_022.txt
Best student: Michael Murphy
Best mark: 89
```

- If the filename specified on the command line cannot be opened the program must make appropriate use of exception handling to display an error message and exit gracefully. For example:

```
$ python3 best_v1_022.py best_v1_input_00_022.txtt
The file best_v1_input_00_022.txtt could not be opened.
```

11.2 Best student version 2

- Extend the exception handling in the above program such that the program exits gracefully if any of the marks in the input file are not integers. Call the new program `best_v2_022.py`. For example:

```
$ cat best_v2_input_00_022.txt
64 Mary Ryan
89 Michael Murphy
22 Pepe
78 Jenny Smith
5a Patrick James McMahon
89 John Kelly
22 Pepe
74 John C. Reilly
90 Penelope Pitstop
```

```
$ python3 best_v2_022.py best_v2_input_00_022.txt
Invalid mark 5a encountered. Exiting.
```

11.3 Best student version 3

- Extend the exception handling in the above program such that rather than exiting on encountering a non-integer mark the program simply ignores it and continues to process all remaining marks. Call the new program `best_v3_022.py`. For example:

```
$ cat best_v3_input_00_022.txt
64 Mary Ryan
89 Michael Murphy
22 Pepe
78 Jenny Smith
5a Patrick James McMahon
89 John Kelly
22 Pepe
74 John C. Reilly
90 Penelope Pitstop
```

```
$ python3 best_v3_022.py best_v3_input_00_022.txt
Invalid mark 5a encountered. Skipping.
Best student: Penelope Pitstop
Best mark: 90
```

11.4 Best students

- The above programs print out a single name even when multiple students share the highest mark. Write a program called `bests_022.py` that prints out the names of all students who achieved the highest mark.
- Student names should be comma-separated and must be printed in the order in which they occur in the input file. For example:

```
$ cat bests_input_00_022.txt
64 Mary Ryan
89 Michael Murphy
22 Pepe
78 Jenny Smith
57 Patrick James McMahon
89 John Kelly
22 Pepe
74 John C. Reilly
```

```
$ python3 bests_022.py bests_input_00_022.txt
Best student(s): Michael Murphy, John Kelly
Best mark: 89
```

11.5 Two files

- Write a program called `twofiles_022.py` that takes two filenames as command line arguments. Each file contains a list of integers (one per line) and each file contains the same number of integers.
- The program should output a list of integers, one per line, where integers are taken alternately from each file. For example:

```
$ cat twofiles_input_a_00_022.txt
1
3
5
7
```

```
$ cat twofiles_input_b_00_022.txt
2
4
6
8
```

```
$ python3 twofiles_022.py twofiles_input_a_00_022.txt twofiles_
↪input_b_00_022.txt
1
2
3
4
5
6
7
8
```

11.6 Exceptions puzzles

- Without running them, write out the output of each of the following programs (run each afterwards to check your answer):

```
#!/usr/bin/env python3

def callme(x):
    y = None
    try:
        y = 1//x
    except ZeroDivisionError:
        print('In exception handler')
    else:
        print('Reached end of try')
    finally:
        print('Exiting the function')
    return y

def main():
    print(callme(1))
    print(callme(0))

if __name__ == '__main__':
    main()
```

```
#!/usr/bin/env python3

def callme(x):
    try:
        y = 1//x
    except ZeroDivisionError:
```

(continues on next page)

(continued from previous page)

```
        print('In exception handler')
    else:
        print('Reached end of try')
    finally:
        print('Exiting the function')
    return y

def main():
    print(callme(1))
    print(callme(0))

if __name__ == '__main__':
    main()
```

```
#!/usr/bin/env python3

def callme(x):
    try:
        y = 1//x
        return y
    except ZeroDivisionError:
        print('In exception handler')
    else:
        print('Reached end of try')
    finally:
        print('Exiting the function')

def main():
    print(callme(1))
    print(callme(0))

if __name__ == '__main__':
    main()
```

- Note there is no marker on Einstein for these puzzles so you do not need to upload anything.

CHAPTER

TWELVE

LECTURE 3.1 : LIST COMPREHENSIONS

12.1 Introduction

- Consider the following programming task: Write a Python function that takes a list of integers as an argument and returns a new list whose members are all the odd integers in input list. To solve such a programming task we might write code such as the following:

```
def extract_odds(numbers):
    odds = []
    for n in numbers:
        if n % 2:
            odds.append(n)
    return odds

print(extract_odds([3, 4, 6, 5, 1, 8]))
```

```
[3, 5, 1]
```

- It works. However it turns out that this pattern of building one list by processing (or transforming) the elements of another is so common that Python provides a short-cut for doing it. The short-cut is called a *list comprehension*.

12.2 List comprehensions

- A list comprehension is a short-cut to building one list from another. Its general form is: `[expression for-clause condition]`.
- The surrounding square brackets indicate we are building a list (i.e. they tell us this is a list comprehension). Let's refer to this list as `new_list`.
- The result of evaluating `expression` for each iteration of the `for-clause` is added to `new_list`. (It is typically an `expression` over elements of the list we are processing.)

- The `for-clause` visits each element of the list we are processing. Let's refer to this list as `old_list`.
- The `condition` allows us to select for inclusion in `new_list` expressions over only those elements of `old_list` that meet certain criteria. The condition is applied to each element of `old_list` and only if the condition evaluates to `True` is that element added to `new_list`.
- Rewriting our `extract_odds` function to use a list comprehension we get the code shown below. Note how compact it is compared to the original version.

```
def extract_odds(numbers):
    odds = [n for n in numbers if n % 2]
    return odds

print(extract_odds([3, 4, 6, 5, 1, 8]))
```

```
[3, 5, 1]
```

- The list comprehension `[n for n in numbers if n % 2]` can be read as:

1. For each `n` in the list `numbers` (for-clause: `for n in numbers`)
2. If `n` is odd (condition: `if n % 2`)
3. Add `n` to the list being built (expression: `n`)

- Note how in this list comprehension `n` is used in all three of the `expression`, the `for-clause`, and the `condition`.

12.3 Another example

- Write a Python function that accepts a string as an argument and returns a new string whose characters are all those of the original string that are non-vowels. To solve this problem we could write code such as the following:

```
def extract_nonvowels(s):
    nonvowels = []
    for c in s:
        if c.lower() not in 'aeiou':
```

(continues on next page)

(continued from previous page)

```
        nonvowels.append(c)
    return ''.join(nonvowels)

print(extract_nonvowels('Are vowels required to understand␣
↪sentences?'))
```

```
r vwls rqrd t ndrstnd sntncs?
```

- Rewriting the code to use a list comprehension we get:

```
def extract_nonvowels(s):
    nonvowels = [c for c in s if c.lower() not in 'aeiou']
    return ''.join(nonvowels)

print(extract_nonvowels('Are vowels required to understand␣
↪sentences?'))
```

```
r vwls rqrd t ndrstnd sntncs?
```

- The list comprehension `[c for c in s if c.lower() not in 'aeiou']` can be read as:

1. For each `c` in the string `s` (for-clause: `for c in s`)
2. If `c` is not a vowel (condition: `if c.lower() not in 'aeiou'`)
3. Add `c` to the list being built (expression: `c`)

12.4 Another example

- Write a Python function that takes a list of integers as an argument and returns a new list whose members are the square of all the even integers in the input list. To solve this problem we could write code such as the following:

```
def even_squares(numbers):
    squares = []
    for n in numbers:
        if not n % 2:
            squares.append(n**2)
    return squares

print(even_squares([3, 4, 6, 5, 1, 8]))
```

```
[16, 36, 64]
```

- Rewriting the code to use a list comprehension we get:

```
def even_squares(numbers):
    return [n**2 for n in numbers if not n % 2]

print(even_squares([3, 4, 6, 5, 1, 8]))
```

```
[16, 36, 64]
```

12.5 A final example

- Write a Python function that takes a list of integers as an argument and returns a new list whose members are the square of all the even integers in the input list and the cube of all the odd integers in the input list. To solve this problem we could write code such as the following:

```
def even_squares_odd_cubes(numbers):
    squares_n_cubes = []
    for n in numbers:
        if not n % 2:
            squares_n_cubes.append(n**2)
```

(continues on next page)

(continued from previous page)

```
        else:
            squares_n_cubes.append(n**3)
    return squares_n_cubes

print(even_squares_odd_cubes([3, 4, 6, 5, 1, 8]))
```

```
[27, 16, 36, 125, 1, 64]
```

- Rewriting the code to use a list comprehension we get:

```
def even_squares_odd_cubes(numbers):
    return [n**3 if n % 2 else n**2 for n in numbers]

print(even_squares_odd_cubes([3, 4, 6, 5, 1, 8]))
```

```
[27, 16, 36, 125, 1, 64]
```

- If you read Python code on-line you will find it often makes use of comprehensions so it is important you understand how they work. Using comprehensions where appropriate is considered the *Pythonic* way of problem-solving.
- When you move on to study other programming languages however you will find they do not support such comprehension constructs so you will revert to our original pattern to building one list from another.

12.6 More comprehensions

- Comprehensions do not apply to lists alone.
- They can be used as short-cuts to building a new *collection* from another *collection*. Thus it makes sense to talk not only about list comprehensions but also *set* and *dictionary comprehensions*. We may look at these later in the course.

CHAPTER

THIRTEEN

LAB 3.1 (DEADLINE FRIDAY 4 FEBRUARY 23:59)

- Upload your code to Einstein to have it verified.

13.1 List comprehensions

- Write a program called `numcomps_031.py` that uses a `for` loop and the `range()` function to generate a list containing the numbers 1, 2, 3, . . . , N (where N is an integer supplied from `stdin`).
- Use *list comprehensions* to have your program extract from the above list the following:
  1. All multiples of 3.
  2. The squares of all multiples of 3.
  3. The double of all multiples of 4.
  4. All multiples of either 3 or 4.
  5. All multiples of both 3 and 4.
- Here is an example of how the program should behave:

```
$ cat numcomps_stdin_00_031.txt
8
12
```

```
$ python3 numcomps_031.py < numcomps_stdin_00_031.txt
Multiples of 3: [3, 6]
Multiples of 3 squared: [9, 36]
Multiples of 4 doubled: [8, 16]
Multiples of 3 or 4: [3, 4, 6, 8]
Multiples of 3 and 4: []
Multiples of 3: [3, 6, 9, 12]
Multiples of 3 squared: [9, 36, 81, 144]
```

(continues on next page)

(continued from previous page)

```
Multiples of 4 doubled: [8, 16, 24]
Multiples of 3 or 4: [3, 4, 6, 8, 9, 12]
Multiples of 3 and 4: [12]
```

13.2 Comprehensions with replacement

- Write a program called `repcomps_031.py` that uses a `for` loop and the `range()` function to generate a list containing the numbers 1, 2, 3, . . . , N (where N is an integer supplied from `stdin`).
- Use a *list comprehension* to have your program duplicate the above list but with all multiples of 3 replaced by ‘X’. For example:

```
$ cat repcomps_stdin_00_031.txt
8
12
```

```
$ python3 repcomps_031.py < repcomps_stdin_00_031.txt
Multiples of 3 replaced: [1, 2, 'X', 4, 5, 'X', 7, 8]
Multiples of 3 replaced: [1, 2, 'X', 4, 5, 'X', 7, 8, 'X', 10, 11,
↪'X']
```

13.3 Primes

- Write a program called `primecomps_031.py` that uses a `for` loop and the `range()` function to generate a list containing the numbers 1, 2, 3, . . . , N (where N is an integer supplied from `stdin`).
- Use a *list comprehension* to have your program extract all prime numbers from the above list. For example:

```
$ cat primecomps_stdin_00_031.txt
8
12
```

```
$ python3 primecomps_031.py < primecomps_stdin_00_031.txt
Primes: [2, 3, 5, 7]
Primes: [2, 3, 5, 7, 11]
```

13.4 More list comprehensions

- Write a program called `wordcomps_031.py` that reads words from `stdin` (one word per line) and stores them all in a list.
- Using *list comprehensions* and ignoring differences in case, have your program print the following lists:
  1. Words that contain exactly 17 letters.
  2. Words that contain 18+ letters.
  3. Words that contain four a's.
  4. Words that contain two or more q's.
  5. Words that contain the sequence 'cie'.
  6. Words that are anagrams of 'angle'.
- Your program should produce the following output when run against dictionary05.txt:

```
$ python3 wordcomps_031.py < dictionary05.txt
Words containing 17 letters: ['contradistinguish',
↪'counterproductive', 'counterrevolution', 'electrocardiogram',
↪'indistinguishable', 'paleoanthropology', 'psychotherapeutic',
↪'spectrophotometer']
Words containing 18+ letters: ['arteriolosclerosis',
↪'counterrevolutionary', 'diethylstilbestrol', 'electrocardiograph
↪', 'electroencephalogram', 'electroencephalograph',
↪'electroencephalography', 'immunoelectrophoresis',
↪'triphenylphosphine']
Words with 4 a's: ['Alabama', 'Alabamian', 'amalgamate', 'Anastasia
↪', 'Appalachia', 'Atalanta', 'Athabascan', 'baccalaureate',
↪'bacchanalian', 'Bhagavadgita', 'extravaganza', 'Macadamia',
↪'Madagascar', 'maharaja', 'Maharashtra', 'Mahayana', 'Nakayama',
↪'Panamanian', 'Paraguayan', 'paraphernalia', 'parliamentarian',
↪'Santayana', 'sarsaparilla', 'tarantara']
Words with 2+ q's: ['Albuquerque']
Words containing cie: ['ancient', 'coefficient', 'concierge',
↪'conscience', 'conscientious', 'deficient', 'efficient',
↪'financier', 'glacier', 'hacienda', 'inefficient', 'insufficient
↪', 'Muncie', 'omniscient', 'proficient', 'science', 'scientific',
↪ 'scientist', 'societal', 'Societe', 'society', 'specie',
↪'species', 'sufficient']
Anagrams of angle: ['angel', 'Galen', 'glean', 'Lange']
```

- Hint: When checking if a word should be included in the new list convert it to lower case for the purposes of the check but add the original word to the list (should it pass the test).

13.5 Q no u

- Write a program called `qnou_031.py` that reads words from `stdin` (one word per line) and stores them all in a list.
- Making appropriate use of a *list comprehension* and ignoring differences in case, have the program print a list of all words in the list that contain a *q* that is not immediately followed by a *u*. For example:

```
$ cat qnou_stdin_00_031.txt
question
Colloq
IQ
Iraq
Iraqi
q
Qatar
QED
q's
seq
inquest
```

```
$ python3 qnou_031.py < qnou_stdin_00_031.txt
Words with q but no u: ['Colloq', 'IQ', 'Iraq', 'Iraqi', 'q',
↪'Qatar', 'QED', "q's", 'seq']
```

- Note: Part of this exercise involves you coming up with test cases not present above but present in the hidden input/output on Einstein.

CHAPTER

FOURTEEN

LECTURE 3.2 : VARIABLES, REFERENCES, IMMUTABLE AND MUTABLE OBJECTS

14.1 Introduction

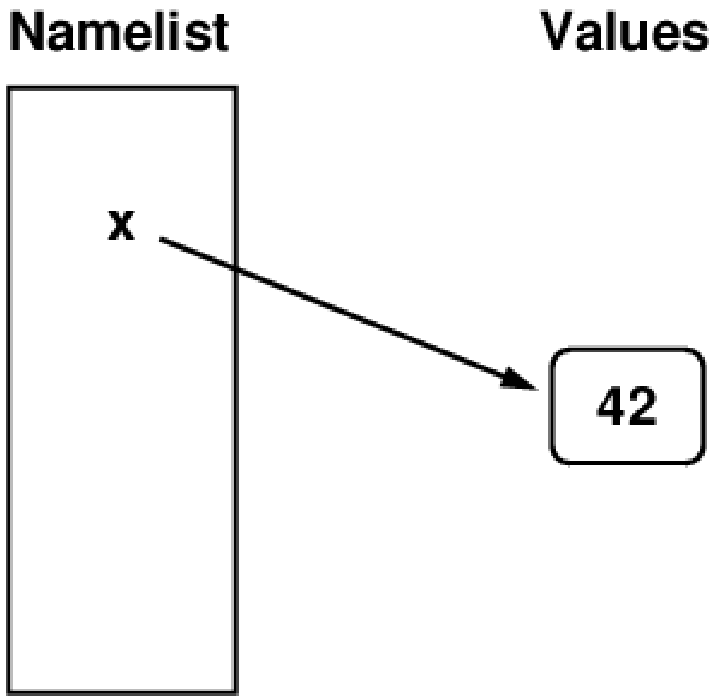
- A variable can contain a reference to an *immutable* object or a *mutable* object. We examine the different behaviours that arise when working with each.

14.2 Variables, references, objects

- What exactly happens when Python executes the following statement?

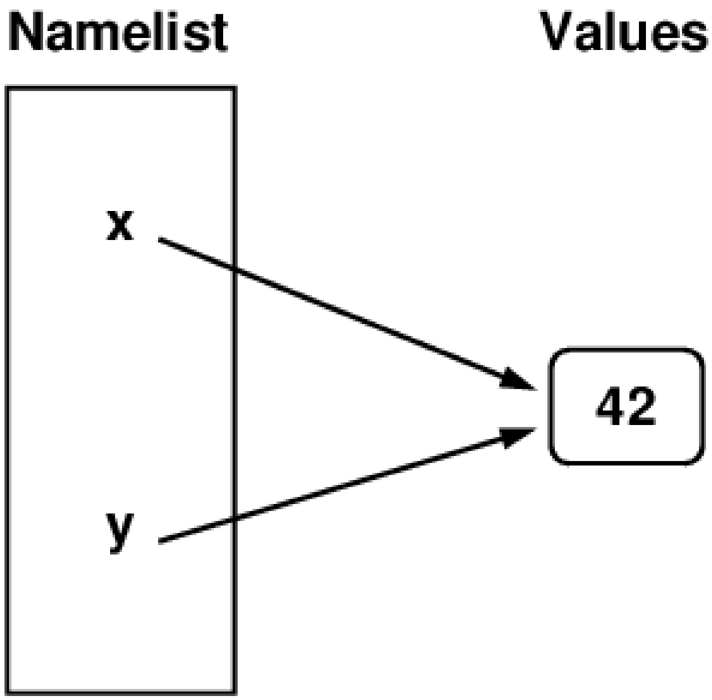
```
x = 42
```

- Some memory is allocated (you can think of this as a box) and `42` is placed inside. The variable `x` then comes into existence and is associated with the box containing `42`.
- Python maintains a *namespace* of mappings from variables to the objects they refer to. After the above statement is executed a mapping from `x` to the value `42` is added to the namespace. This is depicted below.



- When we subsequently refer to `x` in our program, Python dereferences `x` (i.e. it follows the arrow) and finds `42`.
- As we can see, the variable `x` *does not contain* the value `42`. Instead some memory is reserved to hold the value `42` and `x` *points to* that location in memory. We say that `x` contains a *reference* to the number `42` stored in memory.
- What happens the picture when we execute the following statements?

```
x = 42
y = x
```



- We see that having executed `y = x` both `x` and `y` reference the same object in memory, in this case the integer `42`. The variables `x` and `y` are known as *aliases* because they refer to the same object by different names.
- We can verify this is the case by using the `id()` function to print out the location in memory (i.e. memory address) that each of `x` and `y` reference:

```
x = 42
y = x
print(id(x))
print(id(y))
print(id(x) == id(y))
```

```
139648518950480
139648518950480
True
```

- What happens the picture when we modify `x` as shown below?

```
x = 42
y = x
x += 1
```

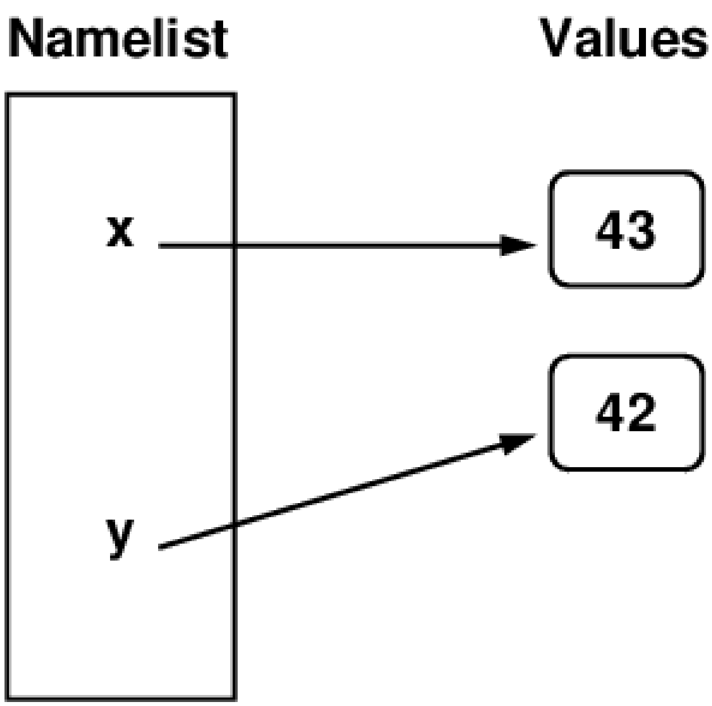
(continues on next page)

(continued from previous page)

```
print(x)
print(y)
```

```
43
42
```

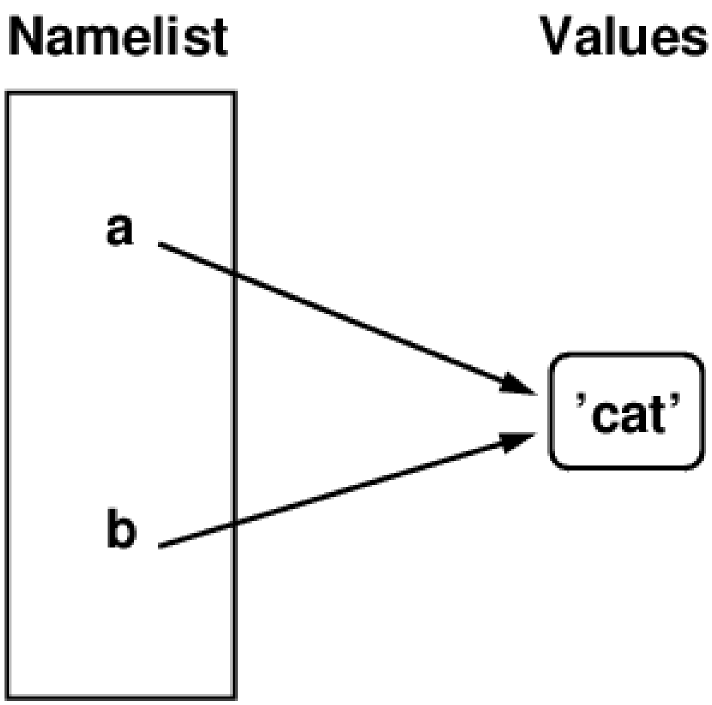
- Executing `x += 1` is equivalent to executing `x = x + 1`. Evaluating the right hand side gives a new integer `43` and we set `x` pointing to it in memory.
- Thus the effect of executing `x += 1` is to *overwrite* `x` with a *new* reference to a *new* integer object. This time the reference is to the integer `43`.
- Note that overwriting the reference in `x` with a new one has *no effect* on the reference in `y`. It still points to `42`.



14.3 Variables, references and immutable objects

- The `int` type is *immutable*. This means operations on an integer by necessity create a *new* integer. Thus a modification of the integer referenced by `x` above creates a *new* integer leaving the integer pointed to by `y` unchanged.
- Integers are not the only immutable type we have met. Strings are also immutable and behave similarly.

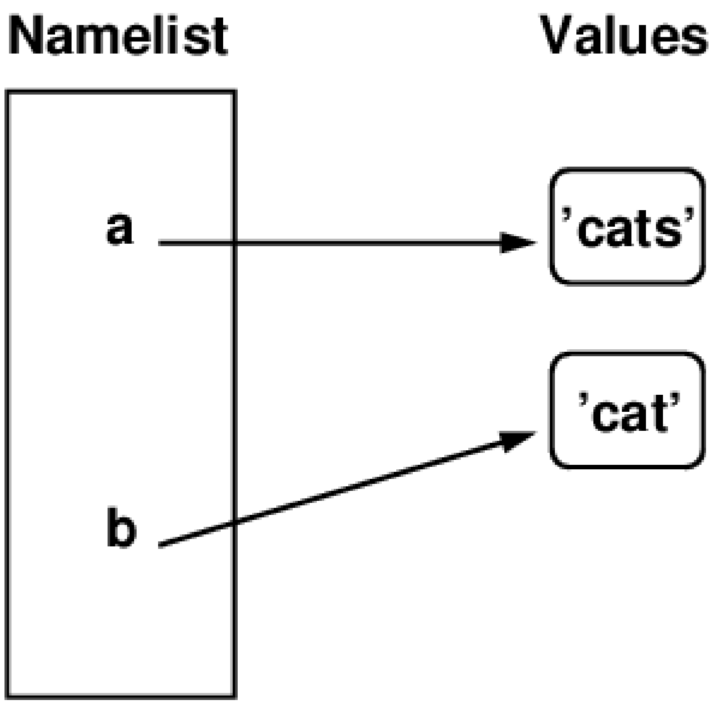
```
a = 'cat'
b = a
```



- If we modify the string referenced by `a` we get a new string. (Remember every adjustment to an *immutable* object by necessity creates a *new* object.)
- Below we overwrite `a` with a reference to this new string leaving `b` unchanged:

```
a = 'cat'
b = a
a += 's'
print(a)
print(b)
```

```
cats
cat
```

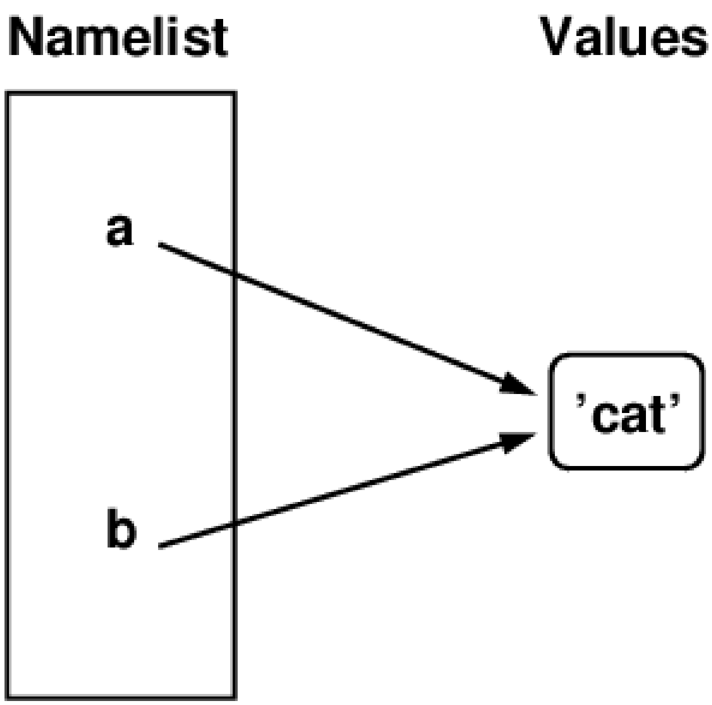


14.4 Equality and identity

- We can check whether the objects referenced by two variables are equal using the == operator.
- We can check whether two variables reference the *same* object using the `is` operator.
- If `a is b` then `a == b`.
- If `a == b` it is not necessarily true that `a is b`.

```
a = 'cat'
b = a
print(a)
print(b)
print(a == b)
print(a is b) # check whether id(a) == id(b)
```

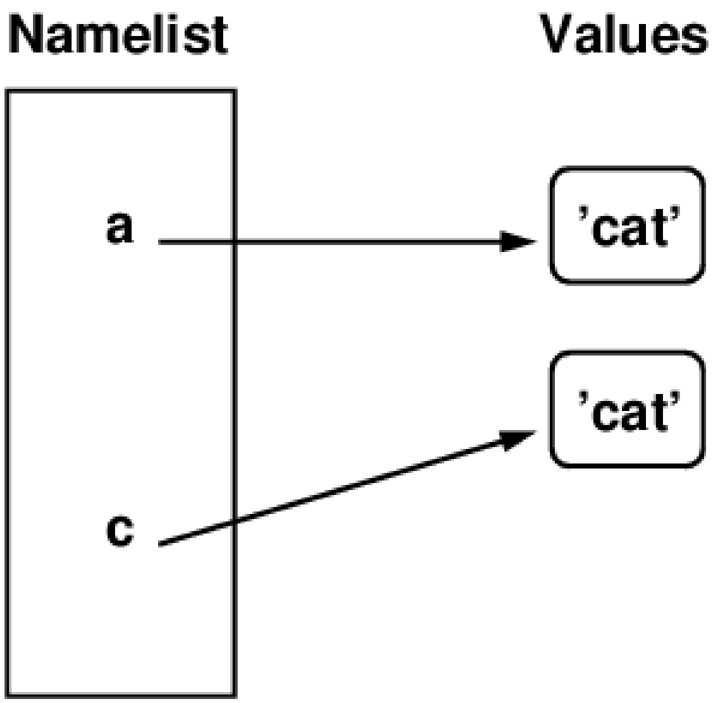
```
cat
cat
True
True
```




```
a = 'cat'
c = 'catastrophe'[:3]
print(a)
print(c)
print(a == c)
print(a is c) # check whether id(a) == id(c)
```

```
cat
cat
True
False
```

- Above we can see that both `a` and `c` point to `cat` but it is not the same `cat` in memory. Thus although they are equal i.e. `a == c` is `True` they do *not* reference the same object i.e. `a is c` is `False`.

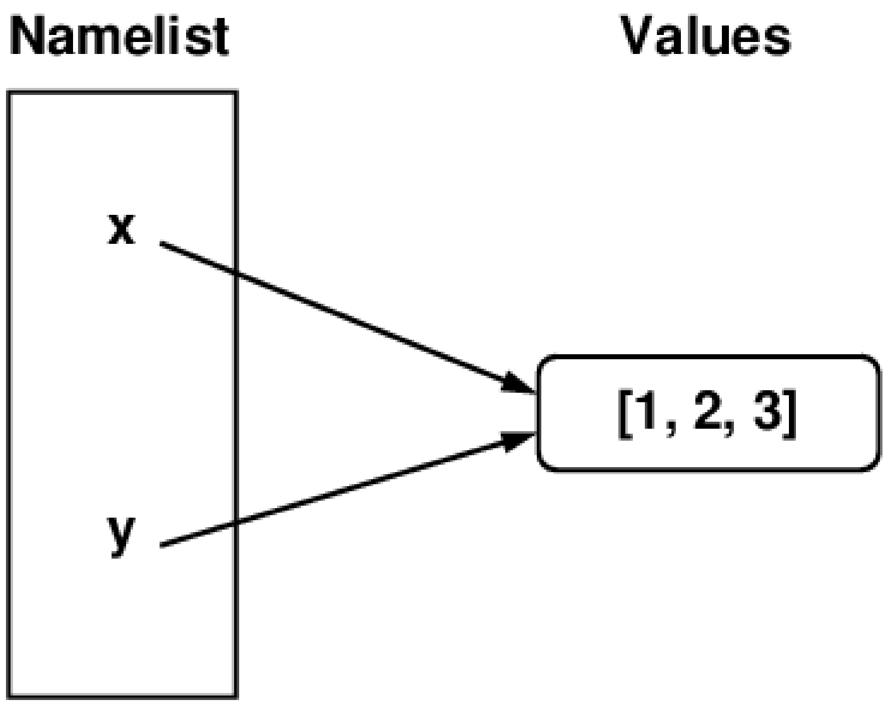


14.5 Variables, references and mutable objects

- Things get more complicated when multiple variables reference the same *mutable* object. We need to be careful as there are consequences to such sharing that may not be immediately obvious.

```
x = [1, 2, 3]
y = x
print(x == y)
print(x is y)
```

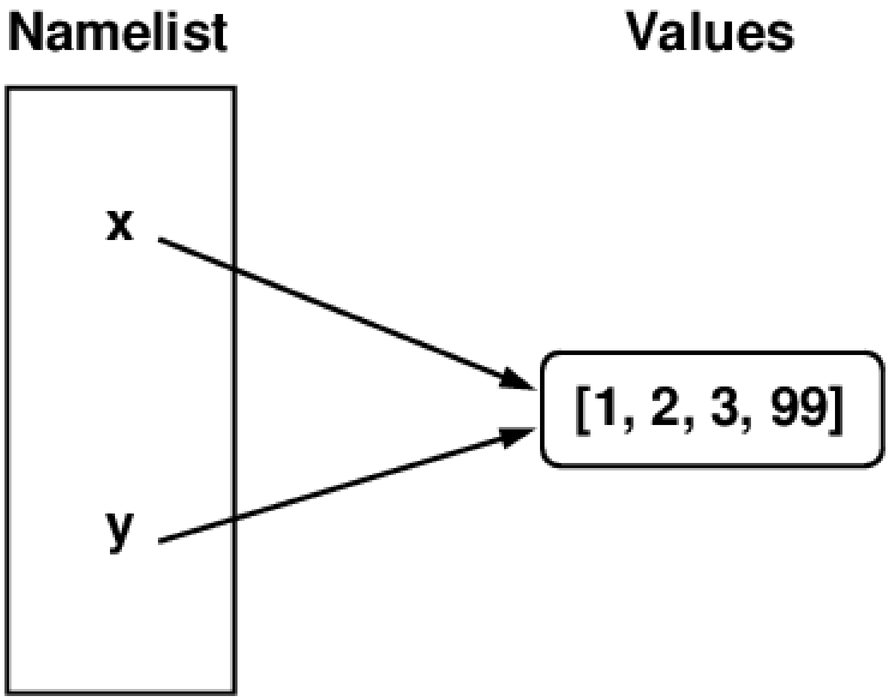
```
True
True
```



- Now consider what happens to `y` when we write *through* the reference in `x` in order to append an element to the underlying list:

```
x = [1, 2, 3]
y = x
x.append(99)
print(x)
print(y)
```

```
[1, 2, 3, 99]
[1, 2, 3, 99]
```



- The crucial point to note here is that because the object pointed to by `x` (and `y`) is *mutable*, modifying it does *not* create a new object and the original reference is *not* overwritten to point to a new object.
- We do not overwrite the reference in the variable `x` but instead we write *through* it to modify the mutable object it points to.

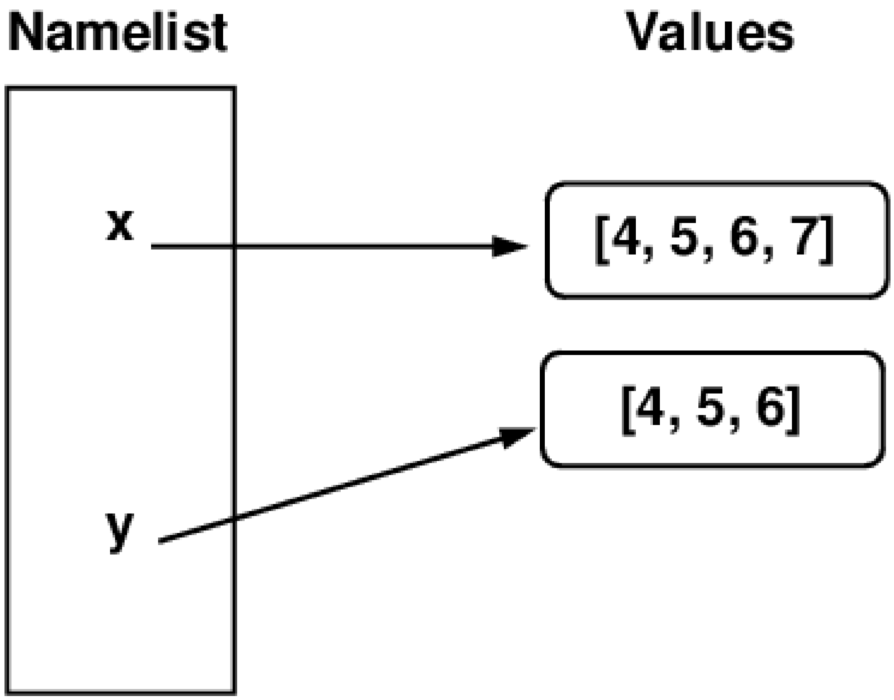
14.6 Gotcha

- There are however some tricky subtleties to this behaviour. One is illustrated below.

```
x = [4, 5, 6]
y = x
x = x + [7]
print(x)
print(y)
```

```
[4, 5, 6, 7]
[4, 5, 6]
```

- Huh? How come `y` was not modified in this example even though we made a change to `x`?
- Well, here we executed the code `x = x + [7]`. However the critical difference is that the latter code does *not* write *through* `x` to modify the underlying list.
- Instead the evaluation of the right hand side (`x + [7]`) builds a *new* list made up of the concatenation of list `x` and list `[7]`.
- A reference to this *new* list overwrites the original reference in `x`. The list referenced by `y` is unchanged. Thus we have the following picture:



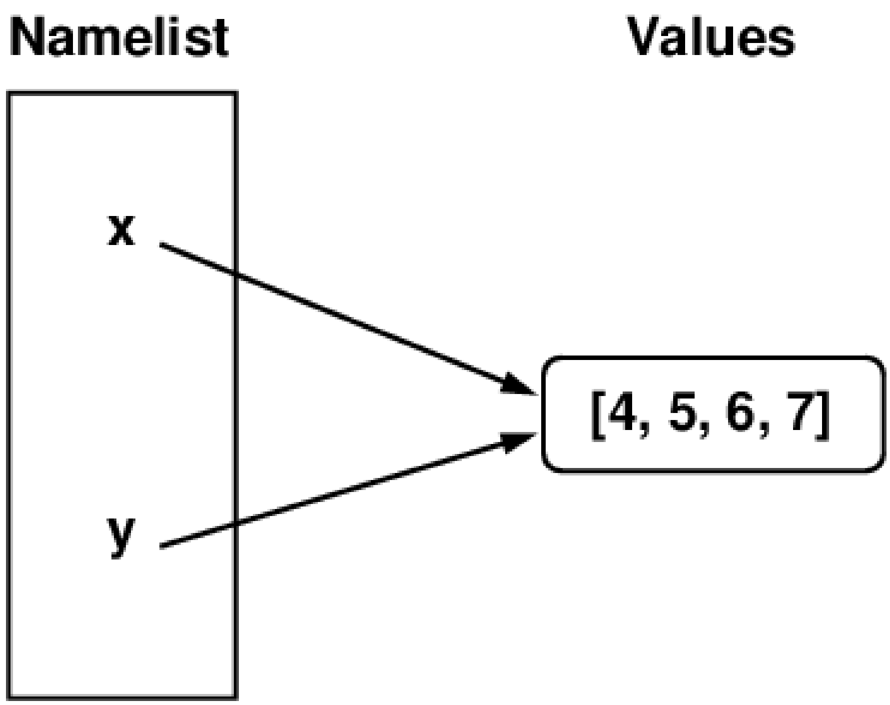
14.7 Another gotcha

- We are not yet finished exploring the subtleties of this behaviour however. Consider the example below.

```
x = [4, 5, 6]
y = x
x += [7]
print(x)
print(y)
```

```
[4, 5, 6, 7]
[4, 5, 6, 7]
```

- Huh? How come `y` was modified in this example? Given the preceding example we might expect `y` to be unchanged despite the change to `x`.
- We might expect `x = x + [7]` (from the previous example) and `x += [7]` (from above) to be equivalent. However they are *not* equivalent and `y` is indeed changed.
- So what is going on? It turns out that the += operator when applied to a list modifies the list *in-place*. This means we effectively write *through* the reference in `x` to append the contents of `[7]` to `x` when we write `x += [7]`. (Thus `x += [7]` is equivalent to `x.append(7)`.)
- Contrast this with where we *overwrite* `x` with a reference to a new list when we write `x = x + [7]`.



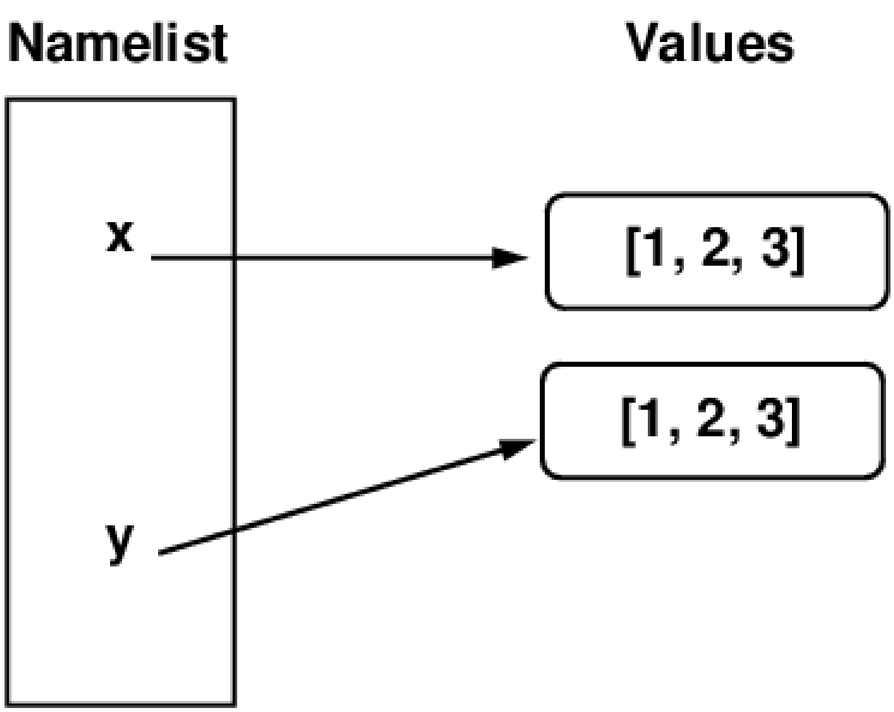
- In summary, an operation on an immutable object must create a new object. An operation on a mutable object *may* create a new object or modify in-place the underlying object. It depends on the particular operator's implementation.

14.8 How do I create a fresh copy of a list?

- As we have seen, to create a *new* copy of a list `x` we cannot simply write `y = x` since this only makes `y` an *alias* for `x` (i.e. they each reference the same object). So how can we create a *new* and *separate* copy of the list referenced by `x`? One approach is to use the slice operator `:` as follows:

```
x = [1, 2, 3]
y = x[:]
print(x)
print(y)
print(x == y)
print(x is y)
```

```
[1, 2, 3]
[1, 2, 3]
True
False
```



CHAPTER

FIFTEEN

LECTURE 3.3 : SHALLOW AND DEEP COPIES

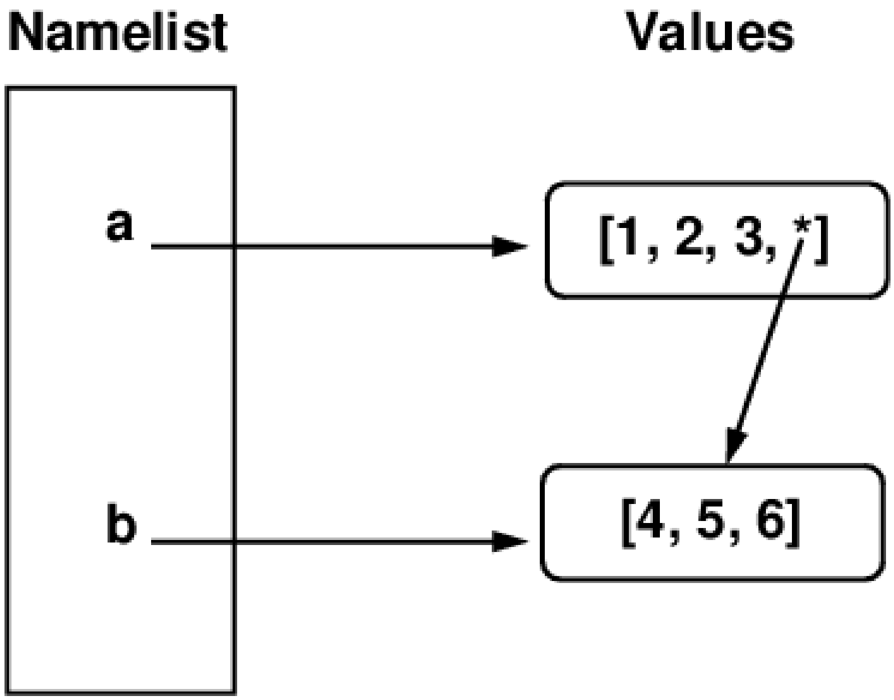
15.1 Introduction

- We conclude our exploration of the relation between variables, references and immutable/mutable objects with another example.

```
a = [1, 2, 3]
b = [4, 5, 6]
a.append(b)
print(a)
```

```
[1, 2, 3, [4, 5, 6]]
```

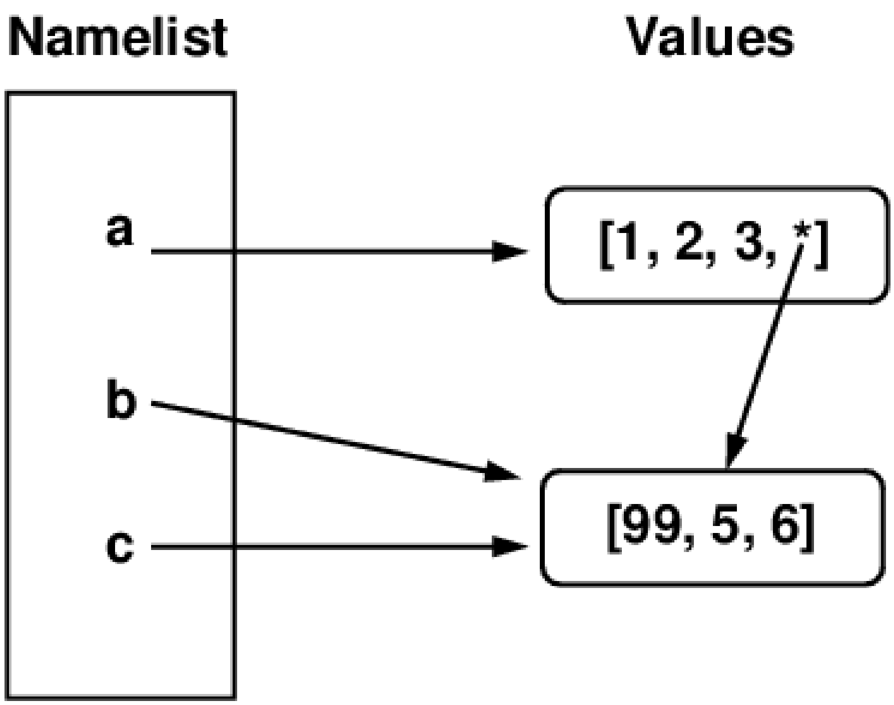
- What is going on here? The diagram below depicts the situation.



- As we can see, when we `a.append(b)` we append a *reference* to `b` to `a`. (To join list `b` to list `a` we would write `a.extend(b)`.)
- Thus after the append operation, `a` contains three integers (really it contains three references to immutable integers) and a reference to the mutable list referenced by `b`.
- The list `[4, 5, 6]` is thus shared by `a` and `b`. Any change to `b` affects `a` as the following example demonstrates.

```
c = b
c[0] = 99
print(c)
print(b)
print(a)
```

```
[99, 5, 6]
[99, 5, 6]
[1, 2, 3, [99, 5, 6]]
```



- It is crucial to note that `a.append(b)` adds a reference to `b` to `a`. It does *not* append a reference to *new copy* of the object referenced by `b` to `a`.

15.2 Shallow copies

- Suppose we wish to copy the list `a` above such that we create new copies of the objects it references rather than duplicating them. Let's try to do it with the slice operator:

```
a = [1, 2, 3]
b = [4, 5, 6]
a.append(b)
c = a[:] # make c a reference to a copy of a
print(a)
print(c)
```

```
[1, 2, 3, [4, 5, 6]]
[1, 2, 3, [4, 5, 6]]
```

```
b[0] = 88
print(a)
```

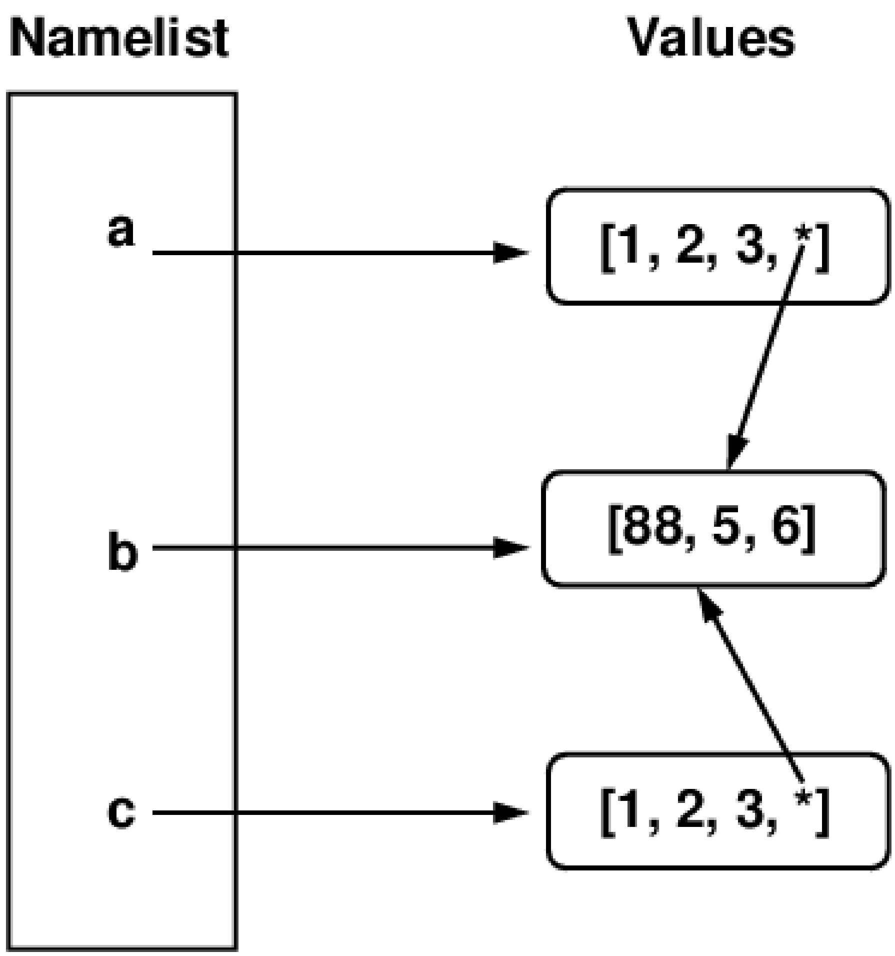
(continues on next page)

(continued from previous page)

```
print(c)
```

```
[1, 2, 3, [88, 5, 6]]
[1, 2, 3, [88, 5, 6]]
```

- That didn't work. What's going on? When we wrote `c = a[:]` the reference to `b` in `a` was copied to `c`. Thus the subsequent change to `b` affected both `a` and `c`.



- When we copy an object where we copy only references it contains and not the referenced objects themselves we are making a *shallow copy*. When we write `c = a[:]` we are making a shallow copy.

15.3 Deep copies

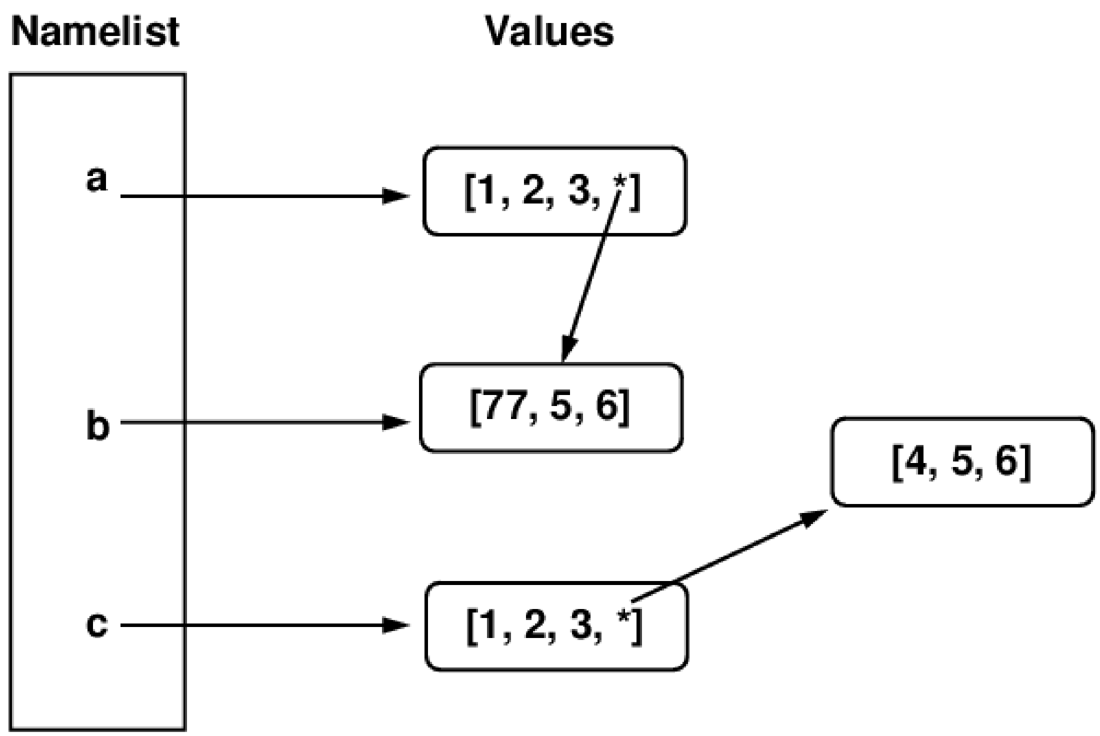
- Suppose we want to make a copy not of the references but the actual objects referenced? How can we do that? How can we make such a *deep copy*?
- Well, in the copy module there is a deepcopy() function that allows us to do just that. Below we illustrate it in action.

```
a = [1, 2, 3]
b = [4, 5, 6]
a.append(b)
print(a)
print(b)
```

```
[1, 2, 3, [4, 5, 6]]
[4, 5, 6]
```

```
from copy import deepcopy
c = deepcopy(a)
b[0] = 77
print(a)
print(c)
```

```
[1, 2, 3, [77, 5, 6]]
[1, 2, 3, [4, 5, 6]]
```



- Above we see that the list referenced by `c` does *not* contain a reference to `b` (it is unaffected by `b[0] = 77`). Instead `c` contains a reference to a new and separate copy of the list referenced from `a`.
- Note this form of copying is slower than the usual approach since it involves following all references in an object and creating new copies of the referenced objects.
- However, where independence from the source object is required in the copied object, it is a deep copy that must be implemented.

CHAPTER

SIXTEEN

LAB 3.2 : (DEADLINE FRIDAY 4 FEBRUARY 23:59)

- Upload your code to Einstein to have it verified.

16.1 More list comprehensions

- Write a program called `wordcomps_032.py` that reads words from `stdin` (one word per line) and stores them all in a list.
- Using *list comprehensions* and ignoring differences in case, have your program print the following:
    1. The shortest word that contains all vowels ('aeiou').
    2. A count of all the words that end in 'iary'.
    3. A list of the words that contain most e's.
- Your program should produce the following output when run against dictionary05.txt:

```
$ python3 wordcomps_032.py < dictionary05.txt
Shortest word containing all vowels: Sequoia
Words ending in iary: 14
Words with most e's: ['dereference', 'teleconference']
```

16.2 Reverse words

- Write a program called `reversecomp_032.py` that reads words from `stdin` (one word per line) and stores them all in a list.
- Making use of a *list comprehension* and ignoring differences in case, have the program print a list of all words that are at least five characters long and whose reverse also occurs in the list.
- Your program should produce the following output when run against dictionary05.txt:

```
$ python3 reversecomp_032.py < dictionary05.txt
['Ababa', 'civic', 'Damon', 'Hannah', 'lager', 'leper', 'level',
↪'lever', 'madam', 'minim', 'nomad', 'radar', 'refer', 'regal',
↪'repel', 'revel', 'rever', 'rotor', 'tenet']
```

- Note that palindromes will appear in the program output since by definition their reverse is in the list of words.
- Note there is a timeout on the program checker that will halt your program if it does not *efficiently* produce its answer.
- You might use *binary search* (as covered in CA116) over the list of sorted words in order to efficiently solve this problem. Here is some code you might use:

```python
# Binary search (adapted from CA116)
def binsearch(query, sorted_list):

    '''Return True if query in sorted_list else False'''

    low = 0
    high = len(sorted_list) - 1

    while low <= high:
        mid = (low + high) // 2

        # print(f'{low} {mid} {high}')

        if sorted_list[mid] < query:
            # Search RHS
            low = mid + 1

        elif sorted_list[mid] > query:
            # Search LHS
            high = mid - 1

        else:
            # Found it
            return True

    # Not found
    return False
```

- Using a dictionary or a set to solve the exercise is not permitted.

16.3 Censor

- Write a program called `censor_032.py` that reads a list of censored strings from a file supplied on the command line.
- The program should then read the text supplied on `stdin` and output the same but with each censored string replaced by a string of ampersands of the same length. For example:

```
$ cat censor_input_00_032.txt
low
rose
smell
he
```

```
$ cat censor_stdin_00_032.txt
Sonnet 98
by William Shakespeare

From you have I been absent in the spring,
When proud-pied April dress'd in all his trim
Hath put a spirit of youth in every thing,
That heavy Saturn laugh'd and leap'd with him.
Yet nor the lays of birds nor the sweet smell
Of different flowers in odour and in hue
Could make me any summer's story tell,
Or from their proud lap pluck them where they grew;
Nor did I wonder at the lily's white,
Nor praise the deep vermilion in the rose;
They were but sweet, but figures of delight,
Drawn after you, you pattern of all those.
Yet seem'd it winter still, and, you away,
As with your shadow I with these did play.
```

```
$ python3 censor_032.py censor_input_00_032.txt < censor_stdin_00_
↪032.txt
Sonnet 98
by William Shakespeare

From you have I been absent in t@@ spring,
W@@n proud-pied April dress'd in all his trim
Hath put a spirit of youth in every thing,
That @@avy Saturn laugh'd and leap'd with him.
Yet nor t@@ lays of birds nor t@@ sweet @@@@@
Of different f@@@ers in odour and in hue
Could make me any summer's story tell,
Or from t@@ir proud lap pluck t@@m w@@re t@@y grew;
```

(continues on next page)

(continued from previous page)

```
Nor did I wonder at t@@ lily’s white,
Nor praise t@@ deep vermilion in t@@ @@@@;
T@@y were but sweet, but figures of delight,
Drawn after you, you pattern of all those.
Yet seem’d it winter still, and, you away,
As with your shadow I with t@@se did play.
```

- Case should be ignored when looking for censored strings. Original case should be retained however in the output.
- Replace censored strings in the order they are listed i.e. in the example above all instances of *rose* should be replaced before all instances of *smell*.

CHAPTER

SEVENTEEN

LECTURE 4.1 : DICTIONARIES 1

17.1 Introduction

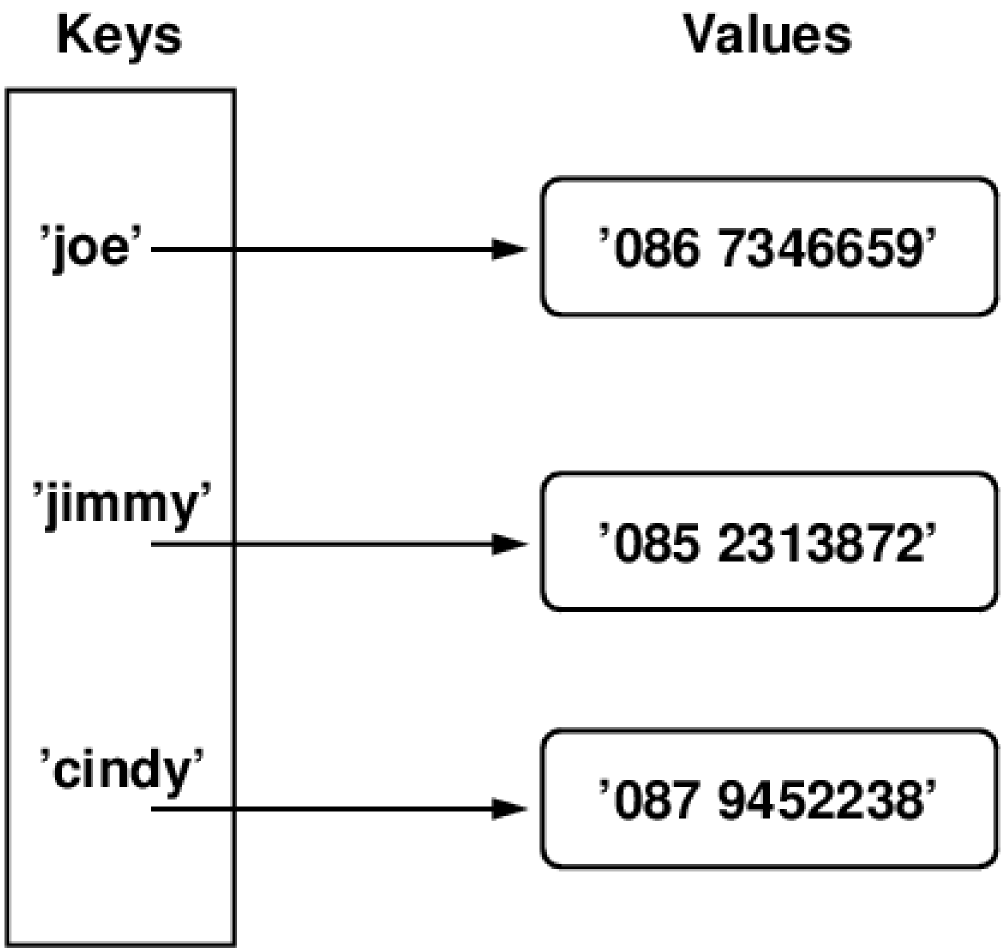
- So far we have met lists, strings and tuples. Each of these is an example of a *data structure*.
- Here we examine another built-in Python data structure: the *dictionary*.
- As we will see dictionaries are an extremely useful and powerful data structure. Knowing when and how to use them effectively will make you a better programmer.

17.2 Dictionaries

- A dictionary is a *collection type* but it is **not** a *sequence type*. That is, its elements are not ordered as they are in a list, string or tuple.
- What is called a dictionary in Python is also sometimes referred to as a *map*, *hashmap*, or *associative array* in other programming languages.
- We can think of a dictionary as a collection of pairs of objects. One element in the pair is the *key* and the other is the *value*. A dictionary thus implements a *mapping* from keys to values.
- When we use a real world dictionary to look up the meaning of a word, the *word* is the *key* and the *meaning of the word* is the *value*.
- A dictionary is designed such that given a *key*, retrieving the associated *value* is a highly efficient operation.
- Once a dictionary has been created we can make changes to the values it contains, add new key-value mappings and remove existing key-value mappings. Clearly a dictionary is a *mutable* type.
- We do not typically know the order in which key-value pairs are stored in a Python dictionary. That information is hidden from us and we should not write programs that rely on it.

17.3 Dictionary example

- We can use a dictionary to implement a simple phone book. A phone book is a mapping from names to phone numbers.
- The dictionary keys are thus names and the dictionary values are phone numbers.
- Once built, the dictionary can be depicted as shown below.



- To find Cindy's phone number in the dictionary we use the key `'cindy'`. That leads us to the value `'087 9452238'`.

17.4 Building dictionaries

- While we use square brackets to create a list, we use curly brackets to create a dictionary.
- Key-value pairs are separated by a colon.
- Keys must be of an immutable type (e.g. strings, integers, tuples) but values can be of any type.
- Let's create the dictionary depicted above.

```
phone_book = { 'joe' : '086 7346659',
    'jimmy' : '085 2313872',
    'cindy' : '087 9452238' }

print(phone_book)
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238'}
```

17.5 Dictionary indexing

- Dictionaries are indexed by keys.

```
print(phone_book['cindy'])
print(phone_book['jimmy'])
```

```
087 9452238
085 2313872
```

- It is an error to index a dictionary with a non-existent key. Specifically, a `KeyError` exception is thrown.

```
print(phone_book['sally'])
```

```
---------------------------------------------------------------------------
↪--------
KeyError                                  Traceback (most recent␣
↪call last)
/tmp/ipykernel_6680/1350413908.py in <module>
```

(continues on next page)

(continued from previous page)

```
----> 1 print(phone_book['sally'])

KeyError: 'sally'
```

- Note dictionaries are *not* sequenced and cannot be indexed by position (as immutable types integers can serve as keys but even then we are indexing by key and not by position).

```
print(phone_book[0])
```

```
---------------------------------------------------------------------
↪---------
KeyError                                  Traceback (most recent␣
↪call last)
/tmp/ipykernel_6680/3343116101.py in <module>
----> 1 print(phone_book[0])

KeyError: 0
```

17.6 Dictionary assignment

- To add an additional mapping to an existing dictionary we use square brackets to index by the new key and then assign the new value (note how *where* a new entry goes in the dictionary is not in general predictable).

```
print(phone_book)
phone_book['louie'] = '087 6551201'
print(phone_book)
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238'}
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238', 'louie': '087 6551201'}
```

17.7 Dictionary updates

- To update an existing key-value pair in the dictionary we index by key and supply the new value.

```
print(phone_book)
phone_book['louie'] = '086 6551201'
print(phone_book)
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238', 'louie': '087 6551201'}
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238', 'louie': '086 6551201'}
```

17.8 Dictionary deletions

- To remove an existing key-value pair in the dictionary we use `del()`.

```
print(phone_book)
# so long louie
del(phone_book['louie'])
print(phone_book)
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238', 'louie': '086 6551201'}
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238'}
```

17.9 Avoiding KeyErrors

- We have a number of options available in order to avoid KeyErrors.

```
def lookup(name):
    # Check membership with in
    if name in phone_book:
        return phone_book[name]
    return None

print(lookup('jimmy'))
print(lookup('sally'))
```

```
085 2313872
None
```

```
def lookup(name):
    try:
        return phone_book[name]
    except KeyError:
        return None

print(lookup('jimmy'))
print(lookup('sally'))
```

```
085 2313872
None
```

```
def lookup(name):
    # get returns None if key not present
    return phone_book.get(name)

print(lookup('jimmy'))
print(lookup('sally'))
```

```
085 2313872
None
```

17.10 Fancy value types

- While keys must be an immutable type values can be any type (strings, tuples, lists, even dictionaries).

```
address_book = { 'joe' : ('Dublin', 'Ireland'),
    'henri' : ('Paris', 'France') }

print(address_book['joe'])
print(address_book['henri'][0])
print(address_book['henri'][1])
```

```
('Dublin', 'Ireland')
Paris
France
```

17.11 Dictionary size

- The `len()` function returns the number of key-value pairs in a dictionary.

```
print(phone_book)
print(len(phone_book))
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238'}
3
```

17.12 Dictionary methods

- We can retrieve a list of all of a dictionary's keys using the `keys()` method.
- We can retrieve a list of all of a dictionary's values using the `values()` method.
- Keys and values returned by the `keys()` and `values()` methods are in corresponding order.

```
print(phone_book)
print(phone_book.keys())
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238'}
dict_keys(['joe', 'jimmy', 'cindy'])
```

```
print(phone_book)
print(phone_book.values())
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238'}
dict_values(['086 7346659', '085 2313872', '087 9452238'])
```

- We can retrieve a list of key-value pairs (tuples) from a dictionary using the `items()` method.

```
print(phone_book)
print(phone_book.items())
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238'}
dict_items([('joe', '086 7346659'), ('jimmy', '085 2313872'), (
↪'cindy', '087 9452238')])
```

17.13 Iterating over a dictionary

- We can use a `for` loop to iterate over the items in a dictionary (key-value pairs) and use multiple assignment to handily access every key and corresponding value.

```
print(phone_book)
print(phone_book.items())
for k, v in phone_book.items():
    print(f'{k} ---> {v}')
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238'}
dict_items([('joe', '086 7346659'), ('jimmy', '085 2313872'), (
↪'cindy', '087 9452238')])
joe ---> 086 7346659
jimmy ---> 085 2313872
cindy ---> 087 9452238
```

CHAPTER

EIGHTEEN

LECTURE 4.2 : DICTIONARIES 2

18.1 Sorting dictionary items on keys

- Suppose we want to print the contents of a dictionary sorted on keys. For example, suppose we want to print out the contents of our phone book in increasing alphabetical order of the names (keys). How would we do that? Well let's try calling `sorted()` on the dictionary's item list and see if that puts things in the desired order.

```
phone_book = { 'joe' : '086 7346659',
   'jimmy' : '085 2313872',
   'cindy' : '087 9452238' }

print(phone_book)
print(phone_book.items())
print(sorted(phone_book.items()))
```

```
{'joe': '086 7346659', 'jimmy': '085 2313872', 'cindy': '087␣
↪9452238'}
dict_items([('joe', '086 7346659'), ('jimmy', '085 2313872'), (
↪'cindy', '087 9452238')])
[('cindy', '087 9452238'), ('jimmy', '085 2313872'), ('joe', '086␣
↪7346659')]
```

- Great. Calling `sorted()` on the items in the phone book sorts them in increasing order of the first member of each returned tuple i.e. the name. This is exactly what we want. So let's employ that approach.

```
for k, v in sorted(phone_book.items()):
   print(f'{k} ---> {v}')
```

```
cindy ---> 087 9452238
jimmy ---> 085 2313872
joe ---> 086 7346659
```

18.2 Sorting dictionary items on values

- Suppose we want to print the contents of a dictionary sorted not on keys but on values. How can we do that? Just calling `sorted()` on items won't work as it sorts on the name.

```
print(phone_book.items())
print(sorted(phone_book.items()))
```

```
dict_items([('joe', '086 7346659'), ('jimmy', '085 2313872'), (
↪'cindy', '087 9452238')])
[('cindy', '087 9452238'), ('jimmy', '085 2313872'), ('joe', '086␣
↪7346659')]
```

- So what can we do? We need to tell `sorted()` to sort on the *second* component of each item returned by `items()` (the phone number). Can we do that?
- We can sort on an arbitrary data member of an object by specifying a custom `key()` function when we invoke the `sorted()` function.
- It is the job of this `key()` function to return the item we wish to sort on.
- In the above example we wish to sort on the second item in a tuple so our `key()` function should be defined as shown below.

```
def tagger(item):
   return item[1]

# sort on the value (i.e. phone number)
for k, v in sorted(phone_book.items(), key=tagger):
   print(f'{k} ---> {v}')
```

```
jimmy ---> 085 2313872
joe ---> 086 7346659
cindy ---> 087 9452238
```

18.3 Key function intuition

- Imagine we wanted to sort all students in a class on how far they lived from DCU. We would ask each student "How far do you live from DCU?". We would tag each student with their answer e.g. 5 if they lived 5 miles from DCU, 4 if they lived 4 miles away, etc.
- With an appropriate tag attached to each student, we would then sort them in increasing/decreasing order of their tags.
- Passing a `key()` function to `sorted()` does a similar tagging job. The job of the `key()` function is to tag each object passed to it. For each object passed to it, it returns the corresponding tag.
- Above we are passing a tuple to the `key()` function (consisting of a dictionary key and value). The `key()` function says "sort on the value" by returning the value.

18.4 More sorting examples

- Below we create a new dictionary and print its contents sorting on keys and values in ascending and descending order.

```
zoo = { 'snakes' : 20,
 'hippos' : 2,
 'tarantulas' : 15,
 'zebras' : 7 }

def tagger(item):
   return item[0]

# increasing key order
for k, v in sorted(zoo.items(), key=tagger):
   print(f'{k} ---> {v}')
```

```
hippos ---> 2
snakes ---> 20
tarantulas ---> 15
zebras ---> 7
```

```
# decreasing key order
for k, v in sorted(zoo.items(), key=tagger, reverse=True):
   print(f'{k} ---> {v}')
```

```
zebras ---> 7
tarantulas ---> 15
```

(continues on next page)

(continued from previous page)

```
snakes ---> 20
hippos ---> 2
```

```
def tagger(item):
    return item[1]

# increasing value order
for k, v in sorted(zoo.items(), key=tagger):
    print(f'{k} ---> {v}')
```

```
hippos ---> 2
zebras ---> 7
tarantulas ---> 15
snakes ---> 20
```

```
# decreasing value order
for k, v in sorted(zoo.items(), key=tagger, reverse=True):
    print(f'{k} ---> {v}')
```

```
snakes ---> 20
tarantulas ---> 15
zebras ---> 7
hippos ---> 2
```

18.5 Tabulating dictionary keys and values

- Suppose we want to both sort and neatly print dictionary keys and corresponding values. How can we do that?
- Firstly we need to work out the width of the widest value in our dictionary. We can do so as shown below.

```
print(zoo.values())
print(max(zoo.values()))
print(str(max(zoo.values())))
print(len(str(max(zoo.values()))))
max_v_width = len(str(max(zoo.values())))
```

```
dict_values([20, 2, 15, 7])
20
20
2
```

- Next we need to work out the width of the widest key in our dictionary. We can do so as shown below (note again the use of an appropriate `key()` function).

```
print(zoo.keys())
print(max(zoo.keys()))
print(max(zoo.keys(), key=len))
print(len((max(zoo.keys(), key=len))))
max_k_width = len((max(zoo.keys(), key=len)))
```

```
dict_keys(['snakes', 'hippos', 'tarantulas', 'zebras'])
zebras
tarantulas
10
```

- We now format our output using the above widths.

```
for k, v in sorted(zoo.items()):
    print(f'{k:>{max_k_width}s} ---> {v:>{max_v_width}d}')
```

```
    hippos --->  2
    snakes ---> 20
tarantulas ---> 15
    zebras --->  7
```

18.6 Other dictionary methods

- There are more dictionary methods. You can look them up using `help` or the `pydoc` command.

```
help(dict)
```

```
Help on class dict in module builtins:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping␣
↪object's
```

(continues on next page)

(continued from previous page)

```
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
 |      d = {}
 |      for k, v in iterable:
 |          d[k] = v
 |  dict(**kwargs) -> new dictionary initialized with the␣
↪name=value pairs
 |      in the keyword argument list.  For example:  dict(one=1,␣
↪two=2)
 |
 |  Built-in subclasses:
 |      StgDict
 |
 |  Methods defined here:
 |
 |  __contains__(self, key, /)
 |      True if the dictionary has the specified key, else False.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate␣
↪signature.
 |
 |  __ior__(self, value, /)
 |      Return self|=value.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
```

(continues on next page)

(continued from previous page)

```
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(self, /)
 |      Return a reverse iterator over the dict keys.
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(...)
 |      D.__sizeof__() -> size of D in memory, in bytes
 |
 |  clear(...)
 |      D.clear() -> None.  Remove all items from D.
 |
 |  copy(...)
 |      D.copy() -> a shallow copy of D
 |
 |  get(self, key, default=None, /)
 |      Return the value for key if key is in the dictionary, else␣
↪default.
 |
 |  items(...)
 |      D.items() -> a set-like object providing a view on D's␣
↪items
 |
```

(continues on next page)

(continued from previous page)

```
 |  keys(...)
 |      D.keys() -> a set-like object providing a view on D's keys
 |
 |  pop(...)
 |      D.pop(k[,d]) -> v, remove specified key and return the␣
↪corresponding value.
 |
 |      If key is not found, default is returned if given,␣
↪otherwise KeyError is raised
 |
 |  popitem(self, /)
 |      Remove and return a (key, value) pair as a 2-tuple.
 |
 |      Pairs are returned in LIFO (last-in, first-out) order.
 |      Raises KeyError if the dict is empty.
 |
 |  setdefault(self, key, default=None, /)
 |      Insert key with a value of default if key is not in the␣
↪dictionary.
 |
 |      Return the value for key if key is in the dictionary, else␣
↪default.
 |
 |  update(...)
 |      D.update([E, ]**F) -> None.  Update D from dict/iterable E␣
↪and F.
 |      If E is present and has a .keys() method, then does:  for␣
↪k in E: D[k] = E[k]
 |      If E is present and lacks a .keys() method, then does: ␣
↪for k, v in E: D[k] = v
 |      In either case, this is followed by: for k in F:  D[k] =␣
↪F[k]
 |
 |  values(...)
 |      D.values() -> an object providing a view on D's values
 |
 |  ------------------------------------------------------------------
↪-------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  fromkeys(iterable, value=None, /) from builtins.type
 |      Create a new dictionary with keys from iterable and values␣
↪set to value.
```

(continues on next page)

(continued from previous page)

```
 |
 |  ----------------------------------------------------------------
↪-------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for␣
↪accurate signature.
 |
 |  ----------------------------------------------------------------
↪-------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

CHAPTER

NINETEEN

LAB 4.1 (DEADLINE MONDAY 14 FEBRUARY 23:59)

- Upload your code to Einstein to have it verified.

19.1 Contact list

- Write a program called `contacts_041.py` that takes the name of a file of contacts as a single command line argument. Each line of the contacts file consists of a name and phone number.
- Your program must read each contact and store it in a dictionary. The dictionary's keys are names and the dictionary's values are the corresponding phone numbers.
- Once the dictionary has been constructed the program should read all lines from `stdin`. Each line consists of a single name.
- For each name the program should retrieve and print the corresponding phone number. If a name cannot be mapped to a phone number the program should print `No such contact`.
- For example, after constructing a dictionary from `contacts_input_00_041.txt` your program should produce the output below when reading from `contacts_stdin_00_041.txt`:

```
$ cat contacts_input_00_041.txt
Sue 085-6442378
Jimmy 086-1223277
Maggie 087-8822001
Amy 087-3240516
Wendy 086-9112645
Sean 085-3445123
```

```
$ cat contacts_stdin_00_041.txt
Jimmy
Sue
Sean
```

(continues on next page)

(continued from previous page)

```
Gwen
Wendy
Tommy
Maggie
Amy
```

```
$ python3 contacts_041.py contacts_input_00_041.txt < contacts_
↪stdin_00_041.txt
Name: Jimmy
Phone: 086-1223277
Name: Sue
Phone: 085-6442378
Name: Sean
Phone: 085-3445123
Name: Gwen
No such contact
Name: Wendy
Phone: 086-9112645
Name: Tommy
No such contact
Name: Maggie
Phone: 087-8822001
Name: Amy
Phone: 087-3240516
```

19.2 Fancy contact list

- Write a new version of the above program called `fancy_041.py`. It functions similarly except contact details consist of two items: a phone number and an email address.
- For example, after constructing a dictionary from `fancy_input_00_041.txt` your program should produce the output below when reading from `fancy_stdin_00_041.txt`:

```
$ cat fancy_input_00_041.txt
Sue 085-6442378 sue@eircom.net
Jimmy 086-1223277 james@apple.com
Maggie 087-8822001 maggie@microsoft.com
Amy 087-3240516 amy@rte.ie
Wendy 086-9112645 wendy@physics.dcu.ie
Sean 085-3445123 sean@tcd.ie
```

```
$ cat fancy_stdin_00_041.txt
Jimmy
```

(continues on next page)

(continued from previous page)

```
Sue
Sean
Gwen
Wendy
Tommy
Maggie
Amy
```

```
$ python3 fancy_041.py fancy_input_00_041.txt < fancy_stdin_00_041.
↪txt
Name: Jimmy
Phone: 086-1223277
Email: james@apple.com
Name: Sue
Phone: 085-6442378
Email: sue@eircom.net
Name: Sean
Phone: 085-3445123
Email: sean@tcd.ie
Name: Gwen
No such contact
Name: Wendy
Phone: 086-9112645
Email: wendy@physics.dcu.ie
Name: Tommy
No such contact
Name: Maggie
Phone: 087-8822001
Email: maggie@microsoft.com
Name: Amy
Phone: 087-3240516
Email: amy@rte.ie
```

19.3 Word frequencies

- Write a program called `words_041.py` that calculates the frequency of words in lines of text read from `stdin`.
- Your program must store the totals as values in a dictionary where the corresponding words are the keys.
- Once totals have been calculated the program should print all words in alphabetical order along with corresponding totals. For example:

```
$ cat words_stdin_00_041.txt
This is a test. And a test is this.
How many tests are required to prove something?
I wonder if there is something wrong with my code...?
I shouldn't worry. I'll sort it in the end.
```

```
$ python3 words_041.py < words_stdin_00_041.txt
a : 2
and : 1
are : 1
code : 1
end : 1
how : 1
i : 2
i'll : 1
if : 1
in : 1
is : 3
it : 1
many : 1
my : 1
prove : 1
required : 1
shouldn't : 1
something : 2
sort : 1
test : 2
tests : 1
the : 1
there : 1
this : 2
to : 1
with : 1
wonder : 1
worry : 1
wrong : 1
```

- Hints:
  1. Convert all words to lower case. `'A'` and `'a'` should not be counted as separate words.
  2. Remember to strip any surrounding punctuation from words. You may find `string.punctuation` useful for this task.
  3. Use the `sorted()` function for sorting.

19.4 Vowel frequencies

- Write a program called `vowels_041.py` that calculates the frequency of each of the vowels *a*, *e*, *i*, *o*, *u* in lines of text read from `stdin`.
- Your program must store the totals as values in a dictionary where the corresponding vowels are the keys.
- When run against gettysburg.txt your program should produce the following output (note the output must be neatly *tabulated* and values must be displayed in *decreasing* order):

```
$ python3 vowels_041.py < gettysburg.txt
e : 167
a : 105
o :  95
i :  69
u :  21
```

- Hints:
  1. Convert all words to lower case. `'A'` and `'a'` should not be counted as separate vowels.
  2. Printing the dictionary items in order of decreasing values is tricky. You will most likely have to study the documentation for the `sorted()` function.
  3. Note how the output must be neatly tabulated. Remember that to print an integer `x` to a given width `w` you can use something like: `print(f'{x:{w}}')`.

CHAPTER

TWENTY

LECTURE 4.3 : SETS

20.1 Introduction

- A *set* is a *collection* of objects of arbitrary type.
- The objects in a set are called its *members*.
- A key property of a set is that it may contain only one copy of a particular object: duplicates are not allowed.
- A set with no elements is *the empty set*.

20.2 Python sets

- A set is created by calling the `set()` constructor or by using curly brackets.
- Note a `dictionary` is also created using curly brackets but while a dictionary consists of *key-value* pairs separated by a colon, the members of a `set` are separated by commas.
- The `set()` constructor requires an iterable be passed to it. Each object in the iterable becomes a member of the set.
- There is no order to the members of a set (like a dictionary).

```
vowel_set = set('aeiou') # vowel_set = {'aeiou'} is equivalent
print(vowel_set)
print(len(vowel_set))
```

```
{'o', 'e', 'a', 'i', 'u'}
5
```

- A set cannot contain duplicates (duplicates in the original iterable are absent in the corresponding set).

```
number_set = set([1, 2, 3, 1, 2, 7])
print(number_set)
```

```
{1, 2, 3, 7}
```

```
s = "Some characters"
char_set = set(s)
print(char_set)
print(len(char_set))
print(len(s))
```

```
{'o', 't', 'e', 'a', 's', ' ', 'm', 'r', 'c', 'S', 'h'}
11
15
```

20.3 Set membership

- We can use the `in` operator to check for membership of a `set`.

```
print(vowel_set)
print('a' in vowel_set)
print('x' in vowel_set)
```

```
{'o', 'e', 'a', 'i', 'u'}
True
False
```

20.4 Iteration over a set

- We can use a `for` loop to iterate over the members of a `set`.
- Since a `set` is unordered, the order in which members are visited by a `for` loop is unknown (and you should not write code that relies on the order).

```
print(vowel_set)
for v in vowel_set:
    print(v)
```

```
{'o', 'e', 'a', 'i', 'u'}
o
e
a
i
u
```

20.5 Set methods

- We can remove an item from a set with `remove()` and add an item with `add()`.

```
vowel_set = set('aeiou')
vowel_set.remove('u')
print(vowel_set)
vowel_set.add('u')
print(vowel_set)
```

```
{'o', 'e', 'a', 'i'}
{'o', 'e', 'a', 'i', 'u'}
```

- We can find the *intersection* of two sets using the `intersection()` method.
- The intersection of sets A and B is the set of elements that are in both A and B.

```
a_set = set('adcb')
b_set = set('ecdf')
print(a_set.intersection(b_set))
print(a_set & b_set) # equivalent to above
```

```
{'d', 'c'}
{'d', 'c'}
```

- We can find the *union* of two sets using the `union()` method.
- The union of sets A and B is the set of elements that are in A or B.

```
a_set = set('adcb')
b_set = set('ecdf')
print(a_set.union(b_set))
print(a_set | b_set) # equivalent to above
```

```
{'d', 'f', 'b', 'e', 'a', 'c'}
{'d', 'f', 'b', 'e', 'a', 'c'}
```

- We can find the *set difference* between two sets using the `difference()` method.
- A set difference B is the set of elements in A but not in B.
- B set difference A is the set of elements in B but not in A.

```
a_set = set('adcb')
b_set = set('ecdf')
print(a_set.difference(b_set))
print(a_set - b_set) # equivalent to above
print(b_set.difference(a_set))
print(b_set - a_set) # equivalent to above
```

```
{'a', 'b'}
{'a', 'b'}
{'e', 'f'}
{'e', 'f'}
```

- We can check whether one set is a *subset* of another using the `issubset()` method.
- Set A is a subset of set B if every member of A is also a member of B.

```
a_set = set('abcd')
b_set = set('bd')
print(a_set.issubset(b_set))
print(a_set <= b_set) # equivalent to above
print(b_set.issubset(a_set))
print(b_set <= b_set) # equivalent to above
```

```
False
False
True
True
```

- We can check whether one set is a *superset* of another using the `issuperset()` method.
- Set A is a superset of set B if every member of B is also a member of A.

```
a_set = set('abcd')
b_set = set('bd')
print(a_set.issuperset(b_set))
print(a_set >= b_set) # equivalent to above
print(b_set.issuperset(a_set))
print(b_set >= a_set) # equivalent to above
```

```
True
True
False
False
```

20.6 Examples

- Write a function that returns `True` if a string contains a vowel and `False` otherwise.

```
vowel_set = set('aeiou')
def contains_vowel(s):
    return len(vowel_set & set(s.lower())) > 0

print(contains_vowel('Owl'))
print(contains_vowel('lynx'))
```

```
True
False
```

- Write a function that returns the number of unique vowels in a sentence.

```
vowel_set = set('aeiou')
def unique_vowels(s):
   return len(vowel_set & set(s.lower()))

print(unique_vowels('Owl'))
print(unique_vowels('lynx'))
print(unique_vowels('Oodles of poodles'))
```

```
1
0
2
```

- Write a function that returns `True` if a string contains all the vowels and `False` otherwise.

```
vowel_set = set('aeiou')
def contains_all_vowels(s):
   return vowel_set & set(s.lower()) == vowel_set

print(contains_all_vowels('sequioa'))
print(contains_all_vowels('oak'))
```

```
True
False
```

- Write a function that returns `True` if a list of numbers contains duplicates and `False` otherwise.

```
def has_duplicates(numbers):
   return len(numbers) > len(set(numbers))

print(has_duplicates([3,1,9,7,4]))
print(has_duplicates([3,1,4,7,4]))
```

```
False
True
```

- Write a function that takes a single string `s` as its argument and returns a dictionary mapping each character in `s` to the number of times it occurs in `s`. Ignore case.

```
def counter(s):
    sl = s.lower()

    # make use of a dictionary comprehension
    d = {c : sl.count(c) for c in set(sl)}
    return d

print(counter('Totally'))
print(counter(''))
```

```
{'o': 1, 't': 2, 'a': 1, 'y': 1, 'l': 2}
{}
```

20.7 Set methods

```
help(set)
```

```
Help on class set in module builtins:

class set(object)
 |  set() -> new empty set object
 |  set(iterable) -> new set object
 |
 |  Build an unordered collection of unique elements.
 |
 |  Methods defined here:
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
```

(continues on next page)

(continued from previous page)

```
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iand__(self, value, /)
 |      Return self&=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate␣
↪signature.
 |
 |  __ior__(self, value, /)
 |      Return self|=value.
 |
 |  __isub__(self, value, /)
 |      Return self-=value.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __ixor__(self, value, /)
 |      Return self^=value.
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __rand__(self, value, /)
 |      Return value&self.
 |
 |  __reduce__(...)
```

(continues on next page)

(continued from previous page)

```
|      Return state information for pickling.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(...)
|      S.__sizeof__() -> size of S in memory, in bytes
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  add(...)
|      Add an element to a set.
|
|      This has no effect if the element is already present.
|
|  clear(...)
|      Remove all elements from this set.
|
|  copy(...)
|      Return a shallow copy of a set.
|
|  difference(...)
|      Return the difference of two or more sets as a new␣
↪set.
|
|      (i.e. all elements that are in this set but not the␣
↪others.)
|
|  difference_update(...)
|      Remove all elements of another set from this set.
|
|  discard(...)
```

(continues on next page)

(continued from previous page)

```
|       Remove an element from a set if it is a member.
|
|       If the element is not a member, do nothing.
|
|   intersection(...)
|       Return the intersection of two sets as a new set.
|
|       (i.e. all elements that are in both sets.)
|
|   intersection_update(...)
|       Update a set with the intersection of itself and␣
↪another.
|
|   isdisjoint(...)
|       Return True if two sets have a null intersection.
|
|   issubset(...)
|       Report whether another set contains this set.
|
|   issuperset(...)
|       Report whether this set contains another set.
|
|   pop(...)
|       Remove and return an arbitrary set element.
|       Raises KeyError if the set is empty.
|
|   remove(...)
|       Remove an element from a set; it must be a member.
|
|       If the element is not a member, raise a KeyError.
|
|   symmetric_difference(...)
|       Return the symmetric difference of two sets as a new␣
↪set.
|
|       (i.e. all elements that are in exactly one of the␣
↪sets.)
|
|   symmetric_difference_update(...)
|       Update a set with the symmetric difference of itself␣
↪and another.
|
|   union(...)
|       Return the union of sets as a new set.
|
```

(continues on next page)

(continued from previous page)

```
 |      (i.e. all elements that are in either set.)
 |
 |  update(...)
 |      Update a set with the union of itself and others.
 |
 |  ----------------------------------------------------------
↪------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  ----------------------------------------------------------
↪------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for␣
↪accurate signature.
 |
 |  ----------------------------------------------------------
↪------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

CHAPTER

TWENTYONE

LAB 4.2 (DEADLINE MONDAY 14 FEBRUARY 23:59)

- Upload your code to Einstein to have it verified.

21.1 Numbers to words

- Write a Python program called `nums2words_v1_042.py` that maps lines of numbers to corresponding lines of text. The lines of numbers are read from `stdin`. The number of lines is arbitrary.
- You can assume that everything read from `stdin` is a number and that number is in the range 0-10.
- For example:

```
$ cat nums2words_v1_stdin_00_042.txt
5 1 2 0 6 7 9 10 8 8 3 4
4 3 9 8 2 6 3 0  7 7 1 9
```

```
$ python3 nums2words_v1_042.py < nums2words_v1_stdin_00_042.txt
five one two zero six seven nine ten eight eight three four
four three nine eight two six three zero seven seven one nine
```

21.2 Numbers to words (with unknowns)

- Write a Python program called `nums2words_v2_042.py` that maps lines of numbers to corresponding lines of text. The lines of numbers are read from `stdin`. The number of lines is arbitrary.
- This time the data read from `stdin` can be anything. Every number in the range 0-10 should be mapped to corresponding text. Everything else however should be mapped to `'unknown'`.

- For example:

```
$ cat nums2words_v2_stdin_00_042.txt
5 1 2 b 6 7 99 10 8 8 3 4
4 3 9 8 2 6 3 0 x 7 1 9 11
```

```
$ python3 nums2words_v2_042.py < nums2words_v2_stdin_00_042.txt
five one two unknown six seven unknown ten eight eight three four
four three nine eight two six three zero unknown seven one nine␣
↪unknown
```

21.3 Numbers to words (with translation)

- Write a Python program called `nums2words_v3_042.py` that maps lines of numbers to corresponding lines of text. The lines of numbers are read from `stdin`. The number of lines is arbitrary.
- You can assume that everything read from `stdin` is a number and that number is in the range 0-10.
- The program takes as an argument a file that contains a mapping from English words to their translation. Your program must make appropriate use of this mapping.
- Here is an example:

```
$ cat nums2words_v3_input_00_042.txt
zero naid
one aon
two do
three tri
four ceathar
five cuig
six se
seven seacht
eight ocht
nine naoi
ten deich
```

```
$ cat nums2words_v3_stdin_00_042.txt
5 1 2 0 6 7 9 10 8 8 3 4
4 3 9 8 2 6 3 0  7 7 1 9
```

```
$ python3 nums2words_v3_042.py nums2words_v3_input_00_042.txt <␣
↪nums2words_v3_stdin_00_042.txt
cuig aon do naid se seacht naoi deich ocht ocht tri ceathar
```

(continues on next page)

(continued from previous page)

```
ceathar tri naoi ocht do se tri naid seacht seacht aon naoi
```

21.4 More numbers to words

- Write a Python program called `nums2words_v4_042.py` that maps lines of numbers to corresponding lines of text. The lines of numbers are read from `stdin`. The number of lines is arbitrary.
- You can assume that everything read from `stdin` is a number and that number is in the range 0-100.
- For example:

```
$ cat nums2words_v4_stdin_00_042.txt
5 11 22 0 66 17 99 100 18 68 73 44
4 35 91 83 27 66 30 0 71 17 16 91
```

```
$ python3 nums2words_v4_042.py < nums2words_v4_stdin_00_042.txt
five eleven twenty-two zero sixty-six seventeen ninety-nine one␣
↪hundred eighteen sixty-eight seventy-three forty-four
four thirty-five ninety-one eighty-three twenty-seven sixty-six␣
↪thirty zero seventy-one seventeen sixteen ninety-one
```

21.5 Swapping dictionary keys and values

- Write a Python *module* `swap_v1_042.py` that defines a function called `swap_keys_values()`. The function accepts a single argument: a dictionary `d`.
- The function must return a new dictionary whose *keys* are the *values* in `d` and whose *values* are the corresponding *keys* in `d`.
- For example:

```
#!/usr/bin/env python3

from swap_v1_042 import swap_keys_values

def main():

    my_dict = {'a' : 4, 'b' : 7, 'c' : 10}
```

(continues on next page)

(continued from previous page)

```
    new_dict = swap_keys_values(my_dict)
    print(sorted(new_dict.items()))

if __name__ == '__main__':
    main()
```

```
[(4, 'a'), (7, 'b'), (10, 'c')]
```

- You may assume that all values in `d` are unique and immutable.

21.6 Swapping more dictionary keys and values

- Write a new Python module called `swap_v2_042.py` that defines a function called `swap_unique_keys_values()`. The function accepts a single argument: a dictionary `d`.
- The function must return a new dictionary whose *keys* are the *unique values* in `d` and whose *values* are the corresponding *keys* in `d`.
- For example:

```
#!/usr/bin/env python3

from swap_v2_042 import swap_unique_keys_values

def main():
    my_dict = {'a' : 4, 'b' : 7, 'c' : 10, 'd' : 7}
    new_dict = swap_unique_keys_values(my_dict)
    print(sorted(new_dict.items()))

if __name__ == '__main__':
    main()
```

```
[(4, 'a'), (10, 'c')]
```

- You may assume all values in `d` are immutable. There may however be duplicate values. Since values are to become keys this presents a problem: we cannot have duplicate keys. Our solution is to simply ignore duplicate values: they will not be used as keys.

CHAPTER

TWENTYTWO

LECTURE 5.1 : FUNCTIONS

22.1 Introduction

- Functions facilitate a **divide-and-conquer** approach to problem-solving: when confronted with a complex problem we break it down into a number of simpler subproblems.
- The solution to each subproblem is implemented as a function.
- This approach allows us to create better quality, more readable code that is simpler to write and maintain.
- Inside a function we place code that typically takes some information (passed to it in the form of arguments), uses that information to calculate a result and returns that result to a caller.
- To use the function we do not need to know how it calculates its result, we merely need to know how to invoke the function.
- This hiding of implementation details is called **encapsulation**.
- Duplication of code is to be avoided.
- Once a function has been coded it can be called from anywhere in your program.
- Related functions can also be placed in a module to be imported and invoked by other programs.
- Thus functions support code **sharing** and **reuse**.
- Because of their simplicity, individual functions are more easily tested and verified compared to more complex, monolithic code blocks.
- Using functions thus produces more **secure** and **reliable** code.

22.2 Functions

- Let's write a function that converts Celsius to Fahrenheit.

```
def celsius2fahrenheit(c):
    f = c * 1.8 + 32
    print(f'That is {f:.1f} degrees Fahrenheit')

celsius = 21
celsius2fahrenheit(celsius)
```

```
That is 69.8 degrees Fahrenheit
```

- Above we see the definition of a function `celsius2fahrenheit()`.
- The `c` variable in the definition of the function is called a *parameter*.
- A parameter is essentially a variable that is *local* to the function: the `c` variable thus cannot be referenced outside of the `celsius2fahrenheit()` function.
- Variables which are created within a function are said to be *local* to the function.
- Local variables are created during the execution of the function, and do not survive after the invocation has completed.
- Values for parameters, supplied at invocation, are called *arguments* or *actual parameters*.
- We see that when we call the `celsius2fahrenheit()` function we pass to it an *argument*.
- The argument in this case is the `celsius` variable.
- What is the relationship between the argument `celsius` and the parameter `c`?
- Well, when the function is invoked the contents of `celsius` *are copied into* `c`.
- By default, arguments and parameters are matched by position i.e. the first argument is copied into the first parameter, the second argument is copied into the second parameter, etc.
- Note how the `celsius2fahrenheit()` function does not return any data to the caller.
- It calculates the temperature in Fahrenheit and prints it.
- Functions which do not return any value are called *procedures*.
- (It turns out that functions that lack a return statement do in fact return a value to their caller in Python, that value is `None`.)

- Procedures effect a change in the world. For example, they display something on a screen, change the values of variables, change the contents of a file, or delete a file from a disk.
- Functions on the other hand merely inspect the world without changing it. The result of their inspection is a value. We can use the function invocation anywhere an expression of the type returned by the function can be used.
- Another way to describe the difference between procedures and functions is to say that procedures are like complex *statements* while functions are like complex *expressions*.
- Suppose we want our function to make available, to the caller, the newly calculated temperature in Fahrenheit. How can we do that?

22.3 Return values

```
def celsius2fahrenheit(c):
    f = c * 1.8 + 32
    return f

celsius = 21
fahrenheit = celsius2fahrenheit(celsius)
print(f'That is {fahrenheit:.1f} degrees Fahrenheit')
```

```
That is 69.8 degrees Fahrenheit
```

- Above we have added a `return` statement to our `celsius2fahrenheit()` function.
- The effect is to *hand back* to the caller of the function the value of `c * 1.8 + 32`.
- Since the function returns a value its caller is expected to collect that value.
- Above we see the caller collects the returned value and assigns the variable `fahrenheit` to it.

22.4 Multiple return statements

```
def bigger(a, b):
    if a > b:
        return a
    return b

print(f'The bigger value is {bigger(3, 4)}')
```

```
The bigger value is 4
```

- As illustrated above, a function may have more than one `return` statement.
- Execution of the function terminates and control returns to the caller as soon as the first `return` statement is executed. As soon as a function has an answer it can return it.

22.5 Returning multiple values

- A function can return only a single *object*.
- If we wish to return multiple values, such as in the example below where we require a function to return both the volume and surface area of a sphere, then we must wrap up those values in a single object and return that object.
- In the case below the object returned is a tuple. The caller unpacks the values from the tuple using multiple assignment and prints them separately.
- Note that although a tuple is used above to wrap the two returned values, any object capable of capturing the two values will do e.g. a list, dictionary, custom object, etc.

```
from math import pi

def sphere(r):
    v = (4.0 / 3.0) * pi * r**3
    sa = 4.0 * pi * r**2
    return v, sa # return a tuple

volume, area = sphere(1)
print(f'The volume is {volume:.1f} m^3')
print(f'The surface area is {area:.1f} m^2')
```

```
The volume is 4.2 m^3
The surface area is 12.6 m^2
```

22.6 Variable scope

- When a function is executed it creates its own *scope*.
- Any variables that come into existence over the course of execution of the function belong to its *namespace* and are *local* to it.
- A variable comes into existence once it is *assigned* to a value.

- Variables that are local to a function can only be referenced within that function and are inaccessible outside that function.
- If a function is invoked repeatedly, its local variables and parameters are created anew for each invocation, and they disappear when the function has completed its execution for that invocation.
- The value of a local variable does not carry over to any following invocation of the function.
- Below we see a failed attempt to reference a local variable outside of the function where it resides.

```
def sphere(r):
   v = (4.0 / 3.0) * pi * r**3
   sa = 4.0 * pi * r**2
   return v, sa # return a tuple

sphere(1)
print(v)
```

```
------------------------------------------------------------------
↪---------
NameError                                 Traceback (most recent␣
↪call last)
/tmp/ipykernel_6734/2718990090.py in <module>
      5
      6 sphere(1)
----> 7 print(v)

NameError: name 'v' is not defined
```

22.7 Global and local scope

- A variable that is visible across the program namespace (rather than being confined to a particular function's namespace) is called a *global* variable.

```
x = 42

def foo():
   print(x)

foo()
print(x)
```

```
42
42
```

22.8 Scope puzzle #1

- It is the act of assignment of a variable to a value that causes the variable to come into existence.
- Thus the assignment in the function below creates a *new* local variable `x` that is distinct from the global one.

```
x = 42

def foo():
    x = 33
    print(x)

foo()
print(x)
```

```
33
42
```

22.9 Scope puzzle #2

- Below we confuse Python and it does not like it.
- The reference to `x` on line 4 is a reference to the *global* variable `x`.
- The assignment `x = 33` creates a *new* local variable `x`.
- Thus `x` is both local and global in the same function.
- Python does not permit this kind of ambiguity and deems `x` to be local throughout the function.
- Thus the reference to `x` on line 4 is an error since the local variable `x` has not yet been assigned to a value.

```
1 x = 42
2
3 def foo():
4     print(x)
```

(continues on next page)

(continued from previous page)

```
    x = 33
    print(x)

foo()
print(x)
```

```
---------------------------------------------------------------------------
↪---------
UnboundLocalError                         Traceback (most recent␣
↪call last)
/tmp/ipykernel_6734/2821423412.py in <module>
      6     print(x)
      7
----> 8 foo()
      9 print(x)

/tmp/ipykernel_6734/2821423412.py in foo()
      2
      3 def foo():
----> 4     print(x)
      5     x = 33
      6     print(x)

UnboundLocalError: local variable 'x' referenced before assignment
```

22.10 Scope puzzle #3

- This is a similar case to the one above. The reference to `x` on the right hand side of `x = x + 1` is a reference to the *global* variable `x`.
- The *assignment* `x = x + 1` however creates a *new* local variable `x`.
- Thus `x` is both local and global in the same function and Python is again unhappy (and says so).

```
x = 42

def foo():
    x = x + 1 # same issue with x += 1
    print(x)

foo()
print(x)
```

```
---------------------------------------------------------------------------
↪---------
UnboundLocalError                         Traceback (most recent␣
↪call last)
/tmp/ipykernel_6734/1231945412.py in <module>
      5     print(x)
      6
----> 7 foo()
      8 print(x)

/tmp/ipykernel_6734/1231945412.py in foo()
      2
      3 def foo():
----> 4     x = x + 1 # same issue with x += 1
      5     print(x)
      6

UnboundLocalError: local variable 'x' referenced before assignment
```

22.11 Scope puzzle #4

- So how can a function update a global variable if the assignment creates a *new* variable?!
- By marking the variable as *global*.
- Line 4 marks `x` in this function as always a referring to the global variable `x`.
- Thus the assignment `x = x + 1` in this case does *not* create a new local variable and instead updates the global variable `x`. Phew!

```
1 x = 42
2
3 def foo():
4     global x
5     x = x + 1
6     print(x)
7
8 foo()
9 print(x)
```

```
43
43
```

22.12 Programs, modules, functions

- As the programs you write get longer and more complex you may want to group related functions into a file to facilitate maintenance and sharing.
- You may also have a handy function that you would like to use in several programs without having to copy its definition into each.
- In Python we place related function definitions in a *module* from where we can *import* them into a *program*.
- (Going further we can group related modules into packages.)
- For example, below we import the `math` module before using its `cos` function and its `pi` definition.

```
import math

print(math.cos(math.pi))
```

```
-1.0
```

- Should we wish to import only particular functions or definitions from a module we can do that too.
- We can then reference them directly in our program without going through the module reference.

```
from math import sin, pi

print(sin(3*pi/2))
```

```
-1.0
```

22.13 Programs as modules

- The Python interpreter maintains a global variable `__name__`.
- We have some code in `hello.py`.
- If `hello.py` is *executed as a program* then `__name__ == '__main__'`.
- If `hello.py` is *imported as a module* then `__name__ == '__hello__'`.
- We can take advantage of this as follows.

```
if __name__ == '__main__':
    main()
```

- If `hello.py` is being executed then run `main`.
- Otherwise we are simply importing the module (so we can use its functions) and we do so without running `main`.
- That's why we always include this snippet of code in our programs/modules.

22.14 Module/program template

- The following template will work irrespective of whether you are asked to write a program or a module.

```
# Imports go here...

# Global variables go here...

# Function definitions go here...

def main():
    # Put here the code that calls the other functions...
    pass

# If I am a program call main
if __name__ == '__main__':
    main()
```

CHAPTER

TWENTYTHREE

LECTURE 5.2 : IMMUTABLE AND MUTABLE FUNCTION ARGUMENTS

23.1 Introduction

- We look at the options available for passing arguments to functions.
- We look at the implications of passing mutable compared to immutable arguments.
- We look at default and keyword parameters.
- We look at a dangerous trap that you should not fall into.

23.2 Immutable arguments

- Below we attempt an alternative approach to returning a value from a function.
- We compute an answer and write it to the the argument passed to the function.
- Will it work? Will the function caller see the answer?
- Let's try it and see!

```
def celsius2fahrenheit(temperature):
    temperature = temperature * 1.8 + 32

temperature = 21
celsius2fahrenheit(temperature)
print(f'That is {temperature:.1f} degrees Fahrenheit')
```

```
That is 21.0 degrees Fahrenheit
```

- Hmm. That did not work. Why not?
- On line 5 we passed an argument `temperature` to the function.
- This `temperature` is a reference to the number `21`.
- On line 1 a *new* local variable called `temperature` is created (it is local to the `celsius2fahrenheit()` function) and into it is copied the reference contained in the `temperature` created on line 4.
- Thus we effectively have two `temperature` variables. One is global (line 4) and one is local (line 1).
- **Each of these is a reference to the same immutable number**.
- Inside the function on line 2 we *overwrite* the local variable `temperature` with a *new* reference to a *new* number.
- However, this update is *invisible* to the caller of the function and has *no effect* on the contents of the global `temperature` which continues to reference the number 21.
- Note that it is always only a reference that is copied from an argument to a parameter and *no new copy* of the underlying object is created.
- The argument and the parameter are instead *aliases* for the same object.

23.3 Mutable arguments

- By contrast, when we pass a *mutable* argument to a function then the function *can* make changes to it that are visible to the caller.
- The crucial difference here is that the called function *does not overwrite the reference* passed to it *but instead writes through the reference* passed to it to update the underlying object.

```
def add2list(alist):
    alist.append(99)

blist = [1, 2, 3]
add2list(blist)
print(blist)
```

```
[1, 2, 3, 99]
```

- When we pass the argument `blist` to the `add2list` function the reference in `blist` is copied into the parameter `alist`.
- Thus both `blist` and `alist` reference the same object.
- When we execute line 2 we write *through* the `alist` reference to append to the underlying object.
- On returning to `main` the change is visible as we have written *through* and *not overwritten* the reference passed to the function.

23.4 Default parameter values

- It is possible in Python to assign a default value to a function parameter.
- If the function call does not supply a corresponding argument then the parameter is assigned its default value for that invocation of the function.
- If a corresponding argument is supplied then its value overrides the default value for that invocation of the function.
- Thus parameters in the function definition with associated default values are *optional* while parameters in the function definition without associated default values are *required*.
- **Parameters with default values must appear rightmost in the parameter list in the function definition.**

23.5 Keyword arguments

- By default, arguments are mapped according to their position to corresponding parameters.
- It is however also possible to map arguments to parameters using keywords.
- Thus it is possible to order arguments differently to parameters as long as `parameter=argument` pairs are supplied in the function call.
- Any "traditional" arguments to the left of any `parameter=argument` pairs are matched by position with parameters.
- **Any parameter=argument pairs must appear rightmost in the argument list in the function call.**
- **Multiple values for a parameter are obviously not allowed. (How would Python know which one to use?!)**

23.6 Exercise

- Provide the output of the following code (or indicate an error where applicable).

```
def arithmetic(a, b, c=3, d=4):
    return a + b + c + d

print(arithmetic(1, 2, 5, 6))

print(arithmetic(3, 4, 5))

print(arithmetic(3, 4))

print(arithmetic(3, 4, d=3))

print(arithmetic(b=5, a=4, d=2, c=1))

print(arithmetic(a=2, b=4, 6))

print(arithmetic(6, a=2, b=4))

print(arithmetic(b=2, a=4, c=6))

print(arithmetic(b=5, 2, 5))
```

23.7 The mutable default parameter value trap

- Consider the output of the following.

```
def add_animal(animal, zoo=[]):
  zoo.append(animal)
  return zoo

animals = add_animal('leopard')
print(animals)

animals = add_animal('giraffe', ['hyena'])
print(animals)

animals = add_animal('tarantula')
print(animals)
```

```
['leopard']
['hyena', 'giraffe']
['leopard', 'tarantula']
```

- The output above is surprising.
- As programmers we do not like surprises.
- Passing a second argument to `add_animal` is optional.
- If no argument is supplied then the corresponding parameter takes on the value `[]` i.e. the empty list.
- We can see that when we first call the function and pass it `'leopard'` it hands back the list `['leopard']`.
- This makes sense as `zoo` defaults to the empty list.
- The second time we call the function we pass it `'giraffe'` and a list `['hyena']` and it hands us back the list `['hyena', 'giraffe']`.
- So far so good. This all makes sense.
- In the final call we pass the function `'tarantula'` and we expect to be returned the list `['tarantula']` after `zoo` defaults to the empty list.
- Instead we get back the list `['leopard', 'tarantula']`.
- Why is that? What's going on? I'm surprised and I don't like it!
- Clearly the `zoo=[]` default assignment of the empty list, when no corresponding argument is supplied, does not work as intended.
- The problem is the this default value, an empty list **is initialised only once** by Python.
- It is initialised when the function `def` is first encountered.
- This means that the default mutable values *have a memory* and anything added to them will stay there.
- **Never not use mutable types as default parameter values.**
- We fix the problem as follows.

```
def add_animal(animal, zoo=None):

  # Create a new empty list each time one is required
  if zoo is None:
```

(continues on next page)

(continued from previous page)

```
        zoo = []

    zoo.append(animal)
    return zoo

animals = add_animal('leopard')
print(animals)

animals = add_animal('giraffe', ['hyena'])
print(animals)

animals = add_animal('tarantula')
print(animals)
```

```
['leopard']
['hyena', 'giraffe']
['tarantula']
```

- That's more like it! Now I can sleep easy.

CHAPTER

TWENTYFOUR

SAMPLE LAB EXAM (DEADLINE MONDAY 14 FEBRUARY 23:59)

24.1 Before starting

- The exam runs 1400-1550.
- Answer all questions.
- Upload all code to Einstein.
- All *lab exam rules* apply.
- To pass all tests submit from L101, L114, L125 or L128.

24.2 Question 1 [25 marks]

- You have some water and some buckets to fill.
- Write a program called `water_051.py` that reads two lines of text from `stdin`.
- Line 1 contains a single integer, N, the number of litres of water available. N is in the range 0-1000.
- Line 2 lists the capacity in litres of one or more buckets. The capacity of each bucket is specified by a positive integer.
- Buckets must be filled in the order specified on line 2.
- Your program should output the number of buckets that can be completely filled before you run out of water.
- In this example we have 10 litres of water. We fill the first bucket (taking 6 litres), we fill the second bucket (taking another 2 litres) but we run out of water before we have completely filled the third bucket (it requires 5 litres). We output 2 (the number of buckets completely filled):

```
$ cat water_stdin_00_051.txt
10
6 2 5 1 1
```

```
$ python3 water_051.py < water_stdin_00_051.txt
2
```

24.3 Question 2 [25 marks]

- A hotel needs your help.
- Write a program called `hotel_051.py` that reads a single line of numbers from `stdin`.
- The first number in each line is, N, the number of rooms in the hotel. Rooms are numbered 1 to N. N is in the range 1-1000.
- Following N is a list of the rooms that are occupied. Occupied room numbers are listed in random order and in the range 1-N. Each occupied room number appears no more than once in the list. It is possible that no rooms are occupied.
- Write a program that reads the above input and outputs the lowest numbered available room or `no room` if none are available.
- In this example there are 5 rooms in the hotel, rooms 4, 2, 1 are occupied. So the lowest numbered available room is 3:

```
$ cat hotel_stdin_00_051.txt
5 4 2 1
```

```
$ python3 hotel_051.py < hotel_stdin_00_051.txt
3
```

- In this example, there are 8 rooms in the hotel and all are occupied so the output is `no room`:

```
$ cat hotel_stdin_01_051.txt
8 7 1 3 4 6 8 5 2
```

```
$ python3 hotel_051.py < hotel_stdin_01_051.txt
no room
```

24.4 Question 3 [25 marks]

- A *pangram* is a phrase that includes at least one occurrence of each of the 26 letters, a-z.
- Write a program called `pangram_051.py` that reads lines of text from `stdin`.
- For each line of text read the program should print `pangram` if that line is a pangram.
- If a line of text is not a pangram the program should print which letters are missing (i.e. those required to make it a pangram). These should be printed in lower case and in increasing alphabetical order.
- A letter occurs in the line of text if it occurs in either upper or lower case.
- You can assume each line contains 1-100 characters.
- For example:

```
$ cat pangram_stdin_00_051.txt
The quick brown fox jumps over the lazy dog.
ThE QUICK brown fox jumps oVer the lazy dog.
ZYXW, vu TSR Ponm lkj ihgfd CBA.
.,?!'" 92384 abcde FGHIJ
```

```
$ python3 pangram_051.py < pangram_stdin_00_051.txt
pangram
pangram
missing eq
missing klmnopqrstuvwxyz
```

24.5 Question 4 [25 marks]

- Write a program called `golf_051.py` that reads an arbitrary number of golf scores from `stdin`.
- Each line consists of a player's name followed by the number of shots taken by that player on each of three golf holes and is structured as follows:

  Player_name shots_hole_1 shots_hole_2 shots_hole_3

- Having read in all lines the program should print the results in ascending order of total shots taken by each player.

- Any player who records an invalid score on any hole is disqualified and omitted from the results table.
- A valid score for each hole is a positive integer.
- If any players have been disqualified their comma-separated names should be printed following the results (in the order encountered when read from `stdin`).
- You can assume the following:
  - There will be no ties.
  - Exactly three scores are entered per player but scores may be invalid.

- For example:

```
$ cat golf_stdin_00_051.txt
Leona Maguire 4 4 4
Tiger 6 7 5
Harold Varner III 2 3 4
Ernie Els 6 6 5
Stephanie Meadow 2 2 3
Sam Burns 7 9 8
```

```
$ python3 golf_051.py < golf_stdin_00_051.txt
Stephanie Meadow: 7
Harold Varner III: 9
Leona Maguire: 12
Ernie Els: 17
Tiger: 18
Sam Burns: 24
```

- For example:

```
$ cat golf_stdin_01_051.txt
Leona Maguire 4 4 4
Tiger 6 7 5
Harold Varner III 2 X 4
Ernie Els 6 6 5
Stephanie Meadow 2 2 3
Sam Burns 7 9 8
```

```
$ python3 golf_051.py < golf_stdin_01_051.txt
Stephanie Meadow: 7
```

(continues on next page)

(continued from previous page)

```
Leona Maguire: 12
Ernie Els: 17
Tiger: 18
Sam Burns: 24
Disqualified: Harold Varner III
```

- For example:

```
$ cat golf_stdin_02_051.txt
Leona Maguire 4 4 4
Tiger 6 7 5
Harold Varner III 2 X 4
Ernie Els 6 6 5
Stephanie Meadow Z 2 3
Sam Burns 7 9 8
```

```
$ python3 golf_051.py < golf_stdin_02_051.txt
Leona Maguire: 12
Ernie Els: 17
Tiger: 18
Sam Burns: 24
Disqualified: Harold Varner III, Stephanie Meadow
```

CHAPTER

TWENTYFIVE

LECTURE 5.3 : MISCELLANEOUS

25.1 Introduction

- We review some previously met and present some new Python functions/objects.
- These may be useful to you when solving programming exercises.

25.2 `range`

- `range` can be useful when you need to generate some integers.
- Use `range(stop)` to generate integers `[0-(stop-1)]`.

```
print(list(range(10)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Use `range(start, stop[, step])` to generate integers `[start, start+step, start+2*step, ..., stop)` (up to but not including `stop`).

```
print(list(range(0, 10, 1))) # equivalent to range(10)
print(list(range(-5, 5, 2)))
print(list(range(5, -5, -2)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[-5, -3, -1, 1, 3]
[5, 3, 1, -1, -3]
```

25.3 `for` loops

- Use a `for` loop when you need to work your way across an iterable collection.

```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

25.4 `break` and `continue`

- Use `break` to exit a loop early (perhaps an answer has been found so there's no point in going further).

```
for i in range(10):
    if i == 5: # we're done
        break
    print(i)
```

```
0
1
2
3
4
```

- Use `continue` to skip to the next iteration of a loop.

```
for i in range(10):
    if i % 2: # skip odd numbers
```

(continues on next page)

(continued from previous page)

```
        continue
    print(i)
```

```
0
2
4
6
8
```

25.5 zip

- Use `zip` to join corresponding elements of iterables into a tuple.

```
numbers = [1, 2, 3, 4, 5]
words = ['one', 'two', 'three', 'four', 'five']

print(list(zip(numbers, words)))

for n, w in zip(numbers, words):
    print(f'{n} ---> {w}')
```

```
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four'), (5, 'five')]
1 ---> one
2 ---> two
3 ---> three
4 ---> four
5 ---> five
```

25.6 enumerate

- Use `enumerate` to associate an index with each element of an iterable yielding a tuple.

```
animals = ['penguins', 'lions', 'snakes']
print(list(enumerate(animals)))

for i, animal in enumerate(animals):
    print(f'At index {i} we find {animal}')
```

```
[(0, 'penguins'), (1, 'lions'), (2, 'snakes')]
At index 0 we find penguins
```

(continues on next page)

(continued from previous page)

```
At index 1 we find lions
At index 2 we find snakes
```

25.7 sorted

- The `sorted` function does not work only on lists, it works on any **iterable**.
- Here we sort the characters in a string.

```
s = 'efdcgba'
print(sorted(s))
print(''.join(sorted(s)))
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
abcdefg
```

- We can also sort items in descending order.

```
s = 'efdcgba'
print(sorted(s, reverse=True))
print(''.join(sorted(s)))
```

```
['g', 'f', 'e', 'd', 'c', 'b', 'a']
abcdefg
```

- By specifying a `key` we can sort on arbitrary item attributes.

```
animals = ['ant', 'aardvark', 'tarantula', 'snake']

print(sorted(animals))
print(sorted(animals, key=len))
```

```
['aardvark', 'ant', 'snake', 'tarantula']
['ant', 'snake', 'aardvark', 'tarantula']
```

- The `key` does not have to be a built-in function.

```
def tagger(s):
    return s.count('t')

animals = ['ant', 'aardvark', 'tarantula', 'snake']

print(sorted(animals, key=tagger))
```

```
['aardvark', 'snake', 'ant', 'tarantula']
```

25.8 Random numbers

- In order to test our code or to run simulations we will often find it useful to generate *random* numbers.
- Python provides a `random` module which defines a number of useful methods in this regard.
- The `random()` method returns a random floating point number in the interval [0, 1). (This means 0 is included in the interval but 1 is not.)

```
from random import random

help(random)
```

```
Help on built-in function random:

random() method of random.Random instance
    random() -> x in the interval [0, 1).
```

```
from random import random

for i in range(3):
    print(f'{random():.2f}')
```

```
0.30
0.51
0.95
```

- The sequence appears random because the next number in the sequence cannot be predicted from previous ones.
- However the generated sequence is entirely determined by the initial *seed* supplied to the underlying algorithm.
- Seeding the generator with the same number causes the same sequence to be produced.
- Such generators are therefore referred to as *pseudo random number generators* (PRNGs).

```
from random import seed, random

seed(5)
for i in range(3):
   print(f'{random():.2f}')
```

```
0.62
0.74
0.80
```

```
from random import seed, random

seed(99)
for i in range(3):
   print(f'{random():.2f}')
```

```
0.40
0.20
0.18
```

```
from random import seed, random

seed(5)
for i in range(3):
   print(f'{random():.2f}')
```

```
0.62
0.74
0.80
```

- If we pass no argument to `seed` the PRNG is seeded with the current clock value. This provides enough randomness for most purposes.

25.9 Other `random` methods

- `randint(a,b)` generates a random integer in the range `[a, b]`.

```
from random import randint

help(randint)
```

```
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end␣
↪points.
```

```
for i in range(3):
   print(randint(10, 20))
```

```
20
18
10
```

- `choice(sequence)` returns a random element of *sequence*.

```
from random import choice

help(choice)
```

```
Help on method choice in module random:

choice(seq) method of random.Random instance
    Choose a random element from a non-empty sequence.
```

```
animals = ['llama', 'scorpion', 'bunny']

print(f'My favourite animal is the {choice(animals)}.')
```

```
My favourite animal is the scorpion.
```

- `shuffle(sequence)` shuffles the order of the elements of *sequence* in place (useful for generating permutations of the elements of a sequence).

```
from random import shuffle

help(shuffle)
```

```
Help on method shuffle in module random:

shuffle(x, random=None) method of random.Random instance
    Shuffle list x in place, and return None.

    Optional argument random is a 0-argument function returning a
    random float in [0.0, 1.0); if it is the default None, the
    standard random.random will be used.
```

```
animals = ['llama', 'scorpion', 'bunny']
shuffle(animals)
print(animals)
```

```
['scorpion', 'bunny', 'llama']
```

- `sample(sequence, N)` returns a new sequence containing *N* randomly selected elements of *sequence*.

```
from random import sample

help(sample)
```

```
Help on method sample in module random:

sample(population, k, *, counts=None) method of random.Random␣
↪instance
    Chooses k unique random elements from a population sequence or␣
↪set.

    Returns a new list containing elements from the population␣
↪while
```

(continues on next page)

(continued from previous page)

```
    leaving the original population unchanged.  The resulting list␣
↪is
    in selection order so that all sub-slices will also be valid␣
↪random
    samples.  This allows raffle winners (the sample) to be␣
↪partitioned
    into grand prize and second place winners (the subslices).

    Members of the population need not be hashable or unique.  If␣
↪the
    population contains repeats, then each occurrence is a possible
    selection in the sample.

    Repeated elements can be specified one at a time or with the␣
↪optional
    counts parameter.  For example:

        sample(['red', 'blue'], counts=[4, 2], k=5)

    is equivalent to:

        sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)

    To choose a sample from a range of integers, use range() for␣
↪the
    population argument.  This is especially fast and space␣
↪efficient
    for sampling from a large population:

        sample(range(10000000), 60)
```

```python
animals = ['llama', 'scorpion', 'bunny']

favs = sample(animals, 2)

print(f"My two favourite animals are the {' and ' .join(favs)}.")
```

```
My two favourite animals are the llama and bunny.
```

CHAPTER

TWENTYSIX

SAMPLE LAB EXAM (DEADLINE MONDAY 14 FEBRUARY 23:59)

26.1 Before starting

- The exam runs 1400-1550.
- Answer all questions.
- Upload all code to Einstein.
- All *lab exam rules* apply.
- To pass all tests submit from L101, L114, L125 or L128.

26.2 Question 1 [25 marks]

- Jimmy loves chocolate. It's all he eats. Once he starts a bar he must finish it.
- Each bar of chocolate contains 400 calories.
- Write a program called `chocolate_052.py` that reads from `stdin` an arbitrarily long list daily calorie requirements for Jimmy (one per line).
- Each calorie requirement is an integer in the range 0-100,000.
- For each line read your program should output an integer representing the minimum whole number of bars of chocolate Jimmy must eat to satisfy his calorie requirement on that day.
- For example:

```
$ cat chocolate_stdin_00_052.txt
300
800
```

```
$ python3 chocolate_052.py < chocolate_stdin_00_052.txt
1
2
```

26.3 Question 2 [25 marks]

- A poker hand consists of five unique cards drawn from a standard 52-card deck.
- Each card is represented by two characters. The first character is the rank of the card which is one of *A23456789TJQK*. The second character is the suit of the card which is one of *CDHS*.
- The strength of a hand is the maximum value k such that there are k cards in a hand that have the same rank.
- Write a program called `poker_052.py` that reads a line of text representing a poker hand from `stdin` and outputs its strength.
- For example:

```
$ cat poker_stdin_00_052.txt
AC KD KS KC 3H
```

```
$ python3 poker_052.py < poker_stdin_00_052.txt
3
```

- For example:

```
$ cat poker_stdin_01_052.txt
4H 5C 4C 5S AC
```

```
$ python3 poker_052.py < poker_stdin_01_052.txt
2
```

26.4 Question 3 [25 marks]

- Write a program called `uppers_052.py` that reads strings from `stdin` (one string per line).
- Each string is a sequence of upper and lower case characters.
- Your program must print the longest sequence of contiguous upper case letters contained in each string.
- You can assume the length of the longest sequence is positive and unique.

```
$ cat uppers_stdin_00_052.txt
aBc
AbcdEFGHIjk
```

```
$ python3 uppers_052.py < uppers_stdin_00_052.txt
B
EFGHI
```

26.5 Question 4 [25 marks]

- Runners run some arbitrary number of races (at least one) in a season.
- Write a program called `race_052.py` that reads runners' race times for the season from `stdin`.
- Each line read from `stdin` is structured as follows: *Runner's_name time_1 time_2 time_3* …
- Each name is a single string.
- Each time is in the form *minutes:seconds*.
- Your program must print the name of the runner with the best race time over the course of the season along with that time (you may assume there will always be a clear winner).
- Should any of the times be invalid then the corresponding runner should be ignored.

```
$ cat race_stdin_00_052.txt
Rachel 8:12 8:32 8:00 7:12 8:09
Fred 11:12 11:13 11:14 11:14 11:10
Naomi 8:45 9:01 10:11 8:18 9:00
Jimmy 8:12 8:2b 8:19 7:13 10:11
Ned 7:34 7:00 6:45 7:19 7:01
```

```
$ python3 race_052.py < race_stdin_00_052.txt
Ned : 6:45
```

CHAPTER TWENTYSEVEN

LECTURE 6.1 : REGULAR EXPRESSIONS

27.1 Introduction

- Imagine we have downloaded the CA117 classlist from the web as an HTML document.
- We would like to e-mail all students in the class and included in the HTML document is every student's email address. So far so good.
- Unfortunately, the document is "noisy": 95% of it is HTML mark-up that obscures the e-mail addresses scattered throughout the document.
- What can we do? We could manually scroll through the document looking for e-mail addresses and copy them to a list. That would be a tedious and error-prone task however. Is there an easier way?
- It would be great if we could specify a "pattern" or "template" that matched and extracted just the information in the document that is of interest to us i.e. e-mail addresses.
- If we could specify a general pattern that every e-mail address follows and then extract everything in the document that matches that pattern then we would have a list of just the required e-mail addresses.
- Regular expressions allow us to do just that!

27.2 Regular expressions

- Regular expressions are used to specify patterns for entities we wish to locate and match in a larger string.
- Examples might be dates, times, e-mail addresses, names, credit card numbers, social security numbers, directory paths, file names, etc.
- Once we have defined a suitable regular expression we can ask questions such as the following: Is there a match for this pattern anywhere in the given string?

- Regular expressions also allow us to efficiently find all substrings of a larger string that match the specified pattern.
- This would seem ideal for our task: If we treat the HTML document as a single large string our task is to extract every substring from it that matches the pattern of an e-mail address.

27.3 Defining patterns

- The simplest of patterns takes the form of an ordinary string.
- Below we define a regular expression `r'cat'` to match occurrences of the pattern 'cat' and we call this regular expression `p` (for pattern).
- When defining a pattern we *always* precede it with 'r' in order to indicate to Python that this is a *raw string* (this prevents Python imposing its own interpretation on any special sequences that might arise in the pattern).
- We match this pattern against the string `s` by calling `findall()`. The latter function returns a list of all substrings of `s` that match the defined pattern.
- Two matches, as we might expect, are returned.

```
# We want to find all matches so import the required function
from re import findall

# We will look for matches in here
s = 'A catatonic cat sat on the mat. Catastrophe!'

# Define our pattern in a raw string
p = r'cat'

# Match and print the result
print(findall(p, s))
```

```
['cat', 'cat']
```

27.4 Character classes

- We can define *character classes* to be matched against.
- The character class `[abc]` will match any *single* character a, or b or c.
- The character class `[a-z]` will match any *single* character a through z.
- The class `[a-zA-Z0-9]` will match any alphanumeric character.

- Let's use a character class to match instances of both 'cat' and 'Cat'.

```
# We will look for matches in here
s = 'A catatonic cat sat on the mat. Catastrophe!'

# Match one of 'C' or 'c' followed by 'at'
p = r'[Cc]at'

# Match and print the result
print(findall(p, s))
```

```
['cat', 'cat', 'Cat']
```

27.5 Character class negation

- We can negate character classes by preceding them with the ^ symbol.

```
# We will look for matches in here
s = 'A catatonic cat sat on the mat. Catastrophe!'

# Match anything but 'C' or 'c' followed by 'at'
p = r'[^Cc]at'

# Match and print the result
print(findall(p, s))
```

```
['tat', 'sat', 'mat']
```

27.6 Sequences

- In addition to defining our own character classes we can call upon a predefined set of character classes when constructing regular expressions.
- Such predefined classes are accessed using *special sequences*.

Sequence	Matches
\d	Matches any decimal digit
\D	Matches any non-digit
\s	Matches any whitespace character (e.g. space, tab, newline)
\S	Matches any non-whitespace character
\w	Matches any alphanumeric character
\W	Matches any non-alphanumeric character
\b	Matches any word boundary (a word is an alphanumeric sequence of characters)

```
# Match one digit
p = r'\d'
print(findall(p, '1 and 2 and 34'))
print(findall(p, 'No digits here'))
```

```
['1', '2', '3', '4']
[]
```

```
# Match one non-digit
p = r'\D'
print(findall(p, '1 and 2 and 34'))
print(findall(p, 'No digits here'))
```

```
[' ', 'a', 'n', 'd', ' ', ' ', 'a', 'n', 'd', ' ']
['N', 'o', ' ', 'd', 'i', 'g', 'i', 't', 's', ' ', 'h', 'e',
↪'r', 'e']
```

```
# Match one whitespace character
p = r'\s'
print(findall(p, '1 and 2 and 34'))
print(findall(p, 'No digits here'))
print(findall(p, '1\n2\t3'))
```

```
[' ', ' ', ' ', ' ']
[' ', ' ']
['\n', '\t']
```

```
# Match one non-whitespace character
p = r'\S'
print(findall(p, '1\n2\t3'))
```

```
['1', '2', '3']
```

```
# Match one alphanumeric character
p = r'\w'
print(findall(p, '1 and 2 and 34'))
print(findall(p, '1\n2\t3'))

# Match one non-alphanumeric character
p = r'\W'
print(findall(p, '1 and 2 and 34'))
print(findall(p, '1\n2\t3'))
print(findall(p, '1 < 3'))
```

```
['1', 'a', 'n', 'd', '2', 'a', 'n', 'd', '3', '4']
['1', '2', '3']
[' ', ' ', ' ', ' ']
['\n', '\t']
[' ', '<', ' ']
```

27.7 Metacharacters

- Most characters simply match themselves.
- Exceptions are metacharacters.
- Metacharacters are special characters that do not match themselves but signal that something else should be matched.
- Here are three common examples:

Metacharacter	Matches
^	Matches the beginning of a string
$	Matches the end of a string
.	Matches any character (except a new line)

27.8 A pattern that occurs once or zero times

- We can match a pattern once or zero times with the ? metacharacter.
- Thus we can use ? to effectively make a pattern *optional*.

```
# Match US and IE spelling
p = r'colou?r'
print(findall(p, 'In America they spell it color'))
print(findall(p, 'Over here we spell it colour'))
```

```
['color']
['colour']
```

27.9 Repeating a pattern a fixed number of times

- With regular expressions we can match portions of a pattern multiple times.
- We do so by specifying the number of required matches inside curly brackets.

```
# Match a date
p = r'\d{2}[-/]\d{2}[-/]\d{2}'
print(findall(p, 'Christmas falls on 25-12-21'))
print(findall(p, "Valentine's Day is 14/02/21"))
```

```
['25-12-21']
['14/02/21']
```

27.10 Groups

- If our pattern contains a *group* of characters that must be matched some number of times then we need to enclose the pattern with `(?:` on the left hand side and `)` on the right hand side.

```
# Match a date
p = r'(?:\d{2}[-/]){2}\d{2}'
print(findall(p, 'Christmas falls on 25-12-21'))
print(findall(p, "Valentine's Day is 14/02/21"))
```

```
['25-12-21']
['14/02/21']
```

27.11 Repeating a pattern at least M and at most N times

- If we need to match a pattern at *least* a number of times *N* and at *most* a number of times *N* then we write `{m, n}`.

```
# The more o's in your yahoo the happier you are
p = r'Yahoo{1,3}!'
print(findall(p, 'Yahoo!'))
print(findall(p, 'Yahooo!'))
print(findall(p, 'Yahoooo!'))
print(findall(p, 'Yahooooo!')) # Too happy to match
```

```
['Yahoo!']
['Yahooo!']
['Yahoooo!']
[]
```

27.12 Repeating a pattern zero or more times

- Some *metacharacters* allow us to handle an *arbitrary* number of matches.
- One such metacharacter for specifying a repeated pattern is *.
- The * metacharacter signifies that the preceding pattern can be matched zero or more times.

```
p = r'Yabba Dabba Do*!'
print(findall(p, 'Yabba Dabba Do!'))
print(findall(p, 'Yabba Dabba Doo!'))
print(findall(p, 'Yabba Dabba Dooooooooooo!'))
print(findall(p, 'Yabba Dabba D!')) # We probably shouldn't match␣
↪this
```

```
['Yabba Dabba Do!']
['Yabba Dabba Doo!']
['Yabba Dabba Dooooooooooo!']
['Yabba Dabba D!']
```

27.13 Repeating a pattern one or more times

- Another metacharacter for specifying a repeated pattern is +.
- It signifies that the preceding pattern can be matched an arbitrary number of times but *must be matched at least once*.
- Note the difference between * and +: with * the specified pattern may not be present at all while with + the specified pattern must be present at least once.

```
p = r'Yabba Dabba Do+!'
print(findall(p, 'Yabba Dabba Do!'))
print(findall(p, 'Yabba Dabba Doo!'))
print(findall(p, 'Yabba Dabba Doooooooooo!'))
print(findall(p, 'Yabba Dabba D!')) # We no longer match this
```

```
['Yabba Dabba Do!']
['Yabba Dabba Doo!']
['Yabba Dabba Doooooooooo!']
[]
```

27.14 Examples

- We have just scratched the surface with regular expressions. They are a mini-programming language in themselves. Even with what we have covered so far however there are some useful things we can do…

```
s = """
Jimmy arrived in work at 3.45pm. His phone number is 087 4567890.
Or you can email him at jimmy.murphy@computing.dcu.ie. Mary arrived
at 9.12am. Her phone number is 085 2345678. She can be contacted at
mary.oneill2@rte.ie. Wendy arrived at 11:18am. Her email address is
wendy@google.com. Her phone number is 086 1234567. Jimmy earns 2.00
euro per hour. Mary earns 14.50 euro per hour. Wendy earns 36.00␣
↪euro
per hour. Some people like to include hyphens in their phone␣
↪numbers,
087-6213344, for example. Valid phone numbers begin with 086 or 087
or 085. This is not a phone number 111 1234567. Examples of invalid
times include 3.71am, 30:19pm and 12.3pm and we do not want to␣
↪match
these when looking for times. Examples of valid times are 1.59am␣
↪and
12.00pm. We do not allow leading zeros in the hour part of the time
```

(continues on next page)

(continued from previous page)

```
so 04.13am is invalid, rather it should be 4.13am. Long dates look
like 1 January 2014 and 18 March 2016. Is this a valid phone␣
↪number:
123087 66543789920?
"""
```

- Let's try to extract all of the mobile phone numbers from the above string.

```
phone = r'\b\d{3}\s\d{7}\b'
print(findall(phone, s))
```

```
['087 4567890', '085 2345678', '086 1234567', '111 1234567']
```

- We have two problems.
- Firstly we are failing to collect phone numbers that have a hyphen between their two components.
- Secondly we are collecting numbers that do not look like phone numbers.
- Let's first collect phone numbers that include hyphens.

```
phone = r'\b\d{3}[-\s]\d{7}\b'
print(findall(phone, s))
```

```
['087 4567890', '085 2345678', '086 1234567', '087-6213344', '111␣
↪1234567']
```

- Now let's insist that a phone number begin with one of *085*, *086* or *087*.

```
phone = r'\b(?:085|086|087)[-\s]\d{7}\b'
print(findall(phone, s))
```

```
['087 4567890', '085 2345678', '086 1234567', '087-6213344']
```

- Let's extract all the times.

```
time = r'\b\d{1,2}[.:]\d{2}[ap]m\b'
print(findall(time, s))
```

```
['3.45pm', '9.12am', '11:18am', '3.71am', '30:19pm', '1.59am', '12.
↪00pm', '04.13am', '4.13am']
```

- How can we exclude invalid times?
- We need a regular expression that matches only hours 1-12 and only minutes 00-59.
- We can do so as follows.

```
time = r'\b(?:[1-9]|1[0-2])[.:](?:0[0-9]|[1-5][0-9])[ap]m\b'
print(findall(time, s))
```

```
['3.45pm', '9.12am', '11:18am', '1.59am', '12.00pm', '4.13am']
```

- Let's extract all the rates of pay.

```
pay = r'\b\d{1,2}\.\d{2}\seuro\b'
print(findall(pay, s))
```

```
['2.00\neuro', '14.50 euro', '36.00 euro']
```

- Let's extract the e-mail addresses.

```
email = r'\b(?:\w+\.)*\w+\@\w+\.\w+(?:\.\w+)*\b'
print(findall(email, s))
```

```
['jimmy.murphy@computing.dcu.ie', 'mary.oneill2@rte.ie',
↪'wendy@google.com']
```

- Let's extract long dates.

```
ldate = r'\b\d{1,2}\s(?
↪:January|February|March|April|May|June|July|August|September|October|Novembe
↪s\d{4}'
print(findall(ldate, s))
```

```
['1 January 2014', '18 March 2016']
```

CHAPTER

TWENTYEIGHT

LAB 6.2 (DEADLINE MONDAY 28 FEBRUARY 23:59)

28.1 Function arguments and parameters

- Without running any code, predict the output of each of the following invocations of the `arithmetic()` function.
- If any invocation raises an error, explain the error.

```python
#!/usr/bin/env python3

def arithmetic(p, q, r=5, s=2):
    return r - p + q + s

def main():

    print(arithmetic(1, 2, 5, 6))

    print(arithmetic(3, 4, 5))

    print(arithmetic(3, 4))

    print(arithmetic(3, 4, s=3))

    print(arithmetic(s=5, q=4, p=2, r=1))

    print(arithmetic(q=2, p=4, 6))

    print(arithmetic(6, r=2, p=4))

    print(arithmetic(p=2, q=4, s=6))

    print(arithmetic(p=5, 2, 5))
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

- Once you have a set of predictions, run the code one call at a time to verify your answers are correct.
- There is no Einstein marker for this exercise.

28.2 Perfect numbers

- Write a *function* `sum_factors()` which specifies an integer parameter `n` (assumed positive) and returns the sum of the factors of `n`.
- The factors of an integer `n` are the positive integers that exactly divide `n`, not including `n` itself.
- For example, the factors of 12 are 1, 2, 3, 4, and 6, and so `sum_factors(12)` should return 16.
- Write a boolean-valued *function* `is_perfect()` which specifies an integer parameter `n` (assumed positive) and returns whether `n` is perfect.
- A positive number is perfect if it is equal to the sum of its factors.
- For example 28 is perfect because its factors are 1, 2, 4, 7, 14, and, 28 = 1 + 2 + 4 + 7 + 14.
- Write a program called `perfect_062.py` which reads a list of positive integers from `stdin` (one per line) and for each one prints `True` if it is a perfect number and `False` otherwise.
- Obviously, you want to make good use of the two functions you developed above.

```
$ cat perfect_stdin_00_062.txt
1
12
33550336
10
28
```

```
$ python3 perfect_062.py < perfect_stdin_00_062.txt
False
False
True
False
True
```

28.3 The mutable default parameter value trap

- The behaviour of the code below is not as the programmer intended.
- Identify and make sure you understand the problem.
- Fix the code and call it `mutable_062.py`.

```
#!/usr/bin/env python3

# Append l1 to l2. If l2 not supplied default to empty list.
def append2list(l1, l2=[]):
    for i in l1:
        l2.append(i)
    return l2

def main():
    list1 = ['fly', 'spider']
    nlist = append2list(list1)
    # nlist should be ['fly', 'spider']
    print(nlist)

    list2 = ['lion']
    nlist = append2list(list2, ['antelope'])
    # nlist should be ['antelope', 'lion']
    print(nlist)

    list3 = ['fox', 'chicken']
    nlist = append2list(list3)
    # nlist should be ['fox', 'chicken']
    print(nlist)


if __name__ == '__main__':
    main()
```

```
['fly', 'spider']
['antelope', 'lion']
['fly', 'spider', 'fox', 'chicken']
```

28.4 Overlapping circles

- Write a function called `overlap()` that returns `True` if two circles overlap and `False` otherwise.
- A circle is defined by its centre's x and y coordinates and its radius.
- Thus our function specifies six parameters in the following order: x1, y1, r1, x2, y2, r2 (you must use these parameter names and in the specified order when defining your function).
- By default, circles are centred at (0, 0) and radii are 1.
- Put your function in a module called `circle_062.py`.

```
#!/usr/bin/env python3

from circle_062 import overlap

def main():
    print(overlap())
    print(overlap(10))
    print(overlap(10,10))
    print(overlap(10,10,10))
    print(overlap(10,0,10))
    print(overlap(10,0,1,8,0,1))
    print(overlap(10,0,1,8,0,2))


if __name__ == '__main__':
    main()
```

```
True
False
False
False
True
False
True
```

- Hint: A formula that works out the distance between two points might be handy.

28.5 Quadratic roots

- The roots of the quadratic:

$$f(x) = ax^2 + bx + c$$

are given by:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Write a program called `roots_062.py` that reads a, b, and c from `stdin` (one set per line) and passes them to a function that computes and returns the corresponding roots.
- Should no real roots exist then the program should print 'None'.
- Otherwise both roots should be printed.

```
$ cat roots_stdin_00_062.txt
1 0 -1
1 1 -2
1 1 2
1 2 1
1 6 5
1 5 6
2 -2 -12
```

```
$ python3 roots_062.py < roots_stdin_00_062.txt
r1 = 1.0, r2 = -1.0
r1 = 1.0, r2 = -2.0
None
r1 = -1.0, r2 = -1.0
r1 = -1.0, r2 = -5.0
r1 = -2.0, r2 = -3.0
r1 = 3.0, r2 = -2.0
```

CHAPTER

TWENTYNINE

LECTURE 7.1: INTRODUCING OBJECT-ORIENTED PROGRAMMING

29.1 Classes and objects we have already met

- We have so far been programming with various built-in types: booleans, integers, floats, strings, lists, tuples, dictionaries, sets, etc.
- These are all *class types*. This means any particular example of a boolean, integer, float, etc. is an *object* of the *class* boolean, integer, float, etc.
- *Everything in Python is an object*, or equivalently, *everything in Python is an instance of a particular class*.
- In other words every integer object, e.g. 5, is an *instance* of the class integer and every string object, e.g. 'daisy', is an *instance* of the class string, etc. Let's use Python to verify this is indeed the case:

```
a = False
print(type(a))
```

```
<class 'bool'>
```

```
b = 5
print(type(b))
```

```
<class 'int'>
```

```
c = 3.3
print(type(c))
```

```
<class 'float'>
```

```
d = 'daisy'
print(type(d))
```

```
<class 'str'>
```

```
e = [1, 2, 3]
print(type(e))
```

```
<class 'list'>
```

```
f = (1, 2, 3)
print(type(f))
```

```
<class 'tuple'>
```

```
g = {'name':'Joe'}
print(type(g))
```

```
<class 'dict'>
```

```
h = {1, 2, 3}
print(type(h))
```

```
<class 'set'>
```

29.2 Interacting with objects through methods

- To implement some operation on a particular object we invoke one of its *methods* (in other words *we invoke a method on the object*).
- This involves writing `object_name.method_name(method_arguments)`.
- Which methods can be invoked on any particular object? Well that depends on the class of the object.
- It is the class that defines the behaviours, i.e. methods, that instances of that class support.
- The list of methods defined by a class tells us the things we can do with an instance of that class.
- To see a list of all the methods a particular object supports use `help(class_name)`, e.g. `help(str)` or `help(list)`.

- We have seen previously but, for review purposes, let's invoke some string methods on a particular string object:

```
s = 'This is a string object, an instance of the string class'

print(s.count('a')) # Invoke the count method on object s

print(s.endswith('class')) # Invoke the endswith method on object s

print(s.find('string')) # Invoke the find method on object s

print(s.isnumeric()) # Invoke the isnumeric method on object s

print(s.split()) # Invoke the split method on object s

print(s.swapcase()) # Invoke the swapcase method on object s
```

```
4
True
10
False
['This', 'is', 'a', 'string', 'object,', 'an', 'instance', 'of',
↪'the', 'string', 'class']
tHIS IS A STRING OBJECT, AN INSTANCE OF THE STRING CLASS
```

29.3 Adding a new type to a program

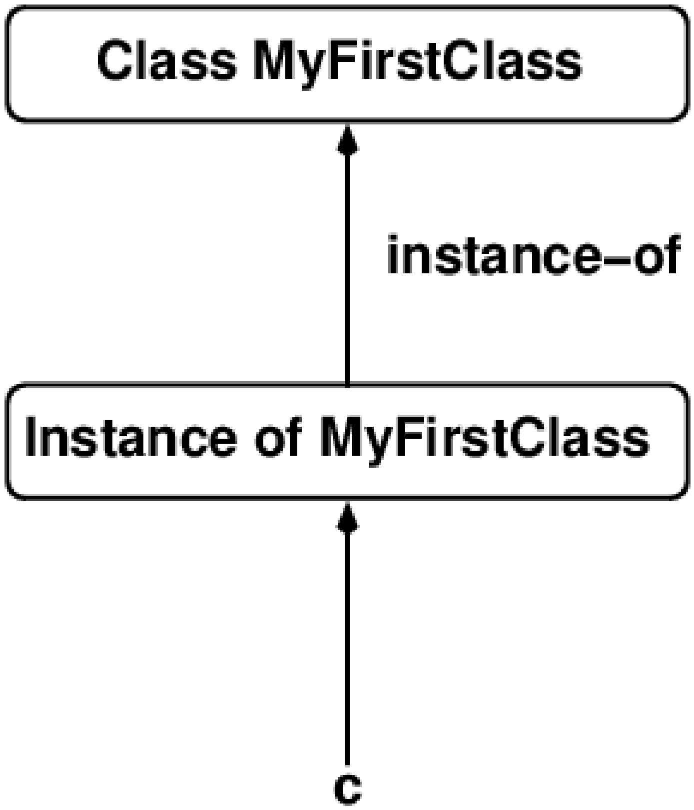
- Knowledge of Python's built-in types and their capabilities is essential to being a good Python programmer.
- However, sometimes, in order to solve a particular problem, Python's built-in types may not entirely suit and we would like to *add our own types* to programs in order to elegantly solve a particular problem.
- Since everything in Python is an object, i.e. an instance of some class type, so too our new types will also be class types.
- How do we define a new class in Python? With the `class` keyword.

```
class MyFirstClass(object):
   pass
```

- The code above defines a new class called `MyFirstClass`.
- The class is empty (it contains only a place-holder `pass` statement so supports only trivial functionality) but it is still a valid class definition.
- (We ignore the `object` reference in the brackets and simply accept that, for now, all of our new class definitions will include it.)
- Note that in Python `class` statements are executable.
- Any `class` statements in a module are executed when that module is imported.
- Once a `class` statement has been executed a new type is defined and objects of that class type can be *instantiated* (i.e. created).
- To create an object of type `MyFirstClass` we call the class as if it were a function (see below).
- Calling the class as a function will instantiate an instance of that class and return a reference to it.
- Here we assign the reference to the variable *c*.
- Inspecting the type of the object *c* shows it is indeed an instance of the `MyFirstClass` class.

```
c = MyFirstClass() # Invoke the class as a function to make an␣
↪object
print(type(c))
```

```
<class '__main__.MyFirstClass'>
```



29.4 Adding another new type to a program

- Let's define a (slightly more useful) class to represent the time in 24-hour format.
- This class defines a method (*functions* inside class definitions are called *methods*) that allows us to set the time on `Time` objects.

```
class Time(object):

   def set_time(time_object, hour, minute, second):
      time_object.hour = hour
      time_object.minute = minute
      time_object.second = second
```

- The `set_time()` method specifies four parameters i.e. it requires four arguments be passed to it in order for it to do its job:
  1. The `time_object` whose time we are setting,

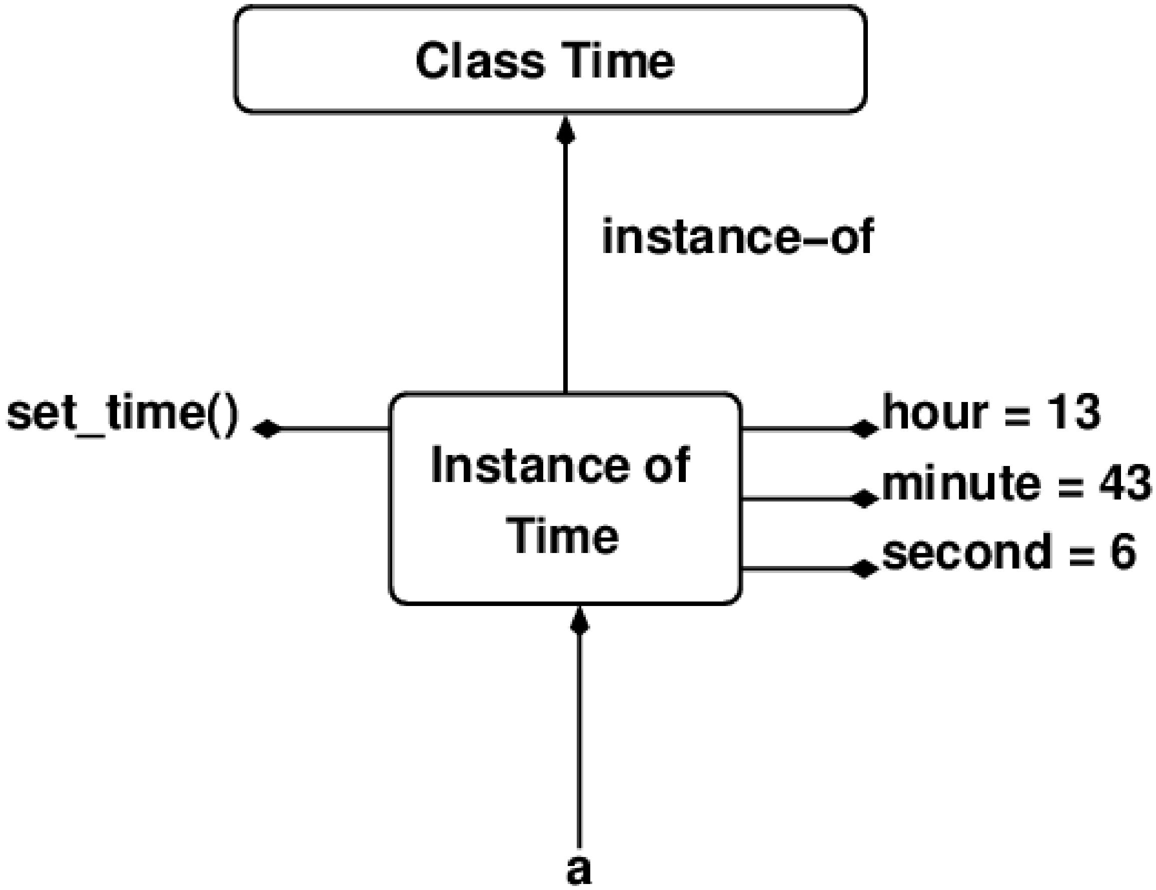
2. the `hour` we want to set on the `time_object`,
3. the `minute` we want to set on the `time_object`,
4. the `second` we want to set on the `time_object`.

- As illustrated above, we access the hour, minute and second *attributes* of the `time_object` using the period operator, e.g. `time_object.minute = minute`.
- Do not confuse the `hour`, `minute` and `second` in the parameter list with `time_object.hour`, `time_object.minute` and `time_object.second`. The former are local variables while the latter are attributes attached to the `time_object`. There is no name clash.
- How can we use this new method? We can do so by writing `class_name.method_name(method_arguments)`.

```
a = Time()
Time.set_time(a, 13, 43, 6)
print(a.hour)
print(a.minute)
print(a.second)
```

```
13
43
6
```

- We invoke the `set_time()` method of the `Time` class on the `a` object.
- The `a` argument to `set_time()` becomes the `time_object` parameter in the method definition (`13` becomes the `hour`, `43` the `minute` and `6` the `second`).
- When the method executes it updates a single object, in this case the object referenced by `a`.
- It updates the object by assigning values to its `hour`, `minute` and `second` attributes.
- We can represent the resulting situation with the following diagram where attributes and (instance) methods are attached to objects:



- Some terminology: the `set_time()` method is called an *instance method* because it acts upon an instance of the class.
- The instance an instance method acts on will always be the first parameter of the method.
- Some more terminology: `time_object.hour`, `time_object.minute` and `time_object.second` are called *instance variables* because they are variables attached to a particular instance of a `Time` object. They are also referred to as an object's *data attributes*.

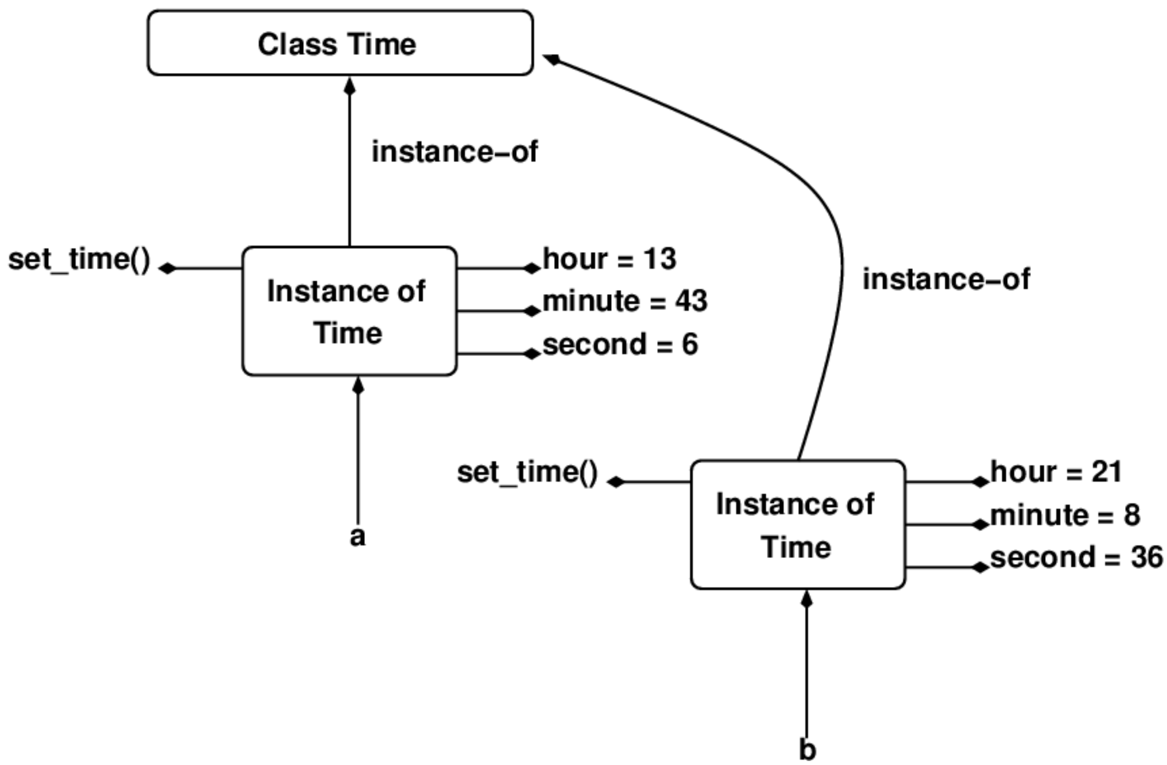
29.5 Multiple objects of the same type

- There is nothing to stop us making multiple objects of type `Time`.
- *Crucially, attached to each object is its own distinct set of instance variables (data attributes).*

```
a = Time()
Time.set_time(a, 13, 43, 6)
print('{:02d}:{:02d}:{:02d}'.format(a.hour, a.minute, a.second))

b = Time()
Time.set_time(b, 21, 8, 36)
print('{:02d}:{:02d}:{:02d}'.format(b.hour, b.minute, b.second))

c = Time()
Time.set_time(c, 3, 4, 7)
print('{:02d}:{:02d}:{:02d}'.format(c.hour, c.minute, c.second))
```

```
13:43:06
21:08:36
03:04:07
```

- Above we instantiate three objects, `a`, `b` and `c`, of the class `Time`.
- We can create as many instances of a class as we wish.
- As a result a class is sometimes referred to as an *object factory*.
- A class definition serves as a *blueprint* for generating objects.
- As we have seen, a new object is generated whenever we call the class as a function.
- Note again how the object being updated serves as an argument to the `set_time()` method. Thus each time the latter method is invoked above it is setting the data attributes of a *different object*.
- We can represent the situation (for `a` and `b`) as follows:



29.6 Adding a method to print the time

- Let's add a function that prints the time.
- Where should we add this function?
- Since the `Time` class encapsulates everything related to processing `Time` objects it makes sense to add it there.
- This is what *object oriented programming* is about.
- Our `Time` class is where we bundle together all `Time`-related data and functions.
- Since functions in a class are called methods, our new function is really a method.
- Here is the resulting expanded class, now boasting two methods.

```
class Time(object):

    def set_time(time_object, hour, minute, second):
        time_object.hour = hour
        time_object.minute = minute
        time_object.second = second

    def show_time(time_object):
```

(continues on next page)

(continued from previous page)

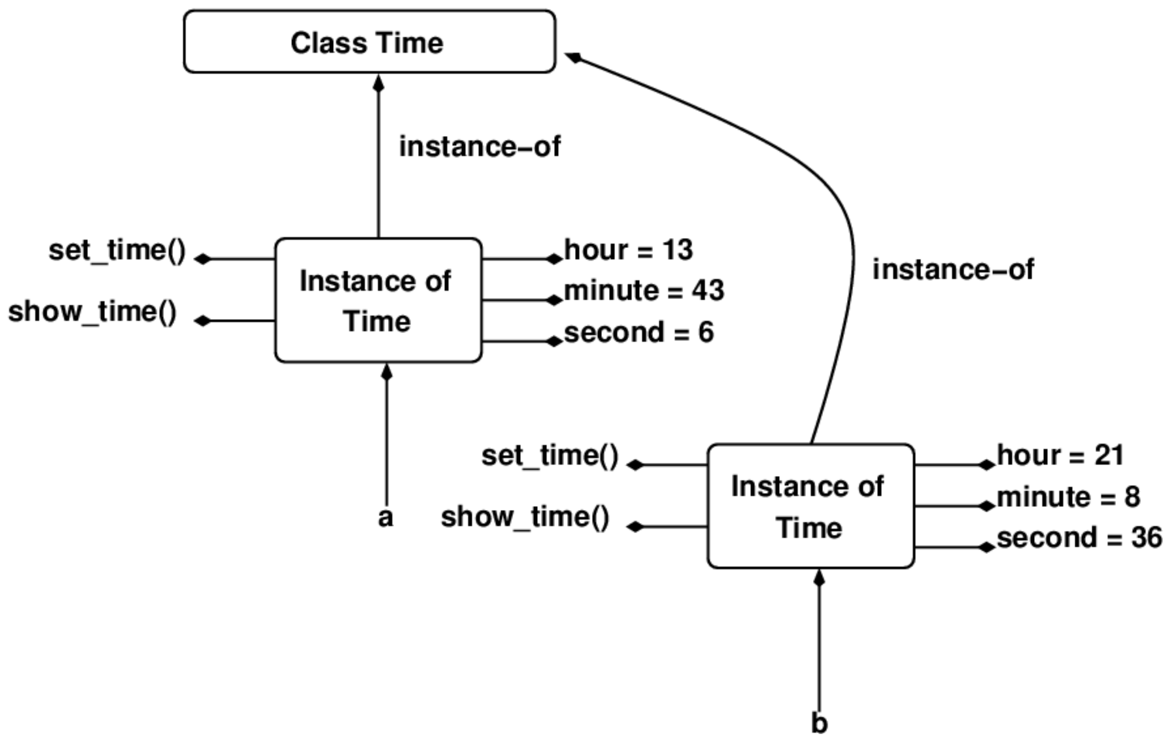
```
    print('The time is {:02d}:{:02d}:{:02d}'.format(
        time_object.hour, time_object.minute, time_object.second))
```

- This method specifies a single parameter i.e. the object's whose time we want to print.

```
a = Time()
Time.set_time(a, 13, 43, 6)
Time.show_time(a)

b = Time()
Time.set_time(b, 21, 8, 36)
Time.show_time(b)

c = Time()
Time.set_time(c, 3, 4, 7)
Time.show_time(c)
```

```
The time is 13:43:06
The time is 21:08:36
The time is 03:04:07
```

- We depict below the change to our objects where a new instance method called `show_time()` is attached to each.



29.7 Is there a handier way to call methods on an object?

- The way we are invoking methods on our `Time` objects requires we name both the class (e.g. `Time`) and the object (e.g. `a`).

```
Time.show_time(a)
```

```
The time is 13:43:06
```

- Remember how we earlier invoked methods on string (and other) objects?
- It seemed handier because it did not require us naming the `str` class and, interestingly, the string object `s` did not *appear* to be an argument for the method.

```
s = 'This is a string object, an instance of the string class'
print(s.count('a')) # Invoke the count method on object s
```

```
4
```

- It turns out, however, that `s.count('a')` is really just shorthand for `str.count(s, 'a')`.
- We can adopt the same approach with objects of our `Time` class i.e. rather than calling the `Time` class's methods by explicitly referencing the class name we can instead invoke our methods directly on an object of the `Time` class.

```
a = Time()
a.set_time(13, 43, 6) # Time.set_time(a, 13, 43, 6)
a.show_time()         # Time.show_time(a)

b = Time()
b.set_time(21, 8, 36) # Time.set_time(b, 21, 8, 36)
b.show_time()         # Time.show_time(b)

c = Time()
c.set_time(3, 4, 7)   # Time.set_time(c, 3, 4, 7)
c.show_time()         # Time.show_time(c)
```

```
The time is 13:43:06
The time is 21:08:36
The time is 03:04:07
```

- To understand how to write and call methods it is vital you appreciate that the code above and the corresponding comments are equivalent.
- Note when we call `a.show_time()` or `a.set_time(13, 43, 6)` the `show_time()` and `set_time()` methods still require the object `a` be passed as an argument.
- **Thus when we invoke an instance method on a Python object that object is automatically supplied as the first argument to the method**.
- This means that whenever we invoke an instance method on an object **we supply one fewer arguments than the number of parameters listed in the method definition** inside the class.
- We do so because Python automatically supplies the missing object argument on our behalf.
- What is *an instance method*? An instance method is one that acts upon a particular instance of an object. It is a method whose first parameter is the object operated upon.

- In our `Time` class both methods `set_time()` and `show_time()` are instance methods and the first parameter of each is a `time_object`.
- A class's methods are by default instance methods whose first parameter will always be the instance on which the method has been invoked.
- By convention the first parameter of instance methods is named `self`. Thus our `Time` class should really be implemented as follows.

```
class Time(object):

    def set_time(self, hour, minute, second):
        self.hour = hour
        self.minute = minute
        self.second = second

    def show_time(self):
        print('The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second))

a = Time()
a.set_time(13, 43, 6)
a.show_time()
```

```
The time is 13:43:06
```

- We will from now on write and invoke our instance methods as outlined in this section.

CHAPTER

THIRTY

LAB 7.1 (DEADLINE MONDAY 7 MARCH 23:59)

- Upload your code to Einstein to have it verified.

30.1 Element

- In `element_071.py` define an `Element` class to model a chemical element.
- An element has four data attributes: `number`, `name`, `symbol`, and `boiling point`.
- The `Element` class defines the following instance methods:
  - `set_attributes()` : sets the instance's attributes to the specified values
  - `print_attributes()` : prints the instance's attributes

- When your class is correctly implemented, running the following program should produce the given output.

```
from element_071 import Element

def main():

    e1 = Element()
    e2 = Element()
    e3 = Element()
    e4 = Element()
    e5 = Element()

    e1.set_attributes(1, 'Hydrogen', 'H', 20.3)
    e1.print_attributes()
```

(continues on next page)

(continued from previous page)

```
    assert(e1.number == 1)
    assert(e1.name == 'Hydrogen')
    assert(e1.symbol == 'H')
    assert(e1.bp == 20.3)

    e2.set_attributes(3, 'Lithium', 'Li', 1615)
    e2.print_attributes()

    e3.set_attributes(11, 'Sodium', 'Na', 1156)
    e3.print_attributes()

    e4.set_attributes(12, 'Magnesium', 'Mg', 1380)
    e4.print_attributes()

    e5.set_attributes(79, 'Gold', 'Au', 3129)
    e5.print_attributes()


if __name__ == '__main__':
    main()
```

```
Number: 1
Name: Hydrogen
Symbol: H
Boiling point: 20.3 K
Number: 3
Name: Lithium
Symbol: Li
Boiling point: 1615 K
Number: 11
Name: Sodium
Symbol: Na
Boiling point: 1156 K
Number: 12
Name: Magnesium
Symbol: Mg
Boiling point: 1380 K
Number: 79
Name: Gold
Symbol: Au
Boiling point: 3129 K
```

30.2 Bank account

- In `bank_071.py` define a `BankAccount` class to model a bank account.
- An bank account has three data attributes: `name`, `number`, and `balance`.
- The `BankAccount` class defines the following instance methods:
  - `set_attributes()` : sets the instance's attributes to the specified values
  - `print_attributes()` : prints the instance's attributes
  - `deposit()` : increases the balance by a given amount
- Once your class is correctly implemented, running the following program should produce the given output.

```
from bank_071 import BankAccount

def main():

    b1 = BankAccount()
    b1.set_attributes('Jim', 12343111, 300)

    assert(b1.name == 'Jim')
    assert(b1.number == 12343111)
    assert(b1.balance == 300)

    b1.print_attributes()
    b1.deposit(100)
    b1.print_attributes()

    assert(b1.balance == 400)


if __name__ == '__main__':
    main()
```

```
Name: Jim
Account number: 12343111
Balance: 300.00
Name: Jim
Account number: 12343111
Balance: 400.00
```

30.3 Point

- In `point_071.py` define a `Point` class to model a point in a two dimensional space.
- A point has two data attributes: `x` and `y`.
- The `Point` class defines the following instance methods:
  - `set_attributes()` : sets the instance's attributes to the specified values
  - `print_attributes()` : prints the instance's attributes
  - `reflect()` : reflects a point's coordinates through the origin (the effect is to negate the point's `x` and `y` coordinates)

- When your class is correctly implemented, running the following program should produce the given output.

```
from point_071 import Point

def main():
    p1 = Point()
    p2 = Point()

    p1.set_attributes(5, 5)
    p2.set_attributes(4.2, 3.8)

    p1.print_attributes()
    p2.print_attributes()

    assert(p1.x == 5)
    assert(p1.y == 5)

    p1.reflect()
    p1.print_attributes()

    assert(p1.x == -5)
    assert(p1.y == -5)

if __name__ == '__main__':
    main()
```

```
x: 5.00
y: 5.00
```

(continues on next page)

(continued from previous page)

```
x: 4.20
y: 3.80
x: -5.00
y: -5.00
```

30.4 Student

- In `student_071.py` define a `Student` class to model a student.
- A student has three data attributes: `sid` (student ID), `name` and `modlist` (the list of modules for which the student is registered).
- The `Student` class defines the following instance methods:
  - `set_attributes()` : sets the instance's attributes to the specified values (see the example below)
  - `print_attributes()` : prints the instance's attributes (see the example below)
  - `add_module()` : adds a module (passed as an argument) to `modlist` (has no effect if `modlist` already contains the module)
  - `del_module()` : removes a module (passed as an argument) from `modlist` (has no effect if the module is not in `modlist`)
- When your class is correctly implemented, running the following program should produce the given output.

```python
from student_071 import Student

def main():

    s1 = Student()

    s1.set_attributes(15234654, 'Jimmy Murphy', ['CA116'])
    s1.print_attributes()

    assert(s1.name == 'Jimmy Murphy')
    assert(s1.sid == 15234654)
    assert(s1.modlist == ['CA116'])
```

(continues on next page)

(continued from previous page)

```
    s1.add_module('CA117')
    s1.print_attributes()

    s1.add_module('CA100')
    s1.del_module('CA116')
    s1.print_attributes()

    assert(s1.modlist == ['CA117', 'CA100'])

if __name__ == '__main__':
    main()
```

```
ID: 15234654
Name: Jimmy Murphy
Modules: CA116
ID: 15234654
Name: Jimmy Murphy
Modules: CA116, CA117
ID: 15234654
Name: Jimmy Murphy
Modules: CA117, CA100
```

CHAPTER

THIRTYONE

LECTURE 7.2 : OBJECT-ORIENTED PROGRAMMING: SPECIAL METHODS

31.1 Introducing special methods: `__init__()`

- So far we have been instantiating our `Time` objects and initialising them in a two step process.

```
class Time(object):

    def set_time(self, hour, minute, second):
        self.hour = hour
        self.minute = minute
        self.second = second

    def show_time(self):
        print('The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second))

a = Time()
a.set_time(13, 43, 6)
```

- Can we create *and* automatically initialise an object in one step?
- We can if in our class we define a *special method* called `__init__()` (there are two underscores before and after `init`).
- If a class contains an `__init__()` method then that method is automatically called immediately an object of that class is created.
- We replace our old `set_time()` method with a suitable `__init__()` method giving the following `Time` class implementation.

```
class Time(object):

    def __init__(self, hour, minute, second):
        self.hour = hour
        self.minute = minute
        self.second = second

    def show_time(self):
        print('The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second))
```

- Now if we try to create a `Time` object as before, we get an error.

```
a = Time()
```

```
---------------------------------------------------------------------
↪--------
TypeError                                 Traceback (most recent␣
↪call last)
/tmp/ipykernel_6878/2028850231.py in <module>
----> 1 a = Time()

TypeError: __init__() missing 3 required positional arguments:
↪'hour', 'minute', and 'second'
```

- We get an error because `__init__()` will be automatically called upon object creation and it expects four arguments to be passed to it.
- One argument is automatically supplied (the object that becomes `self`) meaning we must supply three.
- How do we supply the arguments expected by `__init__()`?
- We supply them in the only place we can i.e. as arguments when creating a `Time` object as follows.

```
a = Time(13, 43, 6)
a.show_time()
```

```
The time is 13:43:06
```

- When we call the `Time` class now the following takes place:
  1. An empty instance of the `Time` class is created,
  2. this empty object is passed along with `13`, `43` and `6` to the `__init__()` method,
  3. the `__init__()` method initialises the object with the supplied arguments,
  4. a reference to the new and now initialised object is returned and assigned to `a` by the caller.

- Note that `__init__()` is a special method.
- The fact that it is special is indicated by the double underscore prefix and suffix.
- Special methods are not normally called directly.
- Thus *under normal circumstances* we will not call `__init__()` directly.
- From now on any classes we write will typically contain an `__init__()` method.
- A suitable `__init__()` method is one of the first things we should start thinking about when writing a new class.

31.2 Default `__init__()` parameter values

- Note an `__init__()` method is just like any other function in that it supports *default parameter values* for unsupplied arguments.
- This is very handy.
- It means we can initialise a new object to some default state when the creator does not supply any arguments during object instantiation.
- Thus our *final* `__init__()` method looks like this:

```
class Time(object):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def show_time(self):
        print('The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second))
```

- Now we can instantiate our `Time` objects with zero, one, two or three arguments.
- Any missing arguments will take on the default values specified in the `__init__()` method.

```
a = Time()
a.show_time()

a = Time(16)
a.show_time()

a = Time(16, 30)
a.show_time()

a = Time(16, 30, 59)
a.show_time()
```

```
The time is 00:00:00
The time is 16:00:00
The time is 16:30:00
The time is 16:30:59
```

31.3 Another special method : `__str__()`

- Another special method that we typically implement is `__str__()`.
- Whenever Python sees `print(class_instance)` it checks whether the class in question defines a method named `__str__()`.
- If it does the method is invoked (and passed a copy of the object as usual in `self`).
- What is printed is the string *returned* by the `__str__()` method.

- We can replace our `show_time()` method with this special method to make printing times handier and more intuitive.
- Below find the updated class and a demonstration of the method in action.

```
class Time(object):

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return 'The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second)


a = Time()
print(a)

a = Time(16)
print(a)

a = Time(16, 30)
print(a)

a = Time(16, 30, 59)
print(a)
```

```
The time is 00:00:00
The time is 16:00:00
The time is 16:30:00
The time is 16:30:59
```

CHAPTER THIRTYTWO

LAB 7.2 (DEADLINE MONDAY 7 MARCH 23:59)

- Upload your code to Einstein to have it verified.

32.1 Lamp

- In `lamp_072.py` define a `Lamp` class to model a lamp.
- A lamp has a single boolean data attribute: `on`. `on` can be either `True` or `False`.
- The `Lamp` class defines the following instance methods
  - `__init__()` : initialises the lamp (defaults to off)
  - `turn_on()` : turns the lamp on (has no effect if already on)
  - `turn_off()` : turns the lamp off (has no effect if already off)
  - `toggle()` : turns the lamp on if currently off and off if currently on
- When your class is correctly implemented, running the following program should produce no output.

```
from lamp_072 import Lamp

def main():
    lamp1 = Lamp()

    assert(not(lamp1.on))
    lamp1.turn_off()
    assert(not(lamp1.on))
    lamp1.turn_on()
```

(continues on next page)

(continued from previous page)

```
    assert(lamp1.on)
    lamp1.turn_on()
    assert(lamp1.on)
    lamp1.turn_off()
    assert(not(lamp1.on))
    lamp1.toggle()
    assert(lamp1.on)
    lamp1.turn_off()
    lamp1.turn_off()
    assert(not(lamp1.on))

    lamp2 = Lamp(True)

    assert(lamp2.on)
    lamp2.toggle()
    assert(not(lamp2.on))

if __name__ == '__main__':
    main()
```

32.2 Bank account

- In `bank_072.py` define a `BankAccount` class to model a bank account.
- A bank account has a single data attribute: `balance` which can be zero or any *positive* floating point value.
- The `BankAccount` class defines the following instance methods:
  - `__init__()` : initialises the bank account (balance defaults to zero)
  - `deposit()` : adds an amount to the balance
  - `withdraw()` : subtracts an amount from the balance if sufficient funds available
  - `__str__()` : returns the current balance as a string

- Once your class is correctly implemented, running the following program should produce the given output.

```python
from bank_072 import BankAccount

def main():
    b1 = BankAccount()

    assert(b1.balance == 0)
    b1.deposit(100)
    b1.deposit(50)
    assert(b1.balance == 150)
    b1.withdraw(200)
    assert(b1.balance == 150)
    b1.withdraw(150)
    assert(b1.balance == 0)
    print(b1)

    b2 = BankAccount(30)

    assert(b2.balance == 30)
    b2.withdraw(0.01)
    print(b2)

if __name__ == '__main__':
    main()
```

```
Your current balance is 0.00 euro
Your current balance is 29.99 euro
```

32.3 Point

- In `point_072.py` define a `Point` class to model a point in a two dimensional space.
- A point has two data attributes: `x` and `y`.
- The `Point` class defines the following instance methods:
  - `__init__()` : initialises the point (coordinates default to zero)
  - `distance()` : returns the distance between two points
  - `__str__()` : returns the point as a string
- When your class is correctly implemented, running the following program should produce

the given output.

```
from point_072 import Point

def main():
    p1 = Point()

    assert(p1.x == 0)
    assert(p1.y == 0)
    print(p1)

    p2 = Point(3, 4)

    assert(p2.x == 3)
    assert(p2.y == 4)
    print(p2)

    print('{:.1f}'.format(p1.distance(p2)))

if __name__ == '__main__':
    main()
```

```
(0.0, 0.0)
(3.0, 4.0)
5.0
```

32.4 Student

- In `student_072.py` define a `Student` class to model a student.
- A student has three data attributes: `sid` (student ID), `name` and `modlist` (the list of modules for which the student is registered).
- The `Student` class defines the following instance methods:
  - `__init__()` : initialises the student (module list default to empty)
  - `add_module()` : adds a module (passed as an argument) to `modlist` (has no effect if `modlist` already contains the module)
  - `del_module()` : removes a module (passed as an argument) from `modlist` (has no effect if the module is not in `modlist`)
  - `__str__()` : returns a string representation of the student

- When your class is correctly implemented, running the following program should produce the given output.

```
from student_072 import Student

def main():

    s1 = Student(15234654, 'Jimmy Murphy')

    assert(s1.name == 'Jimmy Murphy')
    assert(s1.sid == 15234654)
    assert(s1.modlist == [])
    s1.add_module('CA116')
    s1.add_module('CA117')
    print(s1)

    s2 = Student(17234654, 'Harry Byrne', ['CA177', 'CA101'])
    assert(s2.modlist == ['CA177', 'CA101'])
    print(s2)

    s3 = Student(19343112, 'Mindy Malone')
    print(s3)

if __name__ == '__main__':
    main()
```

```
ID: 15234654
Name: Jimmy Murphy
Modules: CA116, CA117
ID: 17234654
Name: Harry Byrne
Modules: CA177, CA101
ID: 19343112
Name: Mindy Malone
Modules:
```

CHAPTER

THIRTYTHREE

LECTURE 7.3 : OBJECT-ORIENTED PROGRAMMING: INSTANCE METHODS

33.1 Adding an instance method

- Let's add another method to our `Time` class.
- The new method is called `is_later_than()` and returns `True` if one time is later than another.
- How will we go about writing this method?
- Let's start with its signature i.e. how many parameters will the method need to specify?
- Since we are comparing two times it seems clear that the method will require two times be passed to it, `t1` and `t2`, both of which will be objects of the `Time` class.
- Next, what will our method return?
- It also seems sensible that our method should return a boolean, `True` if `t1` is later than `t2` and `False` otherwise.
- How will our method be invoked?
- Well, our method operates directly on an *instance* of the `Time` class so it will be an *instance method*.
- It compares one instance of the class with another instance of the same class.
- Thus we will invoke it like this: `t1.is_later_than(t2)` in order to ask "Is `t1` later than `t2`?"
- Here is our updated `Time` class.

```
class Time(object):

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
```

(continues on next page)

(continued from previous page)

```
        self.minute = minute
        self.second = second

    def is_later_than(self, other):
        # Compare hours
        if self.hour > other.hour:
            return True
        if self.hour < other.hour:
            return False

        # Hours are equal so compare minutes
        if self.minute > other.minute:
            return True
        if self.minute < other.minute:
            return False

        # Hours and minutes are equal so compare seconds
        if self.second > other.second:
            return True

        return False

    def __str__(self):
        return 'The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second)

t1 = Time(13, 43, 6)
t2 = Time(14, 52, 7)
t3 = Time(14, 43, 7)
t4 = Time(13, 43, 7)

print(t2.is_later_than(t1))
print(t1.is_later_than(t2))
print(t3.is_later_than(t2))
print(t4.is_later_than(t1))
```

```
True
False
False
True
```

- When we call `t1.is_later_than(t2)` the `t1` argument becomes the `self` parameter in the method while the `t2` argument becomes the `other` parameter.

- (Remember `t1.is_later_than(t2)` is really just shorthand for `Time.is_later_than(t1, t2)` and in the latter it is obvious that `t1` becomes `self` and `t2` becomes `other` inside the method.)
- Study the method to ensure you understand how it works.
- It begins by comparing hours, then minutes and finally seconds.
- Note how it returns `True` or `False` immediately it has a decision.
- *We can return from a method any time*. We do not have to wait until the end of its code has been reached. (This technique can help keep your code succinct.)

33.2 Adding another instance method

- Looking again at our `is_later_than()` method, it could require making many comparisons (through `if` statements) before coming to a conclusion.
- The problem is we have potentially many attributes to compare (`hour`, `minute` and `second` from each of `self` and `other`).
- It might help if we could could convert all of the attributes of `self` and `other` into a single number and compare them instead.
- Any ideas on how to proceed?
- Well, if we convert each `Time` instance's attributes to a total number of seconds since midnight (`00:00:00`) then comparing two times can be done simply with >, <, ==, etc.
- So we need to add another method to our class called `time_to_seconds()` that specifies a single `Time` object parameter and returns a single number representing the corresponding number of seconds since midnight.
- The method will be another of our `Time` class's instance methods.
- We will call this helper method from our updated `is_later_than()` method. Putting everything together we get the following.

```
class Time(object):

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def time_to_seconds(self):
        return self.hour*60*60 + self.minute*60 + self.second
```

(continues on next page)

(continued from previous page)

```
    def is_later_than(self, other):
        return self.time_to_seconds() > other.time_to_seconds()

    def show_time(self):
        print('The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second))

t1 = Time(13, 43, 6)
t2 = Time(14, 52, 7)
t3 = Time(14, 43, 7)
t4 = Time(13, 43, 7)

print(t2.is_later_than(t1))
print(t1.is_later_than(t2))
print(t3.is_later_than(t2))
print(t4.is_later_than(t1))
```

```
True
False
False
True
```

33.3 Adding another instance method

- Let's extend our `Time` class with a more complex instance method. This one will take two `Time` objects and add them to produce and return a *new* `Time` object.
- We want the new `Time` object to be a valid time in the 24-hour format.
- Again, when writing a new method we start with its signature i.e. how will we invoke the method?
- It seems it ought to work as follows: `t3 = t1.plus(t2)` where `t3` is the result of adding time `t2` to `t1`.
- The most straightforward approach to coding our new `plus()` method would seem to be to take the two `Time` instances passed to it and firstly convert each to an equivalent number of seconds.
- We can then add the seconds in each to produce a total number of seconds.
- Finally we need to convert this total number of seconds back into a valid `Time` object to be returned to the caller.
- To to the conversion we will have to add another helper method `seconds_to_time()`.

- Where will we put the helper method `seconds_to_time()`? This is an interesting question. Is it an instance method?
- If it were an instance method we would add it to the class definition as we have done with all of our methods so far.
- It is *not* an instance method however.
- How do we know it is not an instance method?
- *Because it is a method that it makes sense to call in the absence of an instance of the Time class*.
- In other words we should not be required to have an instance of `Time` in order to invoke the method `seconds_to_time()`.
- All we should require is a number of seconds from which we want the method to derive an instance of the class `Time`.
- Given it is not an instance method, for now we will simply add `seconds_to_time()` as a *function* to the *module* containing the definition of our `Time` class.
- The `seconds_to_time()` function makes use of `divmod()` to help us avoid generating `Times` such as `26:78:91`
- What does `divmod()` do? Well `minute, second = divmod(s, 60)` divides `s` by `60` and puts the resulting whole number of minutes in `minute` with any remainder going in `second`.
- So `1, 20 == divmod(80, 60)` or "80 seconds is equal to 1 minute 20 seconds".
- We apply similar logic to working out the final number of minutes and hours in our new `Time` object.
- Our updated `Time` class looks like this.

```
class Time(object):

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def time_to_seconds(self):
        return self.hour*60*60 + self.minute*60 + self.second

    def is_later_than(self, other):
        return self.time_to_seconds() > other.time_to_seconds()

    def plus(self, other):
        return seconds_to_time(self.time_to_seconds() +
```

(continues on next page)

(continued from previous page)

```
                                        other.time_to_seconds())

    def __str__(self):
        return 'The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second)

def seconds_to_time(s):
    minute, second = divmod(s, 60)
    hour, minute = divmod(minute, 60)
    overflow, hour = divmod(hour, 24)
    return Time(hour, minute, second)
```

- Let's see our new method in action.

```
t1 = Time(13, 58, 23)
t2 = Time(0, 10, 0)
t3 = t1.plus(t2)
print(t3)
t4 = Time(16, 18, 36)
t5 = Time(12, 10, 19)
t6 = t4.plus(t5)
print(t6)
```

```
The time is 14:08:23
The time is 04:28:55
```

- Note we do *not* want `t3` to be `13:68:23` as that would not be a a valid time in the 24-hour format. For similar reasons `t6` should not be `28:28:55`.
- So we have correctly handled wraparound in our new method thanks to `divmod()`.

CHAPTER

THIRTYFOUR

LECTURE 8.1 : OBJECT-ORIENTED PROGRAMMING: MORE INSTANCE METHODS

34.1 Adding yet another instance method

- Let's write an instance method that modifies the `Time` instance it is invoked on.
- Our new method increments a time by adding to it another time (it does not return anything).
- If we print the `Time` object before and after invoking the method on it we should find that it differs by the amount of time specified in the second parameter.
- Here is our first attempt at writing such a method:

```
class Time(object):

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def time_to_seconds(self):
        return self.hour*60*60 + self.minute*60 + self.second

    def is_later_than(self, other):
        return self.time_to_seconds() > other.time_to_seconds()

    def plus(self, other):
        return seconds_to_time(
            self.time_to_seconds() + other.time_to_seconds())

    def increment(self, other):
        z = self.plus(other)
        self = z

```

(continues on next page)

(continued from previous page)

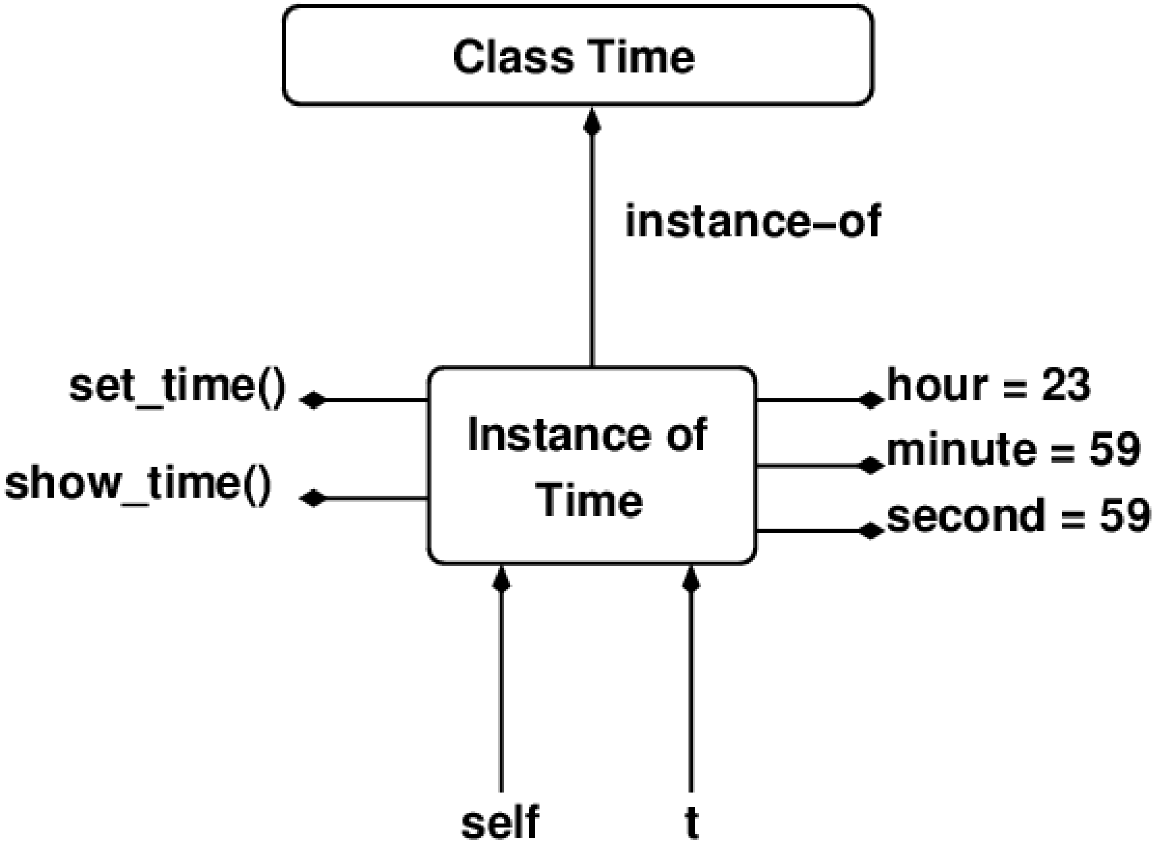
```
    def __str__(self):
        return 'The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second)

def seconds_to_time(s):
    minute, second = divmod(s, 60)
    hour, minute = divmod(minute, 60)
    overflow, hour = divmod(hour, 24)
    return Time(hour, minute, second)
```

- We can see what this new method is trying to do: We pass to it a `Time` to be incremented in `self` and in `other` we pass by how much we want `self` to be incremented.
- The method adds the two times together (by calling the instance method `plus()` which handles any wraparound issues) to produce a new `Time` object `t`.
- Finally we *overwrite* `self` with a reference to this new `Time` object `t`. Will this new method work? Well let's try it and see.

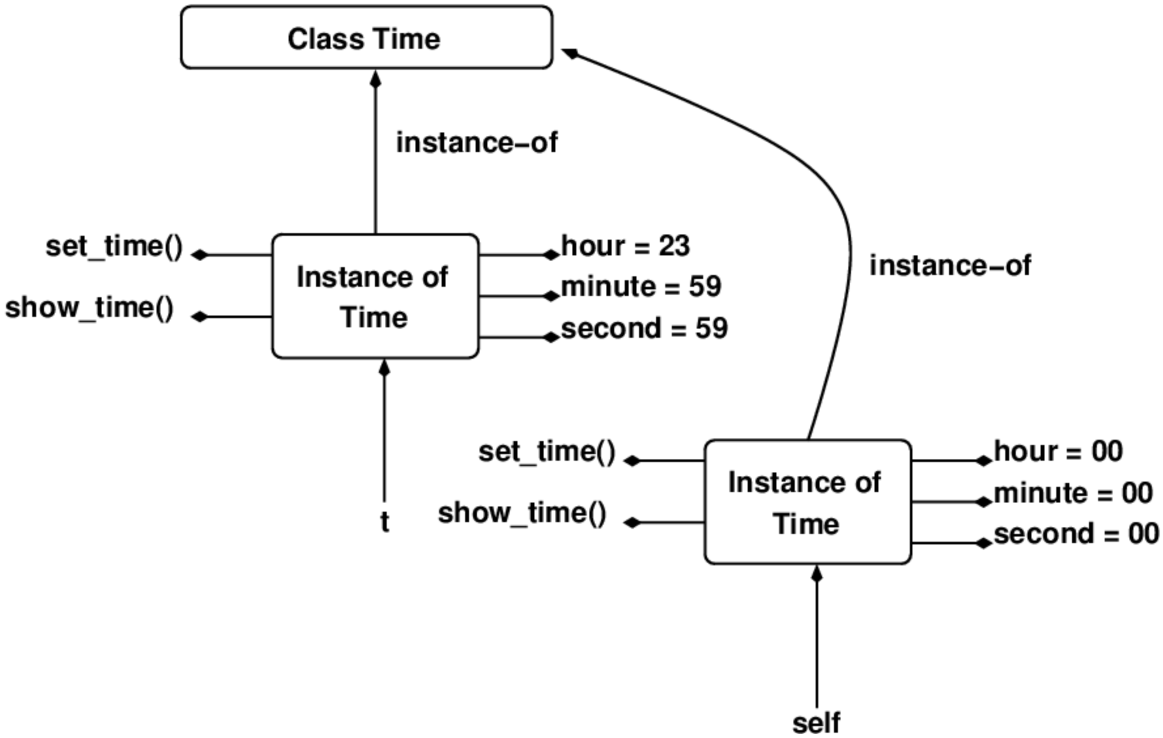
```
t = Time(23, 59, 59)
i = Time(0, 0, 1)
t.increment(i)
print(t)
```

```
The time is 23:59:59
```

- Well that's disappointing! What is going on? Why is `t` unchanged after invoking the `increment()` method? `t` should now be `00:00:00` but our method has had no effect on it.
- The following diagram represents the situation on entering the `increment()` method:



- This diagram represents the situation on leaving the `increment()` method:



- When `increment()` is invoked, `self` becomes a copy of `t`.
- Thus `self` and `t` both reference the same object.
- When the method executes `self = z`, however, `self` is overwritten to point to a new `Time` object `z`.
- Note however that `t` *still points to the original object* and this object remains unchanged. Thus when we print it we get back the original time.
- To update the `t` object via the `increment()` method we must write *through* `self` in order to update the object that both `t` and `self` point to.
- What we cannot do is *overwrite* `self` because doing so will cause a new object to be created (one that is unrelated to `t`).
- Below we write *through* `self` to update its attributes and in so doing we update the attributes of `t` (since `self` and `t` are aliases for the same object):

```
class Time(object):

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
```

(continues on next page)

(continued from previous page)

```
        self.second = second

    def time_to_seconds(self):
        return self.hour*60*60 + self.minute*60 + self.second

    def is_later_than(self, other):
        return self.time_to_seconds() > other.time_to_seconds()

    def plus(self, other):
        return seconds_to_time(
            self.time_to_seconds() + other.time_to_seconds())

    def increment(self, other):
        z = self.plus(other)
        self.hour, self.minute, self.second = z.hour, z.minute, z.
↪second

    def __str__(self):
        return 'The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second)

def seconds_to_time(s):
    minute, second = divmod(s, 60)
    hour, minute = divmod(minute, 60)
    overflow, hour = divmod(hour, 24)
    return Time(hour, minute, second)
```

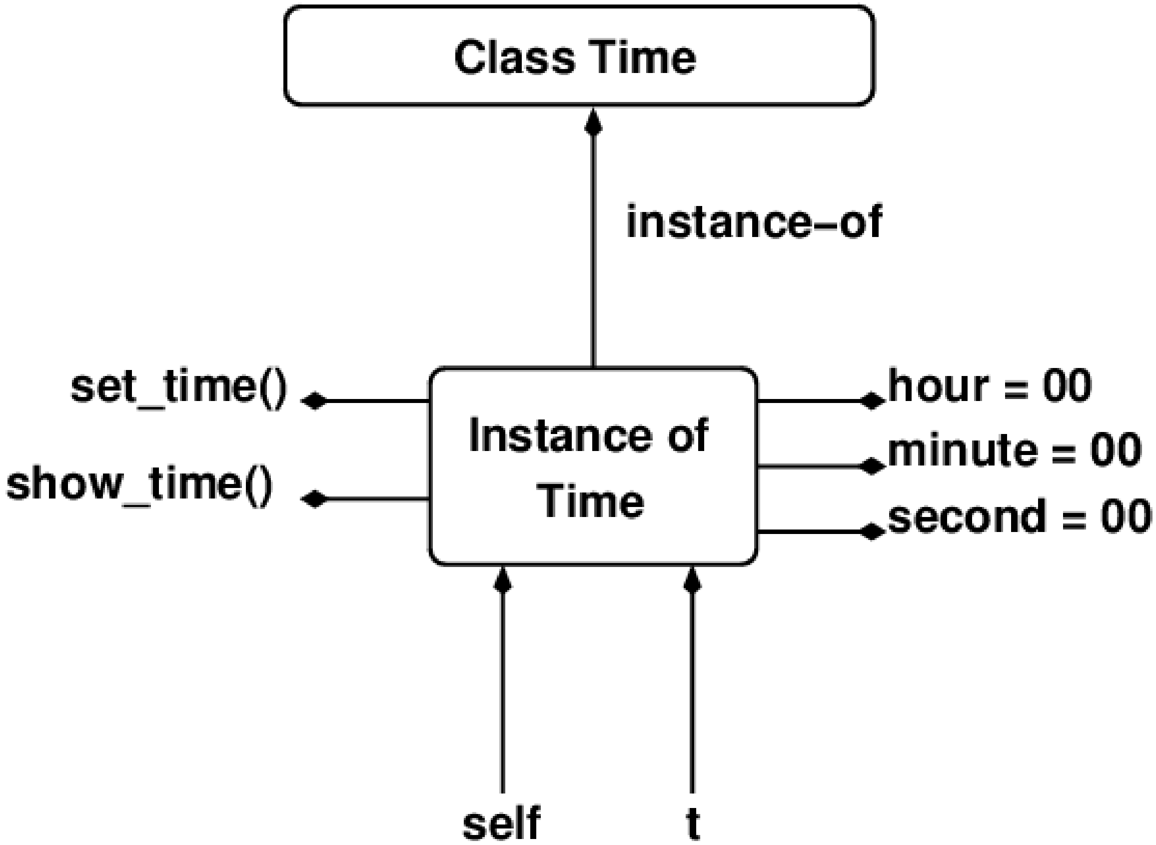
- Let's verify this version works as intended.

```
t = Time(23, 59, 59)
i = Time(0, 0, 1)
t.increment(i)
print(t)
```

```
The time is 00:00:00
```

- That's more like it!

- We can represent this version with the following diagram:



CHAPTER

THIRTYFIVE

LECTURE 8.2 : OBJECT-ORIENTED PROGRAMMING: MORE SPECIAL METHODS

35.1 Testing objects for equality with ==

- Here is our current `Time` class definition.

```
class Time(object):

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def time_to_seconds(self):
        return self.hour*60*60 + self.minute*60 + self.
↪second

    def is_later_than(self, other):
        return self.time_to_seconds() > other.time_to_
↪seconds()

    def plus(self, other):
        return seconds_to_time(
            self.time_to_seconds() + other.time_to_
↪seconds())

    def increment(self, other):
        z = self.plus(other)
        self.hour, self.minute, self.second = z.hour, z.
↪minute, z.second

    def __str__(self):
        return 'The time is {:02d}:{:02d}:{:02d}'.format(
```

(continues on next page)

(continued from previous page)

```
                self.hour, self.minute, self.second)

def seconds_to_time(s):
    minute, second = divmod(s, 60)
    hour, minute = divmod(minute, 60)
    overflow, hour = divmod(hour, 24)
    return Time(hour, minute, second)
```

- Look at the following demonstration of some rather surprising behaviour.

```
t1 = Time(13,30,00)
t2 = Time(13,30,00)
print(t1 == t2)
print(t1 is t2)
```

```
False
False
```

- When dealing with *user-defined classes*, such as the `Time` class above, the == operator tests whether two references are equal i.e. whether two references point to the same object.
- This means that for user-defined types the *default* behaviour of the == operator is identical to that of the `is` operator.
- Can we fix it so that when we write `t1 == t2` as above to compare two objects of the `Time` class, that instead of the default behaviour which compares two references for equality, we compare the *contents* of the two objects for equality?
- In other words, can we *override* the default behaviour of the == operator such that when we compare two `Time` objects with == it is a *user-defined method* of the `Time` class that is invoked?
- The answer (you probably guessed it already) is yes!
- After overriding the behaviour of the == operator its behaviour depends on the objects it compares: if we compare two `Time` objects one method runs but if we compare two objects of a different type e.g. `Dates` then a different method runs.

- This is *operator overloading* where an operator has numerous semantics depending on its operands.
- Operator overloading is a form of *polymorphism* since the behaviour of an operator depends on its operands.

35.2 Operator overloading

- If a class defines an `__eq__()` method then that method is invoked when we use the == operator to compare two instances of that class.
- The `__eq__()` method is a *special method* (like `__init__()`) in that it is not normally called directly (a fact hinted at by the double underscore prefix and suffix).
- If we add such a method to our `Time` class we get the following.

```
class Time(object):

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __eq__(self, other):
        return ((self.hour, self.minute, self.second) == (
            other.hour, other.minute, other.second))

    def time_to_seconds(self):
        return self.hour*60*60 + self.minute*60 + self.second

    def is_later_than(self, other):
        return self.time_to_seconds() > other.time_to_seconds()

    def plus(self, other):
        return seconds_to_time(
            self.time_to_seconds() + other.time_to_seconds())

    def increment(self, other):
        z = self.plus(other)
        self.hour, self.minute, self.second = z.hour, z.minute, z.
↪second

    def __str__(self):
        return 'The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second)
```

(continues on next page)

(continued from previous page)

```
def seconds_to_time(s):
    minute, second = divmod(s, 60)
    hour, minute = divmod(minute, 60)
    overflow, hour = divmod(hour, 24)
    return Time(hour, minute, second)
```

- Now when we compare two `Time` objects with the == operator we observe the desired behaviour.

```
t1 = Time(13,30,00)
t2 = Time(13,30,00)
print(t1 == t2) # Invokes Time.__eq__(t1, t2)
print(t1 is t2)
```

```
True
False
```

- Hmm. This is interesting.
- We have just seen how operator overloading can be used to overload the == operator.
- Can we implement other special methods so that when we use operators like +, −, +=, −=, >, >=, <, <=, etc. with our objects that it is these methods that are invoked?
- If it were possible then we could replace our `plus()`, `is_later_than()` and `increment()` methods with the more intuitive +, > and += operators.
- It turns out that, as usual, the answer is yes!
- There is a large collection of special methods which when implemented will overload (i.e. add special meaning to) every operator you can think of.
- For example:
  - Method `__add__()` overloads + (handles `t1 + t2`)
  - Method `__iadd__()` overloads += (handles `t1 += t2`)
  - Method `__sub__()` overloads − (handles `t1 - t2`)
  - Method `__isub__()` overloads −= (handles `t1 -= t2`)

- Method `__mul__()` overloads * (handles `t * 2`)
- Method `__imul__()` overloads *= (handles `t *= 2`)
- Method `__rmul__()` overloads * (handles `2 * t`)
- Method `__gt__()` overloads > (handles `t1 > t2`)
- Method `__ge__()` overloads >= (handles `t1 >= t2`)
- Method `__lt__()` overloads < (handles `t1 < t2`)
- Method `__le__()` overloads <= (handles `t1 <= t2`)

- What is the difference between `__add__()` and `__iadd__()`?
- Well, they each specify two parameters, `self` and `other`.
- `__add__()` adds `self` and `other` to produce a *new object* and returns a reference to that object to the caller.
- Methods that overload *in-place* operators however, like `__iadd__()`, should avoid returning a new object.
- They instead modify `self` in-place (in the `__iadd__()` case this involves reaching *inside* `self` to update its contents) and return a reference to it. (See the implementations of `__add__()` and `__iadd__()` below.)
- Also of interest are methods such as `__rmul__()`.
- When Python sees an expression such as `t * 2` (where `t` is an instance of `Time`) it checks the left hand object for a `__mul__()` method.
- Provided we have implemented one it is invoked where `self` is a reference to `t` and `other` is a reference to `2`.
- What if Python sees an expression such as `2 * t`? Again it invokes the left hand object's `__mul__()` method.
- But the `__mul__()` method of `2` (an integer) does not know how to work with `Time` objects so we are in trouble.
- But Python does not give up! (Neither should you.)
- It checks whether the right hand object implements an `__rmul__()` method.
- If it does it is invoked where, again, `self` is a reference to `t` and `other` is a reference to `2`.
- Special methods are documented here: https://docs.python.org/3/reference/datamodel.html#special-method-names.

```
class Time(object):

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __eq__(self, other):
        return ((self.hour, self.minute, self.second) == (
            other.hour, other.minute, other.second))

    def __add__(self, other):
        return seconds_to_time(
            self.time_to_seconds() + other.time_to_seconds())

    def __gt__(self, other):
        return self.time_to_seconds() > other.time_to_seconds()

    def __iadd__(self, other):
        z = self + other
        self.hour, self.minute, self.second = z.hour, z.minute, z.
↪second
        return self

    def time_to_seconds(self):
        return self.hour*60*60 + self.minute*60 + self.second

    def __str__(self):
        return 'The time is {:02d}:{:02d}:{:02d}'.format(
            self.hour, self.minute, self.second)

def seconds_to_time(s):
    minute, second = divmod(s, 60)
    hour, minute = divmod(minute, 60)
    overflow, hour = divmod(hour, 24)
    return Time(hour, minute, second)
```

- Here is what we can now do with our `Time` objects thanks to operator overloading.

```
t1 = Time(12,0,0)
t2 = Time(0,0,1)
print(t1 == t2) # Invokes Time.__eq__(t1, t2)
```

(continues on next page)

(continued from previous page)

```
print(t1 != t2)
print(t1 > t2) # Invokes Time.__gt__(t1, t2)
print(t1 < t2) # Invokes Time.__gt__(t2, t1)
print(t2 > t1) # Invokes Time.__gt__(t2, t1)
print(t2 < t1) # Invokes Time.__gt__(t1, t2)
t3 = t1 + t2 # Invokes Time.__add__(t1, t2)
print(t3)
print(t2)
print(t1)
t1 += t2 # Invokes Time.__iadd__(t1, t2)
print(t1)
print(t2)
```

```
False
True
True
False
False
True
The time is 12:00:01
The time is 00:00:01
The time is 12:00:00
The time is 12:00:01
The time is 00:00:01
```

35.3 Everything in Python is an object

- Everything in Python is an object.
- When we ask Python to evaluate `3 + 4` it is easy to forget we are working with objects.
- The following illustrates that even in this simple example we are invoking methods on integer objects.

```
print(3 + 4)
print((3).__add__(4)) # equivalent to the above
```

```
7
7
```

CHAPTER

THIRTYSIX

LAB 8.1 (DEADLINE MONDAY 14 MARCH 23:59)

- Upload your code to Einstein to have it verified.

36.1 Operator overloading: GAA score

- In `gaa_081.py` define a `Score` class to model a GAA score. A score consists of a number of goals and points.
- When your class is correctly implemented, running the following program should produce the given output.

```
from gaa_081 import Score

def main():

    s1 = Score()
    print(s1)

    s2 = Score(3, 12)
    assert(s2.goals == 3)
    assert(s2.points == 12)
    print(s2)

if __name__ == '__main__':
    main()
```

```
0 goal(s) and 0 point(s)
3 goal(s) and 12 point(s)
```

36.2 Operator overloading: GAA score

- In `gaa_comp_081.py` define a `Score` class to model a GAA score.
- Start with a copy of the code you used in `gaa_081.py`.
- Extend the class so it supports the comparison of GAA scores.
- When your class is correctly implemented, running the following program should produce no output.

```python
from gaa_comp_081 import Score

def main():

    s1 = Score()
    s2 = Score(3, 12)
    s3 = Score(4, 9)

    assert(s1 < s2)
    assert(s1 <= s2)
    assert(s2 > s1)
    assert(s2 >= s1)
    assert(s1 != s2)
    assert(s2 == s3)

if __name__ == '__main__':
    main()
```

36.3 Operator overloading: GAA score

- In `gaa_add_081.py` define a `Score` class to model a GAA score.
- Start with a copy of the code you used in `gaa_comp_081.py`.
- Extend the class so it supports the addition of GAA scores.
- When your class is correctly implemented, running the following program should produce the given output.

```python
from gaa_add_081 import Score

def main():

    s1 = Score()
    s2 = Score(3, 12)
```

(continues on next page)

(continued from previous page)

```
    s3 = Score(4, 9)
    s4 = Score(1, 1)

    s5 = s3 + s4
    print(s5)
    assert(isinstance(s5, Score))
    assert(s5 is not s3)
    assert(s5 is not s4)

    before = s2
    s2 += s4
    print(s2)
    assert(isinstance(s2, Score))
    assert(s2 is before)
    assert(s2 is not s4)

if __name__ == '__main__':
    main()
```

```
5 goal(s) and 10 point(s)
4 goal(s) and 13 point(s)
```

36.4 Point

- In `point_081.py` define a `Point` class to model a point in a two dimensional space.
- A point has two data attributes: `x` and `y`.
- The `Point` class defines the following instance methods:
  - `__init__()` : initialises the point (coordinates default to zero)
  - `midpoint()` : returns a new instance of `Point` that is the midpoint between this point and another point
  - `__str__()` : returns the point as a string

- When your class is correctly implemented, running the following program should produce the given output.

```
from point_081 import Point

def main():
    p1 = Point(2, 3)
    p2 = Point(4, 6)

    p3 = p1.midpoint(p2)

    print(p1)
    print(p2)
    print(p3)

    assert(isinstance(p3, Point))

if __name__ == '__main__':
    main()
```

```
(2.0, 3.0)
(4.0, 6.0)
(3.0, 4.5)
```

36.5 Circle

- In `circle_081.py` also define a `Circle` class to model a circle in a two dimensional space.
- A circle has two data attributes: `radius` (a floating point number) and `centre` (an instance of `Point` above).
- (Include in `circle_081.py` a copy of the `Point` class definition you developed in the previous question.)
- For any new `Circle` instance the radius defaults to one and its centre defaults to the `Point` (0,0).
- Define appropriate `__init__()` and `__str__()` methods.
- When your class is correctly implemented, running the following program should produce the given output.

```
from circle_081 import Point, Circle

def main():
    p1 = Point(2, 3)
    c1 = Circle(p1, 5)
```

(continues on next page)

(continued from previous page)

```
    assert(c1.centre is p1)
    assert(c1.radius == 5)

    c2 = Circle(p1)
    assert(c2.centre is p1)
    assert(c2.radius == 1)

    c3 = Circle()
    assert(isinstance(c3.centre, Point))
    assert(c3.radius == 1)

    c4 = Circle()
    assert(c3.centre is not c4.centre)

    print(c1)
    print(c2)
    print(c3)

if __name__ == '__main__':
    main()
```

```
Centre: (2.0, 3.0)
Radius: 5
Centre: (2.0, 3.0)
Radius: 1
Centre: (0.0, 0.0)
Radius: 1
```

36.6 Adding circles

- It is possible to add circles.
- Adding two circles, A and B, produces a new circle whose centre is the midpoint between the centres of A and B and whose radius is the sum of the radii of A and B.
- Add the required instance method to the `Circle` class to support this behaviour and call it `circle_add_081.py`. Include your `Point` class definition.
- Once your classes are correctly implemented, running the following program should produce the given output.

```
from circle_add_081 import Point, Circle
```

(continues on next page)

(continued from previous page)

```
def main():
    p1 = Point()
    p2 = Point(4, 6)

    c1 = Circle(p1, 10)
    c2 = Circle(p2, 5)

    c3 = c1 + c2
    assert(isinstance(c3, Circle))
    print(c3)

if __name__ == '__main__':
    main()
```

```
Centre: (2.0, 3.0)
Radius: 15
```

CHAPTER

THIRTYSEVEN

LAB 8.2 (DEADLINE MONDAY 14 MARCH 23:59)

- Upload your code to Einstein to have it verified.

37.1 Contact

- In `contact_082.py` define a `Contact` class to model a contact.
- A contact has three data attributes: `name`, `phone`, `email`.
- When your class is correctly implemented, running the following program should produce the given output:

```
from contact_082 import Contact

def main():

    c = Contact('Sue', '085-6442378', 'sue@eircom.net')

    assert(c.name == 'Sue')
    assert(c.phone == '085-6442378')
    assert(c.email == 'sue@eircom.net')

    print(c)

if __name__ == '__main__':
    main()
```

```
Sue 085-6442378 sue@eircom.net
```

37.2 Contact list

- In `contact_list_082.py` define a `ContactList` class to model a contact list.
- Include in `contact_list_082.py` a copy of your `Contact` class from the previous question.
- A contact list has a single data attribute: `d`.
- `d` is a dictionary that maps a name to the corresponding instance of the `Contact` class.
- The `ContactList` class defines the following instance methods:
  - `__init__()` initialises a new `ContactList`
  - `add()` : adds a new `Contact` to the contact list (or updates an existing `Contact` if already present)
  - `remove()` : removes a `Contact` from the contact list (no effect if no such contact exists)
  - `get()` : returns the `Contact` with the specified name (or returns `None` if not in the contact list)
  - `__str__()` : returns a string containing all contacts' details listed in increasing alphabetical name order (you might add each contact's details to a list and invoke join on that list in order to build the output of this method)
- Once your classes are correctly implemented, running the following program should produce the given output.

```python
from contact_list_082 import Contact, ContactList

def main():
    clist = ContactList()

    c1 = Contact('Sue', '085-6442378', 'sue@eircom.net')
    clist.add(c1)

    c2 = Contact('Jimmy', '086-1223277', 'james@apple.com')
    clist.add(c2)

    c3 = Contact('Wendy', '086-9112645', 'wendy@physics.dcu.ie')
    clist.add(c3)
```

(continues on next page)

(continued from previous page)

```
    c = clist.get('Wendy')
    assert(c is c3)

    clist.remove('Wendy')
    c = clist.get('Wendy')
    assert(c is None)

    c4 = Contact('Abbey', '087-7586344', 'abbey@gmail.com')
    clist.add(c4)

    print(clist)

if __name__ == '__main__':
    main()
```

```
Contact list
------------
Abbey 087-7586344 abbey@gmail.com
Jimmy 086-1223277 james@apple.com
Sue 085-6442378 sue@eircom.net
```

37.3 Meeting

- In `meeting_082.py` define a `Meeting` class to model a meeting.
- A meeting has three data attributes: a starting hour, a starting minute and a duration.
- When your class is correctly implemented, running the following program should produce the given output:

```
from meeting_082 import Meeting

def main():

    m = Meeting(9, 5, 30)

    assert(m.hour == 9)
    assert(m.minute == 5)
    assert(m.duration == 30)

    print(m)

if __name__ == '__main__':
    main()
```

```
09:05 (30 minutes)
```

37.4 Schedule

- In `schedule_082.py` define a `Schedule` class to model the day's schedule of meetings.
- Include in `schedule_082.py` a copy of your `Meeting` class from the previous question.
- The `Schedule` class defines the following instance methods:
  - `__init__()` initialises a new `Schedule`
  - `add()` : adds a new `Meeting` to the schedule
  - `__str__()` : returns a string containing all meeting details in increasing order of start time
- Once your classes are correctly implemented, running the following program should produce the given output.

```python
from schedule_082 import Meeting, Schedule

def main():
    schedule = Schedule()

    m = Meeting(13, 0, 15)
    schedule.add(m)

    m = Meeting(9, 5, 30)
    schedule.add(m)

    m = Meeting(16, 30, 5)
    schedule.add(m)

    print(schedule)

if __name__ == '__main__':
    main()
```

```
Schedule
--------
```

(continues on next page)

(continued from previous page)

```
09:05 (30 minutes)
13:00 (15 minutes)
16:30 (5 minutes)
Meetings today: 3
```

CHAPTER

THIRTYEIGHT

LECTURE 9.1 : OBJECT-ORIENTED PROGRAMMING: STACKS AND QUEUES

38.1 Stacks

- The *stack* is a fundamental data structure which stores a collection of objects (of arbitrary type) that are inserted and removed in a *last-in, first-out (LIFO)* order.
- Objects can always be added to the stack but the only object accessible at any time is the most recently added object (which lives at the *top* of the stack).
- The *push* operation is used to add an object to the stack (making it the new stack top) while the *pop* operation removes the object currently at the top of the stack.

38.2 Stack methods

- An instance `S` of the stack abstract data type supports at a minimum the following two methods:
  - `S.push(e)`: Add element `e` to the top of the stack `S`.
  - `S.pop()`: Remove and return the element at the top of the stack `S`; an error occurs if the stack `S` is currently empty.
- The following convenience methods are also often implemented:
  - `S.top()`: Return a reference to the top element of stack `S` without removing it; an error occurs if the stack `S` is currently empty.
  - `S.is_empty()`: Return `True` if stack `S` is empty and `False` otherwise.
  - `len(S)`: Return the number of elements in `S`.

38.3 Implementing a stack with a list

```
class Stack(object):

    def __init__(self):
        self.l = []

    def push(self, e):
        self.l.append(e)

    def pop(self):
        return self.l.pop()

    def top(self):
        return self.l[-1]

    def is_empty(self):
        return len(self.l) == 0

    def __len__(self):
        return len(self.l)
```

38.4 Queues

- Another fundamental data structure is the *queue*. A queue stores a collection of objects (of arbitrary type) that are inserted and removed in a *first-in, first-out (FIFO)* order.
- Objects can always be added to the *back* of the queue but the only object accessible at any time is the object that lives at the *front* of the queue i.e. the one which has been longest in the queue.
- The *enqueue* operation is used to add an object to the queue (it goes to the back) while the *dequeue* operation removes the object currently at the front of the queue.

38.5 Queue methods

- An instance `Q` of the queue abstract data type supports at a minimum the following two methods:
  - `Q.enqueue(e)`: Add element `e` to the back of the queue `Q`.
  - `Q.dequeue()`: Remove and return the element at the front of the queue `Q`; an error occurs if the queue `Q` is currently empty.

- The following convenience methods are also often implemented:
  - `Q.first()`: Return a reference to the element at the front of the queue `Q` without removing it; an error occurs if the queue `Q` is currently empty.
  - `Q.is_empty()`: Return `True` if queue `Q` is empty and `False` otherwise.
  - `len(Q)`: Return the number of elements in queue `Q`.

38.6 Implementing a queue with a list

- This is left as a lab exercise.

CHAPTER

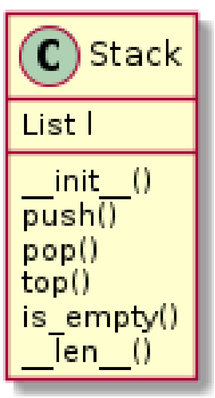
THIRTYNINE

LAB 9.1 (DEADLINE MONDAY 21 MARCH 23:59)

- Upload your code to Einstein to have it verified.

39.1 Stack

- In `stack_091.py` define a `Stack` class to model the stack abstract data type as follows:



- Each box consists of three compartments: class name, data attributes, methods.
- Read the notes on stack methods to determine their required behaviour.
- When your class is correctly implemented, running the following program should produce the given output.

```
from stack_091 import Stack

def main():

    s = Stack()

    print(len(s))
    s.push(1)
```

(continues on next page)

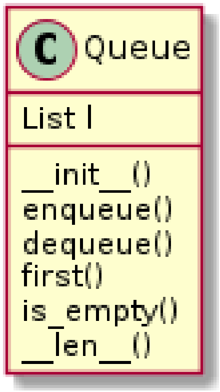
(continued from previous page)

```
    print(s.top())
    print(s.is_empty())
    print(s.pop())
    print(s.is_empty())
    try:
        print(s.pop())
    except IndexError:
        print('Error')
    try:
        print(s.top())
    except IndexError:
        print('Error')
    s.push(1)
    s.push(2)
    s.push(3)
    print(len(s))
    print(s.pop())
    print(s.pop())
    print(s.pop())
    print(s.is_empty())

if __name__ == '__main__':
    main()
```

```
0
1
False
1
True
Error
Error
3
3
2
1
True
```

39.2 Queue

- In `queue_091.py` define a `Queue` class to model the queue abstract data type as follows:



- Each box consists of three compartments: class name, data attributes, methods.
- Read the notes on queue methods to determine their required behaviour.
- When your class is correctly implemented, running the following program should produce the given output.

```
from queue_091 import Queue

def main():

    q = Queue()

    print(len(q))
    q.enqueue(1)
    print(q.first())
    print(q.is_empty())
    print(q.dequeue())
    print(q.is_empty())
    try:
        print(q.dequeue())
    except IndexError:
        print('Error')
    try:
        print(q.first())
    except IndexError:
        print('Error')
    q.enqueue('cat')
    q.enqueue('dog')
    q.enqueue('fish')
    print(len(q))
```

(continues on next page)

(continued from previous page)

```
    print(q.dequeue())
    print(q.dequeue())
    print(q.dequeue())
    print(q.is_empty())

if __name__ == '__main__':
    main()
```

```
0
1
False
1
True
Error
Error
3
cat
dog
fish
True
```

39.3 Brackets

- In `brackets_091.py` define a function called `matcher()` that takes a single string parameter.
- The `matcher()` function checks that all left and right brackets in the supplied string match.
- `matcher()` should return `True` if brackets match and `False` otherwise.
- Brackets that need to be matched are `(){}[]`.
- For example:

```
from brackets_091 import matcher
import sys

tests = ['()',
'(())',
'(({}))',
'(())(){}{(([]))}',
'(()',
'(()){()]',
')(()){([()])}']
```

(continues on next page)

(continued from previous page)

```
def main():

    for test in tests:
        print(matcher(test.strip()))

if __name__ == '__main__':
    main()
```

```
True
True
True
True
False
False
False
```

- Hints:
  1. Make good use of a stack in solving this problem and include a copy of your stack class definition from `stack_091.py` in `brackets_091.py`.
  2. If lefties are `({[` then righties are `)}]`.
  3. Push lefties and pop on meeting a righty.

39.4 RPN calculator

- Reverse Polish Notation (RPN) is a mathematical notation in which every operator follows all of its operands. Examples:
  - `2 + 3` is expressed as `2 3 +`
  - `2 + sqrt(3)` is expressed as `2 3 r +`
  - `1 + 2 * 3` is expressed as `1 2 3 * +`
  - `5 * -2` is expressed as `5 2 n *`
  - `sqrt(-(2*(1-(2+3))))` is expressed as `2 1 2 3 + - * n r`
- In `rpn_091.py` define a function called `calculator()` that takes a single parameter `line` (an RPN expression read from `stdin`). The `calculator()` function computes the value of the RPN expression. Should the RPN expression be invalid then the `calculator()` function raises an `IndexError` exception. For example:

```
from rpn_091 import calculator
import sys

tests = ['5',
'8.5 2 /',
'2 3 +',
'2 3 r +',
'1 2 3 * +',
'5 2 n *',
'1 2 3 + -',
'2 1 2 3 + - *',
'2 1 2 3 + - * n',
'2 1 2 3 + - * n r',
'6 +',
'9 r']

def main():

    for test in tests:
        try:
            a = calculator(test.strip())
            print('{:.2f}'.format(a))
        except IndexError:
            print('Invalid RPN expression')

if __name__ == '__main__':
    main()
```

```
5.00
4.25
5.00
3.73
7.00
-10.00
-4.00
-8.00
8.00
2.83
Invalid RPN expression
3.00
```

- Hints:
  1. Convert all user-supplied numbers to floats.
  2. In solving this problem, again, it might help to make use of a stack.
  3. You might find two dictionaries useful: `binops` maps from each of `+-*/` to a corre-

sponding function while `uniops` maps from each of `nr` to a corresponding function.

4. When you encounter a number in an RPN expression push it onto the stack. If you encounter an operator pop its arguments from the stack (one or two), apply the operator to the popped argument(s) and push the result onto the stack.

5. If after processing an RPN expression you are left with a single number on the stack it is the answer (congratulations!) and your `calculator()` function should return it. Otherwise `calculator()` should raise an `IndexError` exception.

- Here is some code to inspire you:

```
from math import sqrt

binops = {'+': float.__add__,
          '-': float.__sub__,
          '*': float.__mul__,
          '/': float.__truediv__}

uniops = {'n': float.__neg__,
          'r': sqrt}

# To add 3 and 5 we can do something like this...
print(binops['+'](3.0, 5.0))

# To calcualte the square root of 9 we can do something like this..
↪.
print(uniops['r'](9.0))
```

```
8.0
3.0
```

CHAPTER

FORTY

LECTURE 9.2 : RECURSION

40.1 Introduction

- A *recursive* solution is one where the solution to a problem is expressed as an operation on a *simplified* version of the *same* problem.
- For certain problems, recursion may offer an intuitive, simple, and elegant solution.
- The ability to both recognise a problem that lends itself to a recursive solution and to implement that solution is an important skill that will make you a better programmer.
- Furthermore, some programming languages, such as Prolog (which you will meet in second year), make heavy use of recursion.
- We introduce recursion below and implement, in Python, recursive solutions to a selection of programming problems.

40.2 What is recursion?

- Any function that calls itself is *recursive* and exhibits *recursion*.

```
def foo(n):
    return foo(n-1)
```

- The function `foo()` above is recursive i.e. *it calls itself*.
- Let's try calling `foo()` and see what happens:

```
print(foo(10))
```

```
---------------------------------------------------------------------------
↪---------
RecursionError                            Traceback (most recent␣
↪call last)
/tmp/ipykernel_7040/4103014973.py in <module>
----> 1 print(foo(10))

/tmp/ipykernel_7040/879217083.py in foo(n)
      1 def foo(n):
----> 2     return foo(n-1)

... last 1 frames repeated, from the frame below ...

/tmp/ipykernel_7040/879217083.py in foo(n)
      1 def foo(n):
----> 2     return foo(n-1)

RecursionError: maximum recursion depth exceeded
```

- Hmm. Our program crashed! What's going on?
- Well we initially invoke `foo(10)`, which invokes `foo(9)` which invokes `foo(8)` which invokes `foo(7)` which invokes `foo(6)` which invokes…
- Thus our initial `foo(10)` call is the first in an *infinite* sequence of calls to `foo()`.
- Computers do not like an infinite number of anything.
- For each of our `foo()` function invocations Python creates a data structure to represent that particular call to the function.
- That data structure is called a *stack frame*.
- A stack frame occupies memory.
- Our program attempts to create an infinite number of stack frames.
- That would require an infinite amount of memory.
- Our computer does not have an infinite amount of memory so our program crashes (after a while).
- The problem with our recursive function is that it *never* fails to invoke itself and thus exhibits *infinite recursion*.
- To prevent infinite recursion we need to insert a *base case* into our function.

- Let's rewrite our function as `bar()` but this time cause it to stop once its parameter hits zero:

```
def bar(n):
    if n == 0: # base case : no more calls to bar()
        return 0
    return bar(n-1)
```

- Let's try calling `bar()` and see what happens:

```
print(bar(10))
```

```
0
```

- Why does `bar(10)` return zero?
- Well `bar(10)` calls `bar(9)` which calls `bar(8)` ... which calls `bar(0)`.
- The base case is `bar(0)`.
- It returns zero to `bar(1)` which returns zero to `bar(2)` which returns zero to `bar(3)` ... which returns zero to `bar(10)` which returns zero which is our answer.
- That's how recursion works.
- So far so good. But can we use recursion to do something useful?

40.3 Summing the numbers 0 through N

- Assume we have a function `sum_up_to()`.
- Given an argument `N`, `sum_up_to(N)` returns the sum all of the integers 0 through N.
- For example `sum_up_to(10)` sums the sequence 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0.
- Let's look at the `sum_up_to()` function in action:

```
print(sum_up_to(10))
```

```
55
```

- Let's try some more examples:

```
print(sum_up_to(0))
print(sum_up_to(1))
print(sum_up_to(2))
print(sum_up_to(3))
print(sum_up_to(4))
print(sum_up_to(5))
print(sum_up_to(6))
print(sum_up_to(7))
print(sum_up_to(8))
print(sum_up_to(9))
print(sum_up_to(10))
```

```
0
1
3
6
10
15
21
28
36
45
55
```

- Do you notice anything recursive about the above sequence?
- Let's annotate each line to make the recursion obvious:

```
sum_up_to(0)         0        base case returns zero
sum_up_to(1)         1        1 + sum_up_to(0)
sum_up_to(2)         3        2 + sum_up_to(1)
sum_up_to(3)         6        3 + sum_up_to(2)
sum_up_to(4)        10        4 + sum_up_to(3)
sum_up_to(5)        15        5 + sum_up_to(4)
```

(continues on next page)

(continued from previous page)

```
sum_up_to(6)          21          6 + sum_up_to(5)
sum_up_to(7)          28          7 + sum_up_to(6)
sum_up_to(8)          36          8 + sum_up_to(7)
sum_up_to(9)          45          9 + sum_up_to(8)
sum_up_to(10)          55          10 + sum_up_to(9)
```

- For any argument `N`, `sum_up_to(N)` is equal to `N + sum_up_to(N-1)`.
- This is the essence of a recursive solution.
- The solution to the problem `sum_up_to(N)` is broken down into the operation `N +` on a simpler version of the same problem `sum_up_to(N-1)`.
- For example `sum_up_to(10)` is `10 + sum_up_to(9)`.
- The base case ensures recursion stops at some point.
- Our base case encodes the fact that `sum_up_to(0)` is zero.
- Let's write the Python code that implements the `sum_up_to()` function:

```
def sum_up_to(n):
   if n == 0: # base case : no more calls to sum_up_to()
      return 0
   return n + sum_up_to(n-1)
```

- Why does `sum_up_to(10)` return 55?
- Well, `sum_up_to(10)` calls `sum_up_to(9)` which calls `sum_up_to(8) ...` which calls `sum_up_to(0)`.
- The base case is `sum_up_to(0)`.
- The base case returns zero to
  - `sum_up_to(1)` which returns 1 (1+0) to
  - `sum_up_to(2)` which returns 3 (2+1) to
  - `sum_up_to(3)` which returns 6 (3+3) to
  - `sum_up_to(4)` which returns 10 (4+6) to

- `sum_up_to(5)` which returns 15 (5+10) to
- ...
- `sum_up_to(9)` which returns 45 (9+36) to
- `sum_up_to(10)` which returns 55 (10+45) which is our answer.

40.4 Recursive factorial

- Factorial 4 or 4! = 4 * 3 * 2 * 1 and in general N! = N * (N-1) * (N-2) * (N-3) * ... 2 * 1.
- 1! is defined as 1.
- Let's look at some examples of factorial in action:

```
print(factorial(1))
print(factorial(2))
print(factorial(3))
print(factorial(4))
print(factorial(5))
print(factorial(6))
print(factorial(7))
print(factorial(8))
print(factorial(9))
print(factorial(10))
```

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

- Do you notice anything recursive about the above sequence?
- Let's annotate each line to make the recursion obvious:

```
factorial(1)          1              base case returns 1
factorial(2)          2              2 * factorial(1)
factorial(3)          6              3 * factorial(2)
factorial(4)         24              4 * factorial(3)
factorial(5)        120              5 * factorial(4)
factorial(6)        720              6 * factorial(5)
factorial(7)       5040              7 * factorial(6)
factorial(8)      40320              8 * factorial(7)
factorial(9)     362880              9 * factorial(8)
factorial(10)    3628800              10 * factorial(9)
```

- For any argument `N`, `factorial(N)` is equal to `N * factorial(N-1)`.
- This is the essence of a recursive solution.
- The solution to the problem `factorial(N)` is broken down into the operation `N *` on a simpler version of the same problem `factorial(N-1)`.
- For example `factorial(10)` is `10 * factorial(9)`.
- The base case ensures recursion stops at some point.
- The base case encodes the fact that `factorial(1)` is 1.
- Let's write the Python code that implements the `factorial()` function:

```python
def factorial(n):
   if n == 1: # base case : no more calls to factorial()
      return 1
   return n * factorial(n-1)
```

40.5 Fibonacci

- The Fibonacci sequence of numbers is given by: 1, 1, 2, 3, 5, 8, 13, etc.
- The first two numbers of the sequence are both defined to be 1 and thereafter each number in the sequence is defined as the sum of the previous two.
- Let's look at some examples of `fibonacci()` in action:

```python
print(fibonacci(0))
print(fibonacci(1))
print(fibonacci(2))
```

(continues on next page)

(continued from previous page)

```
print(fibonacci(3))
print(fibonacci(4))
print(fibonacci(5))
print(fibonacci(6))
print(fibonacci(7))
print(fibonacci(8))
print(fibonacci(9))
print(fibonacci(10))
```

```
1
1
2
3
5
8
13
21
34
55
89
```

- Do you notice anything recursive about the above sequence?
- Let's annotate each line to make the recursion obvious:

```
fibonacci(0)         1        base case returns 1
fibonacci(1)         1        base case returns 1
fibonacci(2)         2        fibonacci(1) + fibonacci(0)
fibonacci(3)         3        fibonacci(2) + fibonacci(1)
fibonacci(4)         5        fibonacci(3) + fibonacci(2)
fibonacci(5)         8        fibonacci(4) + fibonacci(3)
fibonacci(6)        13        fibonacci(5) + fibonacci(4)
fibonacci(7)        21        fibonacci(6) + fibonacci(5)
fibonacci(8)        34        fibonacci(7) + fibonacci(6)
fibonacci(9)        55        fibonacci(8) + fibonacci(7)
fibonacci(10)        89        fibonacci(9) + fibonacci(8)
```

- In general, `fibonacci(N) = fibonacci(N-1) + fibonacci(N-2)`.

- Our base cases are `fibonacci(0) = 1` and `fibonacci(1) = 1`.
- Let's translate this into Python...

```
def fibonacci(n):
    if n == 0:
        return 1
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

40.6 Reversing a list

- Let's try to come up with a recursive implementation of a function that reverses a list.

CHAPTER

FORTYONE

LAB 9.2 (DEADLINE MONDAY 21 MARCH 23:59)

- Upload your code to Einstein to have it verified.

41.1 Sum up

- In a module named `sumup_092.py` implement a `sumup()` function that when passed an integer n, returns the sum of all numbers 0 through N.
- **Your function must be recursive.**
- When implemented correctly, running the following program should produce the given output.

```
from sumup_092 import sumup

def main():
    print(sumup(0))
    print(sumup(1))
    print(sumup(12))

if __name__ == '__main__':
    main()
```

```
0
1
78
```

41.2 Factorial

- In a module named `factorial_092.py` implement a `factorial()` function that when passed an integer n, returns n! (i.e. n * n-1 * n-2 * … * 2 * 1).
- **Your function must be recursive.**
- When implemented correctly, running the following program should produce the given output.

```
from factorial_092 import factorial

def main():
    print(factorial(0))
    print(factorial(1))
    print(factorial(12))

if __name__ == '__main__':
    main()
```

```
1
1
479001600
```

41.3 Power

- In a module named `power_092.py` implement a `power()` function that when passed two integers m, and n, returns m to the power of n.
- **Your function must be recursive.**
- When implemented correctly, running the following program should produce the given output.

```
from power_092 import power

def main():
    print(power(2,3))
    print(power(4,4))
    print(power(2,32))
    print(power(10,3))
    print(power(8,0))

if __name__ == '__main__':
    main()
```

```
8
256
4294967296
1000
1
```

41.4 Minimum

- In a module named `minimum_092.py` implement a `minimum()` function that when passed a list of integers returns the minimum integer in the list.
- **Your function must be recursive and cannot use the built-in min() function.**
- When implemented correctly, running the following program should produce the given output.

```
from minimum_092 import minimum

def main():
    min = None
    print(minimum([6,5,1,3,4]))
    print(minimum([6,5,11,3,4]))
    print(minimum([6,15,11,13,14]))
    print(minimum([6,15,11,13,4]))

if __name__ == '__main__':
    main()
```

```
1
3
6
4
```

41.5 Maximum

- In a module named `maximum_092.py` implement a `maximum()` function that when passed a list of integers returns the maximum integer in the list.
- **Your function must be recursive and cannot use the built-in max() function.**
- When implemented correctly, running the following program should produce the given output.

```
from maximum_092 import maximum

def main():
    max = None
    print(maximum([6,5,1,3,4]))
    print(maximum([6,5,11,3,4]))
    print(maximum([6,15,11,13,14]))
    print(maximum([6,15,11,13,4]))

if __name__ == '__main__':
    main()
```

```
6
11
15
15
```

41.6 Count

- In a module named `count_092.py` implement a `count_letters()` function that when passed a string returns the number of characters in the string.
- **Your function must be recursive and cannot use the built-in len() function.**
- When implemented correctly, running the following program should produce the given output.

```
from count_092 import count_letters

def main():
    len = None
    print(count_letters('cat'))
    print(count_letters('catastrophe'))
    print(count_letters(''))

if __name__ == '__main__':
    main()
```

```
3
11
0
```

41.7 Reversing a list

- In a module named `reverse_092.py` implement a `reverse_list()` function that returns a new list that is the reverse of the list passed to it.
- **Your function must be recursive.**
- When implemented correctly, running the following program should produce the given output.

```
from reverse_092 import reverse_list

def main():
    print(reverse_list([1,2,3]))
    print(reverse_list([3,4,5,6]))
    print(reverse_list([1,2]))

if __name__ == '__main__':
    main()
```

```
[3, 2, 1]
[6, 5, 4, 3]
[2, 1]
```

41.8 Fibonacci

- In a module named `fibonacci_092.py` implement a `fibonacci()` function that returns the *Nth* number in the Fibonacci sequence.
- **Your function must be recursive.**
- When implemented correctly, running the following program should produce the given output.

```
from fibonacci_092 import fibonacci

def main():
    print(fibonacci(0))
    print(fibonacci(1))
    print(fibonacci(5))
    print(fibonacci(8))

if __name__ == '__main__':
    main()
```

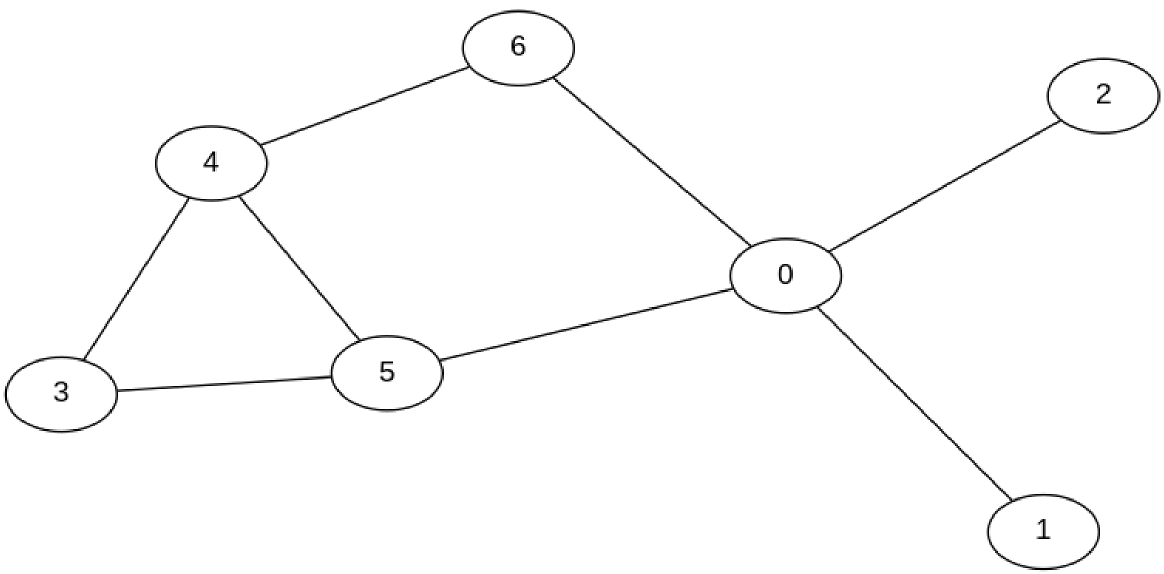
```
1
1
8
34
```

CHAPTER FORTYTWO

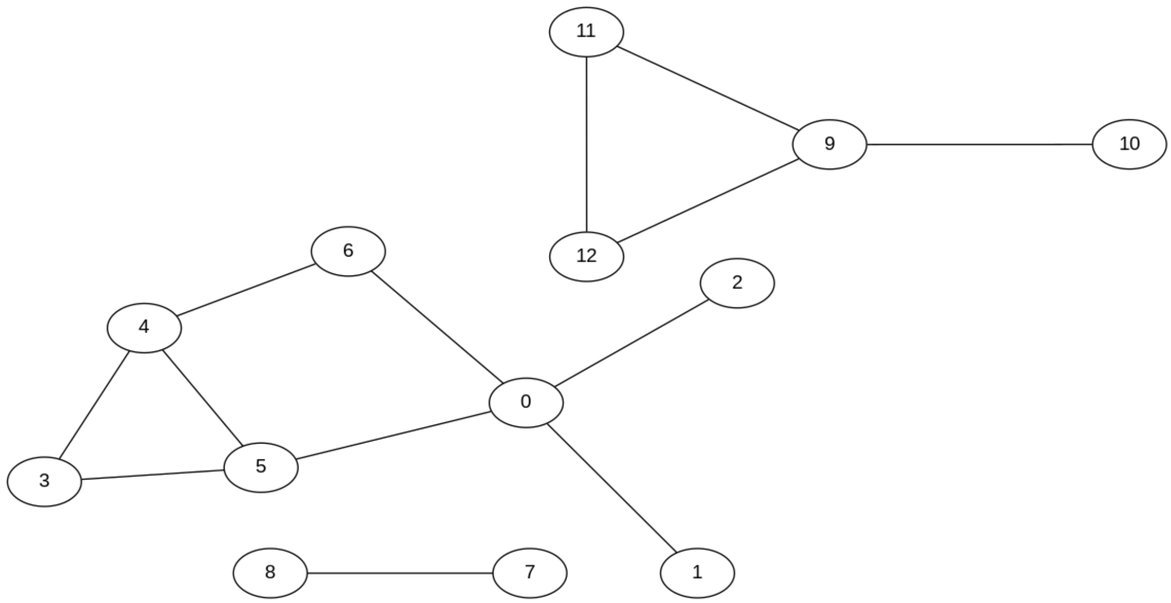
LECTURE 11.1 : GRAPHS

42.1 Introduction

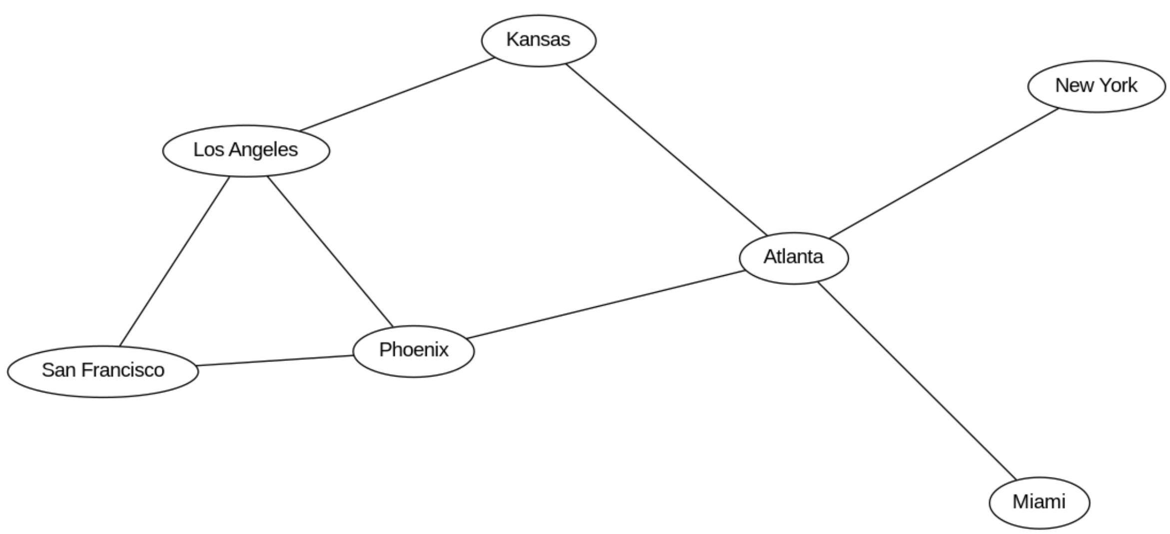
- A graph is a set of vertices connected by pairwise edges.
- Here is an example graph:



- Here is another example (this time with three connected components):



- Graphs have thousands of practical applications i.e. we can apply the graph abstraction to model thousands of real-world problems.
- For example, in a computer network, computers are vertices and fibre optic cables are edges.
- In a social network, people are vertices and friendships are edges.
- In a transport network, airports are vertices and flights are edges.
- Here is a transport network modelled with a graph:



- A *path* is a sequence of vertices connected by edges.
- A *cycle* is a path whose first and last vertices are the same.
- An understanding of and an ability to apply graph algorithms will make you a stand-out programmer.

- We first consider how to represent a graph in Python and then examine a small number of graph algorithms (there are hundreds).

42.2 Graph description

- We will describe a graph with a simple text file where the first line defines the number of vertices in the graph and the following lines define the edges (i.e. which vertices are connected to which).
- The first graph given above is then described by the following text file.

```
$ cat graph01.txt
7
0 1
0 2
0 5
0 6
3 4
3 5
4 5
4 6
```

42.3 Graph representation

- How are we going to represent a graph in Python?
- We will do so by developing a Graph class.
- The vertices can be represented simply by integers.
- If there are V vertices in the graph we will number them *0* to *V-1*.
- What about the edges? How are we going to capture which vertices are connected to which?
- To capture information about edges we will use an *adjacency-list* representation.
- This will involve a dictionary called *adj* which maps from vertex number (dictionary keys) to a list of vertices connected to that vertex (dictionary values).

42.4 Python implementation

```
class Graph(object):

   def __init__(self, V):
      self.V = V
      self.adj = {}
      for v in range(V):
         self.adj[v] = list()

   def addEdge(self, v, w):
      self.adj[v].append(w)
      self.adj[w].append(v)
```

42.5 Initialising our graph

- Below we initialise a graph based on its text description.

```
with open('graph01.txt') as f:
   V = int(f.readline())
   g = Graph(V)

   for line in f:
      v, w = [int(t) for t in line.strip().split()]
      g.addEdge(v, w)
```

42.6 Computing the degree of a vertex

- The degree of a vertex is the number of edges connecting it.
- Let's add a method to our Graph class to return the degree of any vertex v.

```
class Graph(object):

   def __init__(self, V):
      self.V = V
      self.adj = {}
      for v in range(V):
         self.adj[v] = list()

   def addEdge(self, v, w):
```

(continues on next page)

(continued from previous page)

```
        self.adj[v].append(w)
        self.adj[w].append(v)

    def degree(self, v):
        return len(self.adj[v])
```

```
with open('graph01.txt') as f:
    V = int(f.readline())
    g = Graph(V)

    for line in f:
        v, w = [int(t) for t in line.strip().split()]
        g.addEdge(v, w)

print(g.degree(0))
print(g.degree(1))
print(g.degree(4))
```

```
4
1
3
```

42.7 Computing the maximum degree

- Let's add a method to our Graph class to return the maximum degree of all vertices.

```
class Graph(object):

    def __init__(self, V):
        self.V = V
        self.adj = {}
        for v in range(V):
            self.adj[v] = list()

    def addEdge(self, v, w):
        self.adj[v].append(w)
        self.adj[w].append(v)

    def degree(self, v):
        return len(self.adj[v])

    def maxDegree(self):
        return max([self.degree(v) for v in range(self.V)])
```

```python
with open('graph01.txt') as f:
    V = int(f.readline())
    g = Graph(V)

    for line in f:
        v, w = [int(t) for t in line.strip().split()]
        g.addEdge(v, w)

print(g.maxDegree())
```

```
4
```

42.8 Computing the average degree

- Let's add a method to our Graph class to return the average degree of all vertices.

```python
class Graph(object):

    def __init__(self, V):
        self.V = V
        self.adj = {}
        for v in range(V):
            self.adj[v] = list()

    def addEdge(self, v, w):
        self.adj[v].append(w)
        self.adj[w].append(v)

    def degree(self, v):
        return len(self.adj[v])

    def maxDegree(self):
        return max([self.degree(v) for v in range(self.V)])

    def avgDegree(self):
        return sum([self.degree(v) for v in range(self.V)]) / self.V
```

```python
with open('graph01.txt') as f:
    V = int(f.readline())
    g = Graph(V)

    for line in f:
        v, w = [int(t) for t in line.strip().split()]
```

(continues on next page)

(continued from previous page)

```
        g.addEdge(v, w)

print(g.avgDegree())
```

```
2.2857142857142856
```

CHAPTER

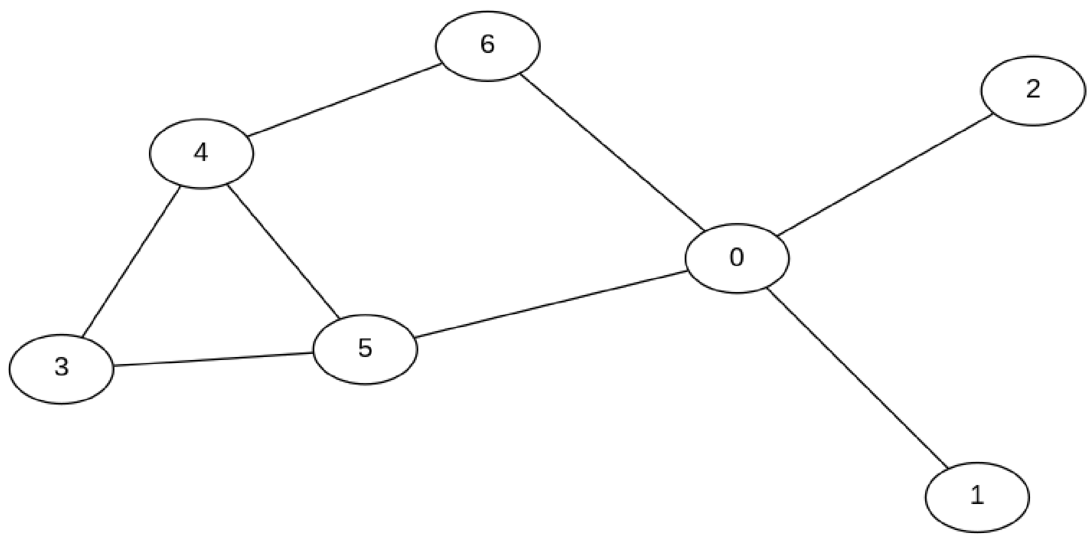
FORTYTHREE

LECTURE 11.2 : SEARCHING GRAPHS

43.1 Introduction

- We present an approach to searching through a graph called *depth-first search*.
- Depth-first search (DFS) is a recursive algorithm that uses back-tracking to identify and explore novel paths.
- DFS can be used to find all vertices connected to a given vertex.
- DFS can be used to find a path between two vertices (should one exist).
- Before we code it, let's look at DFS in action so we can see how it works.
- Here's a video I made.
- Here's a website with DFS animations.
- Here's a website with lots of algorithm animations.

43.2 Our graph



43.3 Graph description

- As usual, we describe a graph with a simple text file where the first line defines the number of vertices in the graph and the following lines define the edges (i.e. which vertices are connected to which).

```
$ cat graph01.txt
7
0 1
0 2
0 5
0 6
3 4
3 5
4 5
4 6
```

43.4 Basic graph class

- Our basic graph class looks as follows.

```
class Graph(object):

   def __init__(self, V):
      self.V = V
      self.adj = {}
      for v in range(V):
         self.adj[v] = list()

   def addEdge(self, v, w):
      self.adj[v].append(w)
      self.adj[w].append(v)
```

43.5 Coding DFS

- We will not add a new method to the `Graph` class but will instead create a new `DFSPaths` class.
- The input to the `DFSPaths` class is the graph we wish to explore using DFS and a starting vertex.

```
class DFSPaths(object):

   def __init__(self, g, s):
      self.g = g
      self.s = s
      self.visited = [False for _ in range(g.V)]
      self.parent = [False for _ in range(g.V)]
      self.dfs(s)

   def dfs(self, v):
      self.visited[v] = True
      for w in self.g.adj[v]:
         if not self.visited[w]:
            self.parent[w] = v
            self.dfs(w)

   # Return True if there is a path from s to v
   def hasPathTo(self, v):
      # This is for you to write
      pass
```

(continues on next page)

(continued from previous page)

```
    # Return path from s to v (or None should one not exist)
    def pathTo(self, v):
        # This is for you to write
        pass
```

43.6 Applying DFS to a graph

```
from graph import Graph, DFSPaths

with open('graph01.txt') as f:

    V = int(f.readline())

    g = Graph(V)

    for line in f:

        v, w = [int(t) for t in line.strip().split()]
        g.addEdge(v, w)

    paths = DFSPaths(g, 0)

    print(paths.hasPathTo(6))

    print(paths.pathTo(6))
```

```
True
[0, 5, 3, 4, 6]
```

CHAPTER

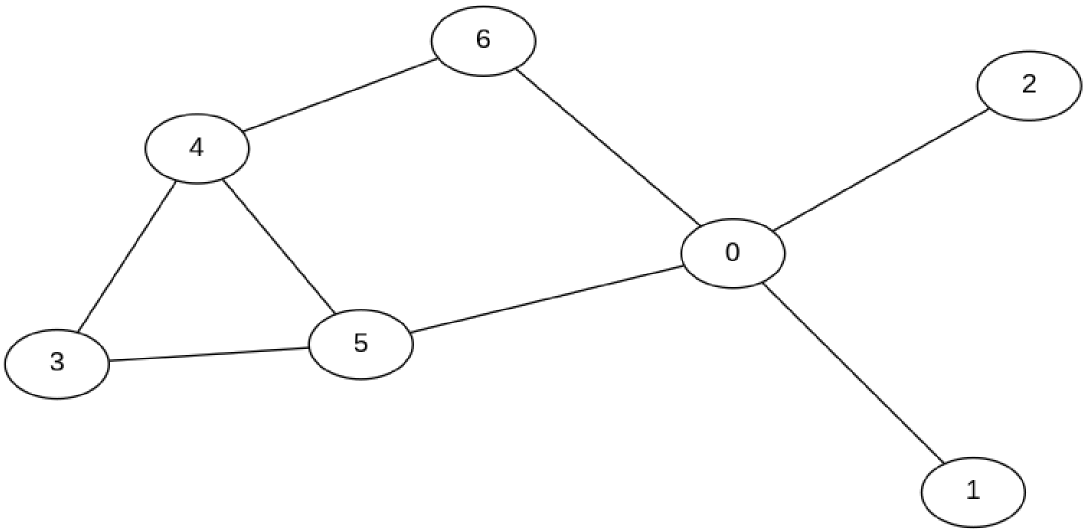
FORTYFOUR

LECTURE 11.3 : SEARCHING GRAPHS (AGAIN)

44.1 Introduction

- We present an approach to searching through a graph called *breadth-first search*.
- We previously saw that depth-first search (DFS) is a recursive algorithm that uses back-tracking to identify and explore novel paths.
- An alternative to DFS is breadth-first search (BFS).
- BFS does not use recursion but instead makes use of a queue.
- BFS can be used to find all vertices connected to a given vertex.
- BFS can be used to find a path betwen two vertices (should one exist).
- Before we code it, let's look at BFS in action so we can see how it works.
- Here's a video I made.
- Here's a website with BFS animations.
- Here's a website with lots of algorithm animations.

44.2 Our graph



44.3 Graph description

- As usual, we describe a graph with a simple text file where the first line defines the number of vertices in the graph and the following lines define the edges (i.e. which vertices are connected to which).

```
$ cat graph01.txt
7
0 1
0 2
0 5
0 6
3 4
3 5
4 5
4 6
```

44.4 Basic graph class

- Our basic graph class looks as follows.

```
class Graph(object):

   def __init__(self, V):
      self.V = V
      self.adj = {}
      for v in range(V):
         self.adj[v] = list()

   def addEdge(self, v, w):
      self.adj[v].append(w)
      self.adj[w].append(v)
```

44.5 Coding BFS

- We will not add a new method to the `Graph` class but will instead create a new `BFSPaths` class.
- The input to the `BFSPaths` class is the graph we wish to explore using BFS and a starting vertex.

```
class BFSPaths(object):

   def __init__(self, g, s):
      self.g = g
      self.s = s
      self.marked = [False for _ in range(g.V)]
      self.parent = [False for _ in range(g.V)]
      self.bfs(s)

   def bfs(self, s):
      queue = [s]
      self.marked[s] = True

      while queue:
         v = queue.pop(0)
         for w in self.adj[v]:
            if not self.marked[w]:
               queue.append(w)
               self.parent[w] = v
               self.marked[w] = True
```

(continues on next page)

(continued from previous page)

```
    # Return True if there is a path from s to v
    def hasPathTo(self, v):
        return self.marked[v]

    # Return path from s to v (or None should one not exist)
    def pathTo(self, v):
        if not self.hasPathTo(v):
            return None
        path = [v]
        while v != self.s:
            v = self.parent[v]
            path.append(v)
        return path[::-1]
```

44.6 Applying BFS to a graph

```
from graph import Graph, BFSPaths

with open('graph01.txt') as f:

    V = int(f.readline())

    g = Graph(V)

    for line in f:

        v, w = [int(t) for t in line.strip().split()]
        g.addEdge(v, w)

    paths = BFSPaths(g, 0)

    print(paths.hasPathTo(6))

    print(paths.pathTo(6))
```

```
True
[0, 6]
```

CHAPTER

FORTYFIVE

LAB 11.1 : SAMPLE LAB EXAM (DEADLINE MONDAY 28 MARCH 23:59)

45.1 Before starting

- The exam runs 1400-1550.
- Answer all questions.
- Upload all code to Einstein.
- All *lab exam rules* apply.

45.2 Triathlete part 1 [10 marks]

- In module `triathlete_v1_111.py` define a `Triathlete` class to model a triathlete.
- A triathlete has a name and ID number.
- When your class is correctly implemented, running the following program should produce the given output.

```
from triathlete_v1_111 import Triathlete

def main():
    t1 = Triathlete('Ian Brown', 21)
    t2 = Triathlete('John Squire', 22)

    assert(t1.name == 'Ian Brown')
    assert(t1.tid == 21)
```

(continues on next page)

(continued from previous page)

```
    print(t1)
    print(t2)

if __name__ == '__main__':
    main()
```

```
Name: Ian Brown
ID: 21
Name: John Squire
ID: 22
```

45.3 Triathlete part 2 [10 Marks]

- In module `triathlete_v2_111.py` extend the `Triathlete` class to support the recording of per-discipline times for a triathlete.
- Disciplines are swimming, cycling and running and times are recorded in seconds.
- When your class is correctly implemented, running the following program should produce the given output.

```
from triathlete_v2_111 import Triathlete

def main():

    t1 = Triathlete('Ian Brown', 21)

    t1.add_time('swim', 100)
    t1.add_time('cycle', 120)
    t1.add_time('run', 150)

    print('Cycle: {}'.format(t1.get_time('cycle')))
    print(t1)

if __name__ == '__main__':
    main()
```

```
Cycle: 120
Name: Ian Brown
ID: 21
Race time: 370
```

45.4 Triathlete part 3 [15 Marks]

- In module `triathlete_v3_111.py` extend the `Triathlete` class to support the comparison of triathletes.
- Comparison is carried out in terms of a triathlete's race time.
- When your class is correctly implemented, running the following program should produce the given output.

```
from triathlete_v3_111 import Triathlete

def main():

    t1 = Triathlete('Ian Brown', 21)
    t2 = Triathlete('John Squire', 22)
    t3 = Triathlete('Alan Wren', 23)

    t1.add_time('swim', 100)
    t1.add_time('cycle', 120)
    t1.add_time('run', 150)

    t2.add_time('swim', 300)
    t2.add_time('cycle', 100)
    t2.add_time('run', 200)

    t3.add_time('swim', 150)
    t3.add_time('cycle', 120)
    t3.add_time('run', 100)

    print(t1)
    print(t2)
    print(t3)

    assert(t1 == t3)
    assert(t1 < t2)
    assert(t2 > t3)

if __name__ == '__main__':
    main()
```

```
Name: Ian Brown
ID: 21
Race time: 370
Name: John Squire
ID: 22
```

(continues on next page)

(continued from previous page)

```
Race time: 600
Name: Alan Wren
ID: 23
Race time: 370
```

45.5 Triathlon part 1 [15 Marks]

- In module `triathlon_v1_111.py` define a `Triathlon` class to model a collection of triathletes.
- A triathlon is essentially a mapping from triathlete IDs to `Triathlete` objects.
- **You must include in triathlon_v1_111.py a copy of your Triathlete class definition from triathlete_v1_111.py.**
- Triathletes can be added to and removed from the triathlon via the `add()` and `remove()` methods respectively.
- A `lookup()` method returns a `Triathlete` object if a given triathlete is in the triathlon and `None` otherwise.
- When your class is correctly implemented, running the following program should produce no output.

```python
from triathlon_v1_111 import Triathlete, Triathlon

def main():

    tn = Triathlon()
    t1 = Triathlete('Ian Brown', 21)
    t2 = Triathlete('John Squire', 22)

    tn.add(t1)
    tn.add(t2)

    t = tn.lookup(21)
    assert(isinstance(t, Triathlete))
    assert(t.name == 'Ian Brown')
    assert(t.tid == 21)

    tn.remove(21)
    t = tn.lookup(21)
    assert(t is None)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

45.6 Triathlon part 2 [15 Marks]

- In module `triathlon_v2_111.py` extend the `Triathlon` class to support the printing of a triathlon.
- Printing a triathlon prints all triathlete details in alphabetical order of their names.
- **You must include in triathlon_v2_111.py a copy of your Triathlete class definition from triathlete_v1_111.py.**
- When your class is correctly implemented, running the following program should produce the given output.

```
from triathlon_v2_111 import Triathlete, Triathlon

def main():

    tn = Triathlon()
    t1 = Triathlete('Ian Brown', 21)
    t2 = Triathlete('John Squire', 22)
    t3 = Triathlete('Alan Wren', 23)

    tn.add(t1)
    tn.add(t2)
    tn.add(t3)

    print(tn)

if __name__ == '__main__':
    main()
```

```
Name: Alan Wren
ID: 23
Name: Ian Brown
ID: 21
Name: John Squire
ID: 22
```

45.7 Triathlon part 3 [15 Marks]

- In module `triathlon_v3_111.py` extend the `Triathlon` class to support retrieval of the Triathletes with the best and worst race times.
- **You must include in triathlon_v3_111.py a copy of your Triathlete class definition from Triathlete_v3_111.py.**
- When your class is correctly implemented, running the following program should produce the given output.

```
from triathlon_v3_111 import Triathlete, Triathlon

def main():

    tn = Triathlon()
    t1 = Triathlete('Ian Brown', 21)
    t2 = Triathlete('John Squire', 22)
    t3 = Triathlete('Alan Wren', 23)

    t1.add_time('swim', 100)
    t1.add_time('cycle', 120)
    t1.add_time('run', 150)

    t2.add_time('swim', 300)
    t2.add_time('cycle', 100)
    t2.add_time('run', 200)

    t3.add_time('swim', 50)
    t3.add_time('cycle', 20)
    t3.add_time('run', 10)

    tn.add(t1)
    tn.add(t2)
    tn.add(t3)

    print(tn.best())
    print(tn.worst())

if __name__ == '__main__':
    main()
```

```
Name: Alan Wren
ID: 23
Race time: 80
Name: John Squire
```

(continues on next page)

(continued from previous page)

```
ID: 22
Race time: 600
```

45.8 Graph search [20 Marks]

- Below are the current contents of module `graph_111.py`.

```
class Graph(object):

    def __init__(self, V):
        self.adj = {}
        self.V = V
        for v in range(V):
            self.adj[v] = list()

    def addEdge(self, v, w):
        self.adj[v].append(w)
        self.adj[w].append(v)

class DFSPaths(object):

    def __init__(self, g, s):
        self.g = g
        self.s = s
        self.visited = [False for _ in range(g.V)]
        self.parent = [False for _ in range(g.V)]
        self.dfs(s)

    def dfs(self, v):
        self.visited[v] = True
        for w in self.g.adj[v]:
            if not self.visited[w]:
                self.parent[w] = v
                self.dfs(w)

    def hasPathTo(self, v):
        pass

    def pathTo(self, v):
        pass
```

- Complete the `hasPathTo()` and `pathTo()` methods.
- The contents of `graph01.txt` are as follows.

```
$ cat graph01.txt
7
0 1
0 2
0 5
0 6
3 4
3 5
4 5
4 6
```

- When your class is correctly implemented, running the following program should produce the given output.

```python
#!/usr/bin/env python3

import sys

from graph_111 import Graph, DFSPaths

def main():

    with open('graph01.txt') as f:

        V = int(f.readline())

        g = Graph(V)

        for line in f:

            v, w = [int(t) for t in line.strip().split()]
            g.addEdge(v, w)

    paths = DFSPaths(g, 0)

    print(paths.hasPathTo(6))
    print(paths.pathTo(6))

if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    main()
```

```
True
[0, 5, 3, 4, 6]
```

CHAPTER FORTYSIX

LAB 11.2 (DEADLINE MONDAY 4 APRIL 23:59)

Some miscellaneous exercises (not graph-related).

46.1 Double vowels

- Write a program called `doubles_112.py` that reads a list of words from `stdin` (one word per line and all lower-case).
- Your program should print the word that contains most *double vowels* (two successive instances of the same vowel constitute a double vowel).
- For example:

```
$ cat doubles_stdin_00_112.txt
artist
engineer
beekeeper
programmer
```

```
$ python3 doubles_112.py < doubles_stdin_00_112.txt
beekeeper
```

- Note that each vowel can be used only once in a double vowel i.e. the string *eee* contains a single double vowel.
- You can assume there will always be a unique winner.

46.2 No duplicates

- Write a program called `nodups_112.py` that reads text from `stdin`.
- Your program should output the same text with every subsequent occurrence of a word (after its first) replaced by a full-stop.
- Ignore case when comparing words.
- For example:

```
$ cat nodups_stdin_00_112.txt
Once upon a time there was a Wicked Witch.
In fact there were so many wicked witches
it was hard to tell which witch was which!
As a result, a walk through the dark forest
was, for Rapunzel and Snow White, fraught
with danger.
```

```
$ python3 nodups_112.py < nodups_stdin_00_112.txt
Once upon a time there was . Wicked Witch.
In fact . were so many . witches
it . hard to tell which . . .
As . result, . walk through the dark forest
. for Rapunzel and Snow White, fraught
with danger.
```

46.3 Symmetric order

- Write a program called `symmetric_112.py` that reads a list of names (ordered by increasing length) from `stdin`.
- Names on consecutive lines (i.e. lines k and k+1, where k is even) have the same length and are pairs.
- Your program should output the same list of names reordered to be symmetric around the longest name(s) in the list i.e. with elements of each pair moved to opposite ends of the list (with the first name in each pair above the second).
- Confused? An example should help:

```
$ cat symmetric_stdin_00_112.txt
Abe
Max
Mary
```

(continues on next page)

(continued from previous page)

```
Jane
Polly
Timmy
```

```
$ python3 symmetric_112.py < symmetric_stdin_00_112.txt
Abe
Mary
Polly
Timmy
Jane
Max
```

- The last name in the list may not be part of a pair. Here is one such example:

```
$ cat symmetric_stdin_01_112.txt
Ben
Una
Sylvia
Thomas
Penelope
```

```
$ python3 symmetric_112.py < symmetric_stdin_01_112.txt
Ben
Sylvia
Penelope
Thomas
Una
```

- You can assume there will always be at least one name in the input list.

46.4 Code breaker

- Jimmy has invented a ground-breaking encryption algorithm.
- After each vowel he adds a *p* followed by the vowel again.
- Write a program called `decode_112.py` that decodes Jimmy's messages.
- Messages (all lower-case) are read, one per line, from `stdin`.
- For example:

```
$ cat decode_stdin_00_112.txt
papapapa
```

(continues on next page)

(continued from previous page)

```
papapripikapa
prepetty popolly
pepeteper pipipeper pipickeped apa pepeck opof pipickleped␣
↪pepeppepers
jipimmy lopovepes sapally opo'bripiepen
```

```
$ python3 decode_112.py < decode_stdin_00_112.txt
papa
paprika
pretty polly
peter piper picked a peck of pickled peppers
jimmy loves sally o'brien
```

CHAPTER FORTYSEVEN

LECTURE 12.1 : GOODBYE

- We are done!