

Lesson 1: Permutations, Heap's algorithm and cryptarithms

Lesson ID: 53359 | Created: 2024-05-27T01:25:56.793607+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 384763 | Index: 1 | Title: Permutations, Heap's algorithm and cryptarithms | Status: unseen

Slide ID: 362186 | Index: 2 | Title: Permutations, Heap's algorithm and cryptarithms | Status: unseen

Lesson 2: Exam Questions with Doctest

Lesson ID: 55026 | Created: 2024-06-15T11:04:30.570648+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 372901 | Index: 10 | Title: Python 3 Cheat Sheet | Status: unseen

Slide ID: 372902 | Index: 11 | Title: Example Exam Question with Doctest | Status: unseen

The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest: To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented. To perform regression testing by verifying that interactive examples from a test file or a test object work as expected. To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation". For the sake of exam questions in this course, we are using way (1) above. We will look at a complete but small example module called example.py For more information about doctest module, see <https://docs.python.org/3/library/doctest.html>

Lesson 3: Week 4 - Notes 5 From Decimal Expansions to Reduced Fractions

Lesson ID: 53314 | Created: 2024-05-27T01:25:38.841144+10:00 | State: active | Status: unattempted | Slide Count: 4

Slides

- Slide ID: 361995 | Index: 2 | Title: From decimal expansions to reduced fractions | Status: unseen
- Slide ID: 361996 | Index: 3 | Title: From decimal expansions to reduced fractions | Status: unseen
- Slide ID: 361997 | Index: 4 | Title: Rational Number | Status: unseen
- Slide ID: 361998 | Index: 5 | Title: From decimal expansions to reduced fractions | Status: unseen

Lesson 4: Special Consideration and Application Process

Lesson ID: 53285 | Created: 2024-05-27T01:25:35.688903+10:00 | State: active | Status: unattempted | Slide Count: 1

Slides

Slide ID: 361890 | Index: 3 | Title: Special Consideration and Application Process | Status: unseen

<https://www.student.unsw.edu.au/special-consideration>

Lesson 5: Blackboard Collaborate - Online Tutorials

Lesson ID: 53288 | **Created:** 2024-05-27T01:25:35.833317+10:00 | **State:** active | **Status:** unattempted | **Slide Count:** 1

Slides

Slide ID: 361893 | **Index:** 1 | **Title:** Online Tutorials using Collaborate | **Status:** unseen

<https://moodle.telt.unsw.edu.au/mod/lti/view.php?id=6800190>

Lesson 6: Meet Karel the Robot

Lesson ID: 53291 | Created: 2024-05-27T01:25:35.951175+10:00 | State: active | Status: unattempted | Slide Count: 8

Slides

Slide ID: 361896 | Index: 1 | Title: 1. Meet Karel | Status: unseen

In the 1970s, a Stanford graduate student, Rich Pattis decided that it would be easier to teach the fundamentals of programming if students could somehow learn the basic ideas in a simple environment free from the complexities that characterize most programming languages. Rich designed an introductory programming environment in which students teach a robot to solve simple problems. That robot was named Karel, after the Czech playwright Karel Čapek, whose 1923 play R.U.R. (Rossum's Universal Robots) gave the word robot to the English language. Karel the Robot was quite a success. Karel has been used in introductory computer science courses all across the world and has been taught to millions of students. Many generations of students have learned how programming works with Karel, and it is still the gentle introduction to coding used at Stanford. What is Karel? Karel is a very simple robot living in a very simple world. By giving Karel a set of commands, you can direct it to perform certain tasks within its world. The process of specifying those commands is called programming. Initially, Karel understands only a very small number of predefined commands, but an important part of the programming process is teaching Karel new commands that extend its capabilities. Karel programs have much the same structure and involve the same fundamental elements as Python, a major programming language. The critical difference is that Karel's programming language is extremely small and as such the details are easy to master. Even so, you will discover that solving a problem can be challenging. By starting with Karel, you can concentrate on solving problems from the very beginning. Problem solving is the essence of programming. And because Karel encourages imagination and creativity, you can have quite a lot of fun along the way.

```
def main():
```

```
    move()
```

```
    for c in range(4):
```

```
        put_beeper()
```

```
    move()
```

```
    for c in range(3):
```

```
        while front_is_clear():
```

```
            move()
```

```
            put_beeper()
```

```
            turn_left()
```

```
        move()
```

```
{ "width": 6, "height": 4, "initialX": 0, "initialY": 0, "initialD": 1, "initialB": 1000, "cells": [] }
```

 Karel's world

Karel's world is defined by rows running horizontally (east-west) and columns running vertically (north-south). The

intersection of a row and a column is called a corner. Karel can only be positioned on corners and must be

facing one of the four standard compass directions (north, south, east, west). A sample Karel world is

shown below. Here Karel is located at the corner of 1st row and 1st column, facing east. Several other

components of Karel's world can be seen in this example. The object in front of Karel is a beeper. As

described in Rich Pattis's book, beepers are "plastic cones which emit a quiet beeping noise." Karel can

only detect a beeper if it is on the same corner. The solid lines in the diagram are walls. Walls serve as

barriers within Karel's world. Karel cannot walk through walls and must instead go around them. Karel's

world is always bounded by walls along the edges, but the world may have different dimensions depending

on the specific problem Karel needs to solve. Karel's commands

When Karel is shipped from the factory, it responds to a very small set of commands.

Command	Description
---------	-------------

move()Asks Karel to move forward one block. Karel cannot respond to a move() command if there is a wall blocking its way.

turn_left()Asks Karel to rotate 90 degrees to the left (counterclockwise).

pick_beeper()Asks Karel to pick up one beeper from a corner and stores the beeper in its beeper bag, which can hold an infinite number of beepers. Karel cannot respond to a pick_beeper() command unless there is a beeper on the current corner.

put_beeper()Asks Karel to take a beeper from its beeper bag and put it down on the current corner. Karel cannot respond to a put_beeper() command unless there are beepers in its beeper bag.

The empty pair of parentheses that appears in each of these commands is part of the common syntax shared by Karel and Python and is used to specify the...

Slide ID: 361897 | Index: 2 | Title: 2. Programming Karel | Status: unseen

The simplest style of Karel program uses text to specify a sequence of built-in commands that should be executed when the program is run. Consider the simple Karel program below. The text on the left is the program. The state of Karel's world is shown on the right:# Program: FirstKarel

```
# -----
```

```
# The FirstKarel program defines a "main" function
# with three commands. These commands cause Karel
# to move forward one block, pick up a beeper and
# then move ahead to the next corner.
```

```
from karel.stanfordkarel import *
```

```
def main():
```

```
    move()
```

```
    pick_beeper()
```

```
    move(){"width":6,"height":4,"initialX":0,"initialY":0,"initialID":1,"initialB":1000,"cells":{"x":1,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":2,"y":0,"wall":{"0":false,"1":true,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":3,"y":0,"wall":{"0":true,"1":false,"2":false,"3":true},"beepers":0,"colour":"BLANK"},{"x":4,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":5,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":3,"y":1,"wall":{"0":false,"1":false,"2":true,"3":false},"beepers":0,"colour":"BLANK"},{"x":4,"y":1,"wall":{"0":false,"1":false,"2":true,"3":false},"beepers":0,"colour":"BLANK"},{"x":5,"y":1,"wall":{"0":false,"1":false,"2":true,"3":false},"beepers":0,"colour":"BLANK"}]]
```

Press the "Run" button to execute the program. Programs are typically written in a text editor or a special application called an IDE. This reader has the ability to execute programs in order to help you see how things work as you learn. The program is composed of several parts. The first part consists of the following lines:# Program: FirstKarel

```
# -----
```

```
# The FirstKarel program defines a "main" function
# with three commands. These commands cause Karel
# to move forward one block, pick up a beeper
# and then move ahead to the next corner.
```

These lines are an example of a comment, which is simply text designed to explain the operation of the program to human readers. Comments in both Karel and Python begin with the characters # and include the rest of the line. In a simple program, extensive comments may seem silly because the effect of the program is obvious, but they are extremely important as a means of documenting the design of larger, more complex programs. The second part of the program is the line:from karel.stanford import *

This line requests the inclusion of all definitions from the karel.stanford library. This library contains the

basic definitions necessary for writing Karel programs, such as the definitions of the standard operations `move()` and `pick_beeper()`. Because you always need access to these operations, every Karel program you write will include this import command before you write the actual program. The final part of the Karel program consists of the following function definition:

```
def main():
    move()
    pick_beeper()
    move()
```

These lines represent the definition of a new function, which specifies the sequence of steps necessary to respond to a command. As in the case of the `FirstKarel` program itself, the function definition consists of two parts that can be considered separately: The first line constitutes the function header and the indented code following is the function body. If you ignore the body for now, the function definition looks like this:

body of the function definition

The first word in the function header, `def`, is part of Python's syntactic structure. It says that you are creating a new function. The next word on the header line specifies the name of the new function, which in this case is `main`. Defining a function means that Karel can now respond to a new command with that name. The `main()` command plays a special role in a Karel program. When you start a Karel program it creates a new Karel instance, adds that Karel to a world that you specify, and then issues the `main()` command. The eff...

Slide ID: 361898 | **Index:** 3 | **Title:** 3. Defining New Functions | **Status:** unseen

In the last chapter we wrote a program to help Karel climb a simple ledge:

```
# -----
```

```
# Karel picks up a beeper and places it on a ledge.
```

```
from karel.stanfordkarel import *
```

```
def main():
    move()
    pick_beeper()
    move()
    turn_left()
    move()
    turn_left()
    turn_left()
    turn_left()
    move()
    move()
    put_beeper()
    move()
```

{ "width":6, "height":4, "initialX":0, "initialY":0, "initialID":1, "initialIB":1000, "cells":{ "x":1, "y":0, "wall":{ "0":false, "1":false, "2":false, "3":false }, "beepers":1, "colour":"BLANK" }, { "x":2, "y":0, "wall":{ "0":false, "1":true, "2":false, "3":false }, "beepers":0, "colour":"BLANK" }, { "x":3, "y":0, "wall":{ "0":true, "1":false, "2":false, "3":true }, "beepers":0, "colour":"BLANK" }, { "x":4, "y":0, "wall":{ "0":true, "1":false, "2":false, "3":false }, "beepers":0, "colour":"BLANK" }, { "x":5, "y":0, "wall":{ "0":true, "1":false, "2":false, "3":false }, "beepers":0, "colour":"BLANK" }, { "x":3, "y":1, "wall":{ "0":false, "1":false, "2":true, "3":false }, "beepers":0, "colour":"BLANK" }, { "x":4, "y":1, "wall":{ "0":false, "1":false, "2":true, "3":false }, "beepers":0, "colour":"BLANK" }, { "x":5, "y":1, "wall":{ "0":false, "1":false, "2":true, "3":false }, "beepers":0, "colour":"BLANK" } } }

Even though the `FirstKarel` program above demonstrates that it is possible to perform the `turn_right()` operation using only Karel's built-in commands, the resulting program is not particularly clear conceptually. In your mental design of the program, Karel turns right when it reaches the top of the ledge. The fact that you have to use three `turn_left()` commands to do so is annoying. It would be much simpler if you could simply say `turn_right()` and have Karel understand this command. The resulting program would not only be shorter and easier to write, but also significantly easier to read.

Commands Fortunately, the Karel programming language makes it possible to define new commands simply by including new function definitions. Whenever you have a sequence of Karel commands that performs some useful task--such as turning right--you can define a new function that executes that sequence of commands. The format for defining a new Karel function has much the same as the definition of `main()` in the preceding examples, which is a function definition in its own right. A typical function definition looks like this:

```
def name():  
    commands that make up the body of the function
```

In this pattern, `name` represents the name you have chosen for the new function. To complete the definition, all you have to do is provide the sequence of commands in the lines after the colon, which are all indented by one tab. For example, you can define `turn_right()` as follows:

```
def turn_right():  
    turn_left()  
    turn_left()  
    turn_left()
```

Similarly, you could define a new `turn_around()` function like this:

```
def turn_around():  
    turn_left()  
    turn_left()
```

You can use the name of a new function just like any of Karel's built-in commands. For example, once you have defined `turn_right()`, you could replace the three `turn_left()` commands in the program with a single call to the `turn_right()` function. Here is a revised implementation of the program that uses `turn_right()`:

Program: `BeeperPickingKarel`

```
# -----  
# The BeeperPickingKarel program defines a "main"  
# function with three commands. These commands cause  
# Karel to move forward one block, pick up a  
# beeper and then move ahead to the next corner.
```

```
from karel.stanfordkarel import *
```

```
def main():  
    move()  
    pick_beeper()  
    move()  
    turn_left()  
    move()  
    turn_right()  
    move()  
    move()  
    put_beeper()  
    move()
```

```
def turn_right():  
    turn_left()  
    turn_left()  
    turn_left()
```

`{"width":6,"height":4,"initialX":0,"initialY":0,"initialID":1,"initialB":1000,"cells":{"x":1,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},"x":2,"y":0,"wall":{"0":false,"1":true,"2":false,"3":false},"beepers":0,"colour":"BLANK"},"x":3,"y":0,"wall":{"0":true,"1":false,"2":false,"3":true},"beepers":0,"colour":"BLANK"},"x":4,"y":0,"wall":{"0":true,"1":false,"...`

Slide ID: 361899 | Index: 4 | Title: 4. Decomposition | Status: unseen

As a way of illustrating more of the power that comes with being able to define new functions, it's useful to have Karel do something a little more practical than move a beeper from one place to another. The roadways often seem to be in need of repair, and it might be fun to see if Karel can fill potholes in its

abstract world. For example, imagine that Karel is standing on the “road” shown in the following figure, one corner to the left of a pothole in the road. Karel’s job is to fill the hole with a beeper and proceed to the next corner. The following diagram illustrates how the world should look after the program execution. If you are limited to the four predefined commands, the main() function to solve this problem would look like this:

```
def main():
    move()
    turn_left()
    turn_left()
    turn_left()
    move()
    put_beeper()
    turn_left()
    turn_left()
    move()
    turn_left()
    turn_left()
    turn_left()
    move()
```

The initial motivation for defining the turn_right() function was that it was cumbersome to keep repeating three turn_left() commands to accomplish a right turn. Defining new functions has another important purpose beyond allowing you to avoid repeating the same command sequences every time you want to perform a particular task. The power to define functions unlocks the most important strategy in programming—the process of breaking a large problem down into smaller pieces that are easier to solve. The process of breaking a program down into smaller pieces is called decomposition, and the component parts of a large problem are called subproblems. As an example, the problem of filling the hole in the roadway can be decomposed into the following subproblems: Move up to the hole Fill the hole by dropping a beeper into it Move on to the next corner If you think about the problem in this way, you can use function definitions to create a program that reflects your conception of the program structure. The main function would look like this:

```
def main():
    move()
    fill_pothole()
    move()
```

The correspondence with the outline is immediately clear, and everything would be great if only you could get Karel to understand what you mean by fill_pothole(). Given the power to define functions, implementing fill_pothole() is extremely simple. All you have to do is define a fill_pothole() function whose body consists of the commands you have already written to do the job, like this:

```
def fill_pothole():
    turn_right()
    move()
    put_beeper()
    turn_around()
    move()
    turn_right()
```

Here is the complete program. Notice how you can understand the programmer’s intent simply from reading the main function. When you run the program, the line highlighting shows how a computer will execute it, step by step. However, because the program is nicely broken down we can understand it on a human thought level:

```
from karel.stanfordkarel import *

def main():
    move()
    fill_pothole()
    move()
```

Fills the pothole beneath Karel’s current position by

```
# placing a beeper on that corner. For this function to work
# correctly, Karel must be facing east immediately above the
# pothole. When execution is complete, Karel will have
# returned to the same square and will again be facing east.
```

```
def fill_pothole():
    turn_right()
    move()
    put_beeper()
    turn_around()
    move()
    turn_right()
```

```
# Turns Karel 90 degrees to the right.
```

```
def turn_right():
    turn_left()
    turn_left()
    turn_left()
```

```
# Turns Karel around 180 degrees.
```

```
def turn_around():
    turn_left()
    turn_left(){"width":5,"height":4,"initialX":0,"initialY":1,"initialID":1,"initialB":1000,"cells":[{"x":0,"y":0,"wall":{"0":
true,"1":true,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":1,"y":0,"wall":{"0":false,"1":true,"2":false,
"3":true},"beepers":0,"colour":"BLANK"},{"x":2,"y":0,"wall":{"0":true,"1":false,"2":false,"3":true},"beepers":0,"
colour":"BLANK"},{"x":3,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x"
:4,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":0,"y":...
```

Slide ID: 361900 | **Index:** 5 | **Title:** 5. For Loops | **Status:** unseen

One of the things that computers are especially good at is repetition. How can we convince Karel to execute a block of code multiple times? To see how repetition can be used, consider the task of placing 42 beepers:Basic For LoopSince you know that there are exactly 42 beepers to place, the control statement that you need is a for loop, which specifies that you want to repeat some operation a fixed number of times. The structure of the for statement appears complicated primarily because it is actually much more powerful than anything Karel needs. The only version of the for syntax that Karel uses is:for i in range(count):

statements to be repeated

We will go over all the details of the for loop later in the class. For now you should read this line as a way to express, "repeat the statements in the function body count times." We can use this new for loop to place 42 beepers by replacing count with 42 and putting the command put_beeper() inside of the for loop code block. We call commands in the code block the body:# Program: PlaceManyBeepers

```
# -----
```

```
# Places 42 beepers using a for loop
```

```
from karel.stanfordkarel import *
```

```
def main():
    move()
    # Repeat put_beeper many times
    for i in range(42):
        put_beeper()
    move(){"width":4,"height":4,"initialX":0,"initialY":0,"initialID":1,"initialB":1000,"cells":[]}]The code above is
```

editable. Try to change it so that it places only 15 beepers. Matching Postconditions with Preconditions The previous example gives the impression that a for loop repeats a single line of code. However the body of the for loop (the statements that get repeated) can be multiple lines. Here is an example of a program that puts a beeper in each corner of a world:

Program: CornerBeepers

Places one beeper in each corner

```
from karel.stanfordkarel import *
```

```
def main():
```

```
# Repeat the body 4 times
```

```
for i in range(4):
```

```
    put_beeper()
```

```
    move()
```

```
    move()
```

```
    move()
```

```
    turn_left() {"width":4,"height":4,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]}
```

Pay very close attention to the way that the program flows through these control statements. The program runs through

the set of commands in the for loop body one at a time. It repeats the body four times. Perhaps the single

most complicated part of writing a loop is that you need the state of the world at the end of the loop (the

postcondition) to be a valid state of the world for the start of the loop (the precondition). In the above

example the assumptions match. Good times. At the start of the loop, Karel is always on a square with no

beepers facing the next empty corner. What if you deleted the turn_left() at the end of the loop? The

postcondition at the end of the first iteration would no longer satisfy the assumptions made about Karel

facing the next empty corner. The code is editable. Try deleting the turn_left() command to see what

happens! Nested Loops Technically the body of a for loop can contain any control flow code, even other

loops. Here is an example of a for loop that repeats a call to a function which also has a for loop. We call

this a "nested" loop. Try to read through the program, and understand what it does, before running it:

Program: CornerFiveBeepers

Places five beepers in each corner

```
from karel.stanfordkarel import *
```

```
def main():
```

```
# Repeat once for each corner
```

```
for i in range(4):
```

```
    put_five beepers()
```

```
    move_to_next_corner()
```

```
# Reposition karel to the next corner
```

```
def move_to_next_corner() :
```

```
    move()
```

```
    move()
```

```
    move()
```

```
    turn_left()
```

```
# Places 5 beepers using a for loop
```

```
def put_five beepers() :
```

```
    for i in range(5):
```

```
        put_beeper() {"width":4,"height":4,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]}
```

Based on work by Chris Piech and Eric Roberts at Stanford

University. <https://compedu.stanford.edu/karel-reader/docs/python/en/intro.html>

The technique of defining new functions, and defining for loops—as useful as they are—does not actually enable Karel to solve any new problems. Every time you run a program it always does exactly the same thing. Programs become a lot more useful when they can respond differently to different inputs. As an example. Let's say you wanted to write a program to have Karel move to a wall. But you don't simply want this program to work on one world with a fixed size. You would like to write a single program that could work on any world.

Program: MoveToWall

Uses a "while" loop to move Karel until it hits

a wall. Works on any sized world.

from karel.stanfordkarel import *

the program starts with main

def main():

call the move to wall function

move_to_wall()

this is a very useful function

def move_to_wall():

repeat the body while the condition holds

while front_is_clear():

move(){"width":7,"height":7,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]}]Try changing the world by clicking the numbers above the world. For any sized world, Karel will move until it hits a wall. Notice that this feat can not be accomplished using a for loop. That would require us to know the size of the world at the time of programming.

Basic While Loop

In Karel, a while loop is used to repeat a body of code as-long-as a given condition holds. The while loop has the following general form:

while test:
statements to be repeated

The control-flow of a while loop is as follows. When the program hits a while loop it starts repeating a process where it first checks if the test passes, and if so runs the code in the body. When the program checks if the test passes, it decides if the test is true for the current state of the world. If so, the loop will run the code in the body. If the test fails, the loop is over and the program moves on. When the program runs the body of the loop, the program executes the lines in the body one at a time. When the program arrives at the end of the while loop, it jumps back to the top of the loop. It then rechecks the test, continuing to loop if it passes. The program does not exit the loop until it gets to a check, and the test fails.

Karel has many test statements, and we will go over all of them in the next chapter. For now we are going to use a single test statement: `front_is_clear()` which is true if there is no wall directly in front of Karel.

Fencepost Bug

Let's modify our program above to make it more interesting. Instead of just moving to a wall, have Karel place a line of beepers, one in each square. Again we want this program to work for a world of any size.

Program: BeeperLineBug

Uses a while loop to place a line of beepers.

This program works for a world of any size.

However, because each world requires one fewer

moves than `put_beepers` it always misses a beeper.

from karel.stanfordkarel import *

def main():

Repeat until karel faces a wall

while front_is_clear():

Place a beeper on current square

put_beeper()

Move to the next square

`move()`{`"width":7,"height":7,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]`}That looks great. Except for one problem. On every world Karel doesn't place a beeper on the last square of the line (look closely). When Karel is on the last square, the program does not execute the body of the loop because the test no longer passes -- Karel is facing a wall. You might be tempted to try switching the order of the body so that Karel moves before placing a beeper. The code is editable so go try it!There is a deeper problem that no rearrangement of the body can solve. For the world with 7 columns, Karel needs to put 7 beepers, but should only move 6 times. Since the while loop executes both lines when a test passes, how can you get the program to execute one command one more time than the other?The bug in this program is an example of a programming problem called a fencepost e...

Slide ID: 361902 | **Index:** 7 | **Title:** 7. If Statements | **Status:** unseen

The final core programming control-flow construct to learn are conditional statements (if and if/else).
Basic ConditionalsAn if/else statement executes an "if" code-block if and only if the provided test is true for the state of the world at the time the program reaches the statement. Otherwise the program executes the "else" code-block.
if test:

if code-block

else:

else code-block

To get a sense of where conditional statements might come in handy, let's write a program that has Karel invert a line of beepers. If a square previously had a beeper, Karel should pick it up. If a square has no beeper, Karel should put one down.
Program: UpAndDown

```
from karel.stanfordkarel import *
```

```
def main():
```

```
while front_is_clear():
```

```
invert_beeper()
```

```
move()
```

```
# Prevent a fencepost bug
```

```
invert_beeper()
```

```
# Picks up a beeper if one is present
```

```
# Puts down a beeper otherwise
```

```
def invert_beeper():
```

```
# An if/else statement
```

```
if beepers_present():
```

```
pick_beeper()
```

```
else:
```

```
put_beeper(){"width":8,"height":8,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[{"x":1,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":3,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":4,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":6,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"}]}Note that an if statement does not need to have an else block -- in which case the statement operates like a while loop that only executes one time:
```

```
if test:
```

```
if code-block
```

ConditionsThat last example used a new condition. Here is a list of all of the conditions that Karel knows of:

TestOppositeWhat it checks

front_is_clear()

front_is_blocked()

Is there a wall in front of Karel?

beepers_present()

no_beepers_present()

Are there beepers on this corner?

left_is_clear()

left_is_blocked()

Is there a wall to Karel's left?

right_is_clear()

right_is_blocked()

Is there a wall to Karel's right?

beepers_in_bag()

no_beepers_in_bag()

Does Karel have any beepers in its bag?

facing_north()

not_facing_north()

Is Karel facing north?

facing_south()

not_facing_south()

Is Karel facing south?

facing_east()

not_facing_east()

Is Karel facing east?

facing_west()

not_facing_west()

Is Karel facing west?

Putting it all togetherCongrats! You now know all of the core programming control-flow blocks. While you learned them with Karel, methods, while loops, for loops, if/else statements work in the same way in almost all major languages, including Python. Now that you have the building blocks you can put them

together to build solutions to ever more complex problems. To a large extent, programming is the science of solving problems by computer. Because problems are often difficult, solutions—and the programs that implement those solutions—can be difficult as well. In order to make it easier for you to develop those solutions, you need to adopt a methodology and discipline that reduces the level of that complexity to a manageable scale. Based on work by Chris Piech and Eric Roberts at Stanford University. <https://compedu.stanford.edu/karel-reader/docs/python/en/intro.html>

Slide ID: 361903 | **Index:** 8 | **Title:** 8. Code | **Status:** unseen

Write and test any code here! `from karel.stanfordkarel import *`

```
def main():  
    # your code here...  
    move()
```


Lesson 7: Week 1 - Python Programming Fundamentals

Lesson ID: 53299 | **Created:** 2024-05-27T01:25:36.564438+10:00 | **First Viewed:** 2025-10-01T22:23:01.477696+10:00 | **Last Viewed Slide ID:** 361915 | **State:** active | **Status:** attempted | **Slide Count:** 14

Outline: You'll learn to write and run your first Python program

Slides

Slide ID: 361915 | **Index:** 4 | **Title:** Program, language, and Python | **Status:** completed

A computer program (or code) is a set of instructions for a computer to follow. Computer programming (or coding) is the activity of writing computer programs. When you write a program you must use a language that the computer can understand. There are many such languages - Java, Python, C, Ruby, JavaScript, PHP, and many, many more. Each language has its own advantages and disadvantages, and there is no "best" language that will be the most appropriate choice in all situations. While, for expediency, this course will focus on a single language (Python), many of the fundamental principles you will learn, such as program flow, data manipulation and code modularity, carry over to almost every other language. Like natural languages, such as English and French, programming languages have a specific grammar; this is known as the language's syntax. Unlike natural languages, programming languages require you to be precise and unambiguous. Python has very simple syntax and is considered one of the best designed programming languages in this respect. In this course, you will learn the Python language. Python is one of the most popular languages for several reasons: It is a general-purpose language that can be used in a wide variety of situations. It is mature enough that many comprehensive libraries for many disciplines have been built and extensively tested. It was developed recently enough to be built upon many of the fundamental programming principles that have been developed and fine-tuned over the years. It has a human-friendly syntax that is easy to learn. Python has become one of the most popular languages for working with data. It has data processing and visualisation libraries such as pandas and Matplotlib that make handling data very easy. Even though you will be learning just one language, many of the fundamental principles you will learn, such as controlling program flow, manipulating data, and modularising code, carry over to almost every other language. Like natural languages, programming languages tend to change over time. You'll be using the latest version of Python throughout this course - Python 3. You may encounter some Python 2 code from time to time. The language differences between these two are minor however they can be incompatible with each other. In this course, code will usually be presented in a web interpreter as seen below, the language the code is written in can be seen in the top right corner of the web interpreter, most code will be editable and runnable. Try running the code below to see the exact version of Python 3 we are using (in ed):

```
import sys
print(sys.version)
```


Python nuts and bolts
Python is an interpreted language which means that code is passed directly to an interpreter (a program running on the host machine) for execution rather than being compiled into machine-readable instructions to be executed directly by the computer. This means that executing Python code can be as simple as starting an interpreter and typing commands to run. If you installed a version of Python on your computer then there will be an interpreter as part of that installation. Alternatively, you can access an interpreter through the Workspaces and code snippets here on ed platform. Increasing usability by simplifying the process from program writing to execution makes Python a very attractive language for the novice programmer.

Slide ID: 361916 | **Index:** 10 | **Title:** Python Programming Basics | **Status:** unseen

Here is a very simple Python program: `print('Hello, world!')`
Execution
You can execute the program (i.e., run it) by clicking 'Run'. When you do, the Python interpreter executes each line of the program, one at a time, from beginning to end (in this case there is only one line). When you run a program like this, the Python interpreter executes each line of the program, one at a time, from beginning to end (in this case

there is only one line). Whenever you see a runnable piece of code like this you can also modify it yourself. This is a great way for you to experiment with Python (don't worry, you won't break anything, and everything gets restored when you refresh the page). Try modifying the code above to get it to print your name rather than "world".

Statements Each line in the program is called a statement. Your programs will typically contain many statements. Here is one with two statements:

```
print('Hello, world!')
print('Goodbye, world!')
```

The `print()` function `print()` is a Python function, one that you will find yourself using often. 'Hello world!' is an argument of the function - some input that we give to the function. The `print()` function performs the task of printing the argument.

Try modifying the program below to make it print your name (you need to use quotation marks):

```
print()
```

If you supply `print()` with multiple arguments it will print them all, separated by spaces. You will find this very useful: `print('The value is', 10*2)`

A very handy technique when using `print()` is to use an f-string. If you append an 'f' to the start of a quoted string (before the first quote) you can get Python to do things inside the string before it prints, such as perform calculations. You do this by using curly brackets. Here are some examples:

```
print(f'The result is {10*2}.')
print(f'The results are {10*2}, {10*3}, and {10*4}.')
```

Input Getting user input Another Python function that you will use often is `input()`, which you can use to get input from the user. For example:

```
print(input('What would you like to print? '))
```

This program asks the user for some input and then prints it.

Syntax errors Before executing a program the interpreter first checks that it has correct syntax. If it finds any syntax errors it will tell you, and not run the program. The following program contains a syntax error (it's missing a quotation mark) - try running it:

```
print('Hello, world!)
```

Runtime errors Sometimes the syntax of a program is fine, but Python raises an error during its execution. These are called runtime errors. When this happens, execution stops, and Python returns some information about the error. For example, the following program asks Python to divide a number by zero, which is impossible. It generates a runtime error:

```
print(10/0)
```

Logical errors Sometimes the syntax of a program is fine, and it executes without any runtime errors, but it doesn't do what you intended it to do. This is called a logical error. For example, suppose you write the following program to add 1 and 2 and then multiply the result by 3:

```
print(1 + 2 * 3)
```

The result you're expecting is 9, because 1 + 2 is 3, and 3 times 3 is 9. But the program above prints 7. Error! This is not a syntax error, nor a runtime error - it's a logical error. The problem is that Python assumes that you want the multiplication to happen first, then the addition (you will learn more about this). So it calculates $2 \times 3 = 6$, then $1 + 6 = 7$. You should write your program like this instead:

```
print((1 + 2) * 3)
```

This removes the logical error.

Comments In Python, anything on the same line after a # will be ignored by the Python interpreter. This allows you to add comments. It is important that the programs you write are easily understandable by someone who reads them. To help with this it can be a good idea to add comments throughout. Here is an example from above, with an explanatory comment added:

```
# Ask the user for input and then print it:
print(input('What would you like to print? '))
```

Comments can start anywhere on a line (but use comments on t...

Slide ID: 361917 | Index: 13 | Title: Importing modules | Status: unseen

You won't get far using Python before you need to import a module. A module is just a file that contains Python code, with a ".py" extension. Although the core Python language contains a lot of functionality, some of the functionality you'll want is included in modules, rather than in the core. You can access that functionality by importing those modules.

For example, there is a lot of non-core mathematics functionality in the "math.py" module. You can import that module as follows (note that you don't include ".py" - Python will figure it out):

```
import math
```

This makes all of the code that has been saved in the math.py file available for use in your program. For example, in math.py there is a function `floor()`, which rounds a number down to the nearest integer. Having imported math.py you can now call this function:

```
import math
print(math.floor(3.14))
```

Note that you call the function using `math.floor()`, rather than using just `floor()`. If you'd like to be able to use this function without having to add the prefix, you can do so using the `from` keyword when you import:

```
from math import floor
print(floor(3.14))
```

If you'd like to import more functions then you can just separate their names by commas:

```
from math import floor, ceil
print(floor(3.14))
print(ceil(3.14))
```

You can import everything by using an asterisk:

```
from math import *
```

```
print(floor(3.14))
print(ceil(3.14))
```

But you should avoid doing this because you might end up with the same function name being used twice as shown in the examples below:

```
from math import *

def ceil(x): # my ceil() doubles the number given
    return 2*x
```

```
print(ceil(3.14)) # you will get 6.28
```

```
def ceil(x): # my ceil() doubles the number given
    # return 2*x
```

```
print(ceil(3.14)) # you will get 4
```

```
import math

def ceil(x): # my ceil() doubles the number given
    return 2*x
```

```
print(math.ceil(3.14)) # you will get 4
```

```
print(ceil(3.14)) # you will get 6.28
```

You can use the `dir()` function to see what is available to you in a module that you have imported:

```
import math
print(dir(math))
```

When you import a module you can give it an alias using the `as` keyword, to save yourself some typing. One module that we will be using a lot in the second half of this course is the `pandas` module. It is standard to give it the alias `"pd"` when importing it:

```
import pandas as pd
print(dir(pd))
```

When Python gets installed on a computer many modules are installed with it. These are called built-in modules. These are automatically available for you to import. If you want to import other modules they will first need to be installed on the computer. You can also write your own modules, and import them.

Slide ID: 361918 | **Index:** 14 | **Title:** Objects | **Status:** unseen

When you program with Python (and many other languages) you will work a lot with objects. Objects encapsulate a piece of data (which may be simple, such as a single number; or more complex, such as a collection of smaller objects) and are the building blocks of Python. For example, in the statement `print('The value is', 10*2)`, `'The value is'`, `10` and `2` are all objects. Indeed, even the `print()` function is an object. Types

Every object is of a certain type. The type of an object determines how Python interacts with it, for example it makes sense to add two numbers, but it does not make sense to add functions. In Python, the main types are:

- Integer (`int`). A whole number, positive or negative, including zero (i.e. ..., -2, -1, 0, 1, 2, ...).
- Floating-point number (`float`). A positive or negative number, not necessarily whole, including zero (e.g. 3.14, -0.12, 89.56473).
- String (`str`). A sequence of characters (e.g. `'Hello'`, `'we34t&2*'`). Used to store text.
- Boolean (`bool`). A truth value, either true or false.
- List (`list`). An ordered container of objects.
- Tuple (`tuple`). An immutable list (i.e. one that cannot be changed).
- Set (`set`). An unordered container of unique objects.
- Dictionary (`dict`). A set of key-object pairs.
- Function. A piece of code that can be run by calling it.
- Class. A user-defined type of object.

Python also has a special object, `None`, which represents the absence of an object. It is of type `NoneType`. It is the only object of this type.

You will be learning more about the objects of each type. This week you will learn about integers, floats, strings, and booleans. Next week you will learn about lists, tuples, sets, and dictionaries. In Week 3 you will learn about functions and classes.

Checking the type of an object

You can find out the type of an object by using Python's `type()` function:

```
print(type(1))
```

```
print(type(3.14))
```

```
print(type('Hello'))
```

```
print(type(None))
```

```
print(type(print))
```

Notice in the last example above that we have provided the `print` function as an argument

to the type function. You can do that - the print function is an object, just like numbers and strings are, and you can provide it as an argument to the type() function to find out what type of object it is. Most of the time you'll be calling print(), and providing arguments to it, but occasionally you might provide it as an argument to some other function, as we have done above. In fact, you can even provide print as an argument to print() function itself: print(print). When an object is of a certain type we say that it is an instance of that type. You can also use Python's isinstance() function to check whether an object is of a certain

```
type: print(isinstance(1, int)) # True
```

```
print(isinstance(1, float)) # False
```

```
print(isinstance(3.14, float)) # True
```

```
print(isinstance(3.14, int)) # False
```

```
print(isinstance('Hello', str)) # True
```

```
print(isinstance('Hello', float)) # False
```

Attributes Objects have attributes. Attributes are properties that are specific to the object. For example, the string object 'Hello', has an attribute upper(), which is a function that produces an upper-case version of the string. You can access this attribute of the string by using dot notation: print('Hello'.upper())

It can be useful to think of '.' as representing " 's " (i.e. apostrophe-s) - the

statement 'Hello'.upper() is instructing Python to execute the upper() function of the 'Hello' object. When an attribute is a function, like this one is, it is also called a method. Attributes which are not methods are also

known as fields. Which attributes an object has depends upon what type of object it is. String objects have

the upper() attribute (method), but integer objects do not. If you try to access this attribute of an integer you will get an error: print("Hello".upper())

```
number = 12
```

```
print(number.upper())
```

Part of learning Python, and other languages, is learning what attributes the different types of objects have.

Slide ID: 361919 | **Index:** 16 | **Title:** Expressions | **Status:** unseen

To work with an object you need to refer to it, and to refer to it you use an expression. Expressions come in two varieties: simple expressions and complex expressions. We'll consider simple expressions first, and then complex expressions. There are three types of simple

expressions: Literals Variables Constants

Literals A literal shows explicitly which object they refer to. Here are

some examples: Integer literals: 1, 26, -14 Floating-point literals: 3.14, 0.06, -9.7 String literals: 'Hello',

"Goodbye" (you can use single or double quotes, but they must match) Boolean literals: True, False (there

are only two) None literal: None (there is only one) You might be wondering whether -14 counts as a literal.

The answer is no: Python understands this expression not as a literal for the number -14, but as the

application of the - operator to the number 14 (more about number operators in the next

slide). Variables You can also introduce your own names for objects, and these are called variables. When

you introduce a variable you have to specify which object it refers to. This is called assigning the variable,

and you do it using the assignment operator =. This is sometimes also called binding the variable - you

bind it to an object. Here is an example: message = 'Hello there'

```
print(message)
```

The first line introduces a variable message and assigns it the string object 'Hello there'.

The second line uses this variable to print the object. Once you introduce a variable you can use it as many

```
times as you like throughout your program: message = 'Hello there'
```

```
print(message)
```

```
print(message)
```

```
print(message)
```

```
print(message)
```

The object that the variable refers to is often called the value of the variable. You can

change the value of a variable (i.e., change which object the variable refers to) as often as you

like: message = 'Hello there'

```
print(message)
```

```
message = 3.14 # Assign a new value
```

```
print(message)
```

```
message = True # Assign a new value
```

```
print(message)
```

Notice that the variable message in the program refers to different types of object as the

program proceeds - first it refers to a string, then to a floating-point number, and then to a boolean.

Because variables in Python can do this we say that Python has flexible typing. (Some languages, such as C and Java, do not allow variables to change their reference to a different type of object.) If you want to check the type of the object that a variable refers to, you can use the `type()` function:

```
x = 'Hello there'
print(type(x)) # str
```

```
x = 3.14
```

```
print(type(x)) # float
```

```
x = True
```

```
print(type(x)) # bool
```

```
x = print
```

```
print(type(x)) # function
```

Any type of object can be assigned to a variable, not just numbers, strings or booleans. Look at the last example above - we have assigned the `print` function to the variable `x`. Having done that, `x` is then another name for the `print` function, and we can use it accordingly:

```
x('Hello, world!')
```

You can use variables to assign other variables:

```
message = 'Hello there'
new_message = message
```

```
print(new_message)
```

Be careful to understand what's going on here. In line 2 you are getting `new_message` to refer to whatever it is that `message` refers to at the time. If a different object is later assigned to `message`, it will not automatically be assigned to `new_message` as well - there will be no change in what `new_message` refers to. Let's check that:

```
message = 'Hello there'
new_message = message
```

```
message = 'Goodbye' # Value of new_message not changed
```

```
print(new_message)
```

You must assign a value to a variable before you use it, otherwise Python will generate an error:

```
print(message) # Error - message has no value
```

If you want to introduce a variable but don't yet have any significant value to assign it then you can assign it the value `None`:

```
message = None
```

```
print(message) # No error - message has a value
```

You can assign multiple variables the same value:

```
x = y = z = 0 # All get assigned 0
```

```
print(x, y, z)
```

Or different values, using a technique called multiple assignment:

```
x, y, z = 1, 2, 3 # x gets 1, y gets 2, z gets 3
```

```
print(x...
```

Slide ID: 361920 | **Index:** 88 | **Title:** Working with numbers | **Status:** unseen

Python has 3 built-in types for numbers: `int` - representing whole numbers (positive and negative integers) with unlimited precision (i.e. there is no a priori maximum or minimum value an `int` object can have)

`float` - representing floating point numbers (non-whole numbers) with a limited precision (i.e. a `float` object is limited to a certain number of significant figures)

`complex` - representing complex numbers. A complex number is a number with two distinct components: a real part and an imaginary part.

While numeric types are largely interchangeable (for example you can add an `int` to a `float`), a common source of runtime errors is when objects of the wrong type are being used (for example using a `float` when an `int` was expected or vice versa).

Operating on numbers You can add, subtract, multiply, and divide numbers by using the operators `+`, `-`, `*`, `/`, respectively, and combinations of them:

```
print(1 + 2)
```

```
print(10 - 5)
```

```
print(3 * 4)
```

```
print(20/4)
```

```
print((1 + 2)*(3 + 4))
```

```
print(10/(3-1))
```

Complex numbers examples:

```
n = 1 + 2j
print(n)
```

```
print(n.real)
```

```
print(n.imag)
```

```
a = 1 + 2j
```

```
b = 3 - 4j
```

```
print(a + b)
```

```
print(a - b)
```

```

print(a * b)
print(a / b)

```

You can raise one number to the power of another number by using the exponentiation operator, `**`: `print(10 ** 2)` # 10 to the power 2

`print(2 ** 3)` # 2 to the power 3

You can divide one number by another number and round down to the nearest whole number by using the integer division operator, `//`: `print(10 // 3)` # 10 divided by 3 and rounded down

Note the effect of using `//` on negative numbers: `print(-10 // 3)` # -10 divided by 3 and rounded down

You can divide one number by another number and get the remainder by using the modulus operator, `%`: `print(10 % 3)` # The remainder when 10 is divided by 3

Order of operations Unless you specify the order in which operations are to be performed, by using brackets, Python performs them in a very particular order: First, `**` is performed, from right to left. Next, `*`, `/`, `//`, and `%` are performed, from left to right. Next, `+` and `-` are performed, from left to right. `print(1 + 2 * 3)` # Same as `1 + (2 * 3)`

`print(2 ** 3 * 4)` # Same as `(2 ** 3) * 4`

`print(24 / 6 * 2)` # Same as `(24 / 6) * 2`

`print(24 / 6 / 2)` # Same as `(24 / 6) / 2`

`print(2 ** 2 ** 3)` # Same as `2 ** (2 ** 3)`

Augmented assignment operators You will often find yourself wanting to operate on a variable and then re-assign the result to that same variable. Python provides augmented assignment operators to allow for more concise code: `+=`, `-=`, `*=`, `/=`, `//=`, and `%=`. `x += 2` is equivalent to `x = x + 2`. `x -= 2` is equivalent to `x = x - 2`. `x *= 2` is equivalent to `x = x * 2`. `x /= 2` is equivalent to `x = x / 2`. `x //= 2` is equivalent to `x = x // 2`. `x %= 2` is equivalent to `x = x % 2`. `x = 12`

`x += 2` # Equivalent to `x = x + 2` (assign to x the result of x + 2)

`print(x)`

Comparing numbers Python has a number of comparison operators which you can use to compare numbers: `num1 == num2` is True if the value of num1 equals the value of num2, otherwise it is False. `num1 is num2` is True if the value of num1 equals the value of num2, and num1 and num2 have the same type, otherwise it is False. `num1 != num2` is True if the value of num1 does not equal the value of num2, otherwise it is False. `num1 is not num2` is True if the value of num1 does not equal the value of num2, or num1 and num2 have different types, otherwise it is False. `num1 is True` if the value of num1 is less than the value of num2, otherwise it is False. `num1 is True` if the value of num1 is less than or equal to the value of num2, otherwise it is False. `num1 > num2` is True if the value of num1 is greater than the value of num2, otherwise it is False. `num1 >= num2` is True if the value of num1 is greater than or equal to the value of num2, otherwise it is False. `x = 1`

`y = 1.0`

`print(1 == 1.0)`

`print(x is y)`

Floating-point numbers and `==` Because floating point numbers are stored with limited precision you might experience strange results when attempting to compare them with `==`. For example: `print(0.1 + 0.2 == 0.3)` Why does this comparison return False? Because 0.1...

Slide ID: 361921 | Index: 89 | Title: Working with strings | Status: unseen

A string is a sequence of characters. The characters might be ones you can type on your keyboard, or any of the hundreds of thousands of other unicode characters, including symbols and foreign language letters.

String literals As you have seen, string literals have the characters between quotes, either single quotes or double quotes. `literal_one = 'A string'` # You can use single quotes

`literal_two = "A string"` # Or double quotes

`print(literal_one)`

`print(literal_two)`

The empty string One of the most useful strings is the empty string - a string with no characters. The literal for an empty string is `"`, or `""`. Note that the empty string is not the same thing as `None` - they are different objects: `empty_string = ""`

`print(empty_string)`

`print(None)`

`print(empty_string is None)`

Escaping special characters Sometimes you will need to use a string literal that contains quote marks or other characters that have special meaning in Python. You can do this by escaping those special characters, which means prefixing them with a backslash `\`. # Backslash used to

```
escape a quote mark
print('Penny\'s dog')
print("Penny's dog is called \"Mac\".")
```

```
# Backslash used to escape a new line symbol
print('This is one line.\nThis is a second line')
```

```
# Backslash used to escape a tab symbol
print('Here\tThere')This means that the backslash itself has a special meaning in strings, so to include it
you must escape it:print("You can have tea neither\nor coffee")
print("You can have tea neither\\nor coffee")If you have a lot of backslashes and no special characters
then you can tell Python to ignore the special meaning of the backslash by preceding the string with r (for
"raw"):print(r"You can have tea neither\nor coffee")You can avoid having to escape single quote marks by
enclosing the whole string in double quotes. Similarly, you can avoid having to escape double quote marks
by enclosing the whole string in single quotes:print("Penny's dog")
print('Penny said, "This is my dog".')If you like to include line breaks in a string then you can either use \n,
as in the example above, or you can use triple quotes around the string:# Using \n
text = 'This is one line.\nThis is a second line'
print(text)
```

```
# Using triple single quotes
text = """This is one line.
This is a second line."""
print(text)
```

```
# Using triple double quotes
text = """This is one line.
This is a second line."""
print(text)
```

```
text = """\
This is one line.
This is a second line."""
print(text)You can also use triple quotes around a string that contains both single and double quotes:text =
"""She said, "I don't know how you do it!"""
print(text)
text = """She said, 'I don't know what "" means'"""
print(text)You'll get an error if you end up with four quotes in a row.Concatenating stringsYou can
concatenate strings (i.e. join them) using the + operator:first_name = 'Leo'
last_name = 'Tolstoy'
full_name = first_name + ' ' + last_name
print(full_name) You can also use the += augmented assignment operator:name = 'Leo'
name += ' '
name += 'Tolstoy'
print(name)You can duplicate a string a given number of times by using the * operator:print('a' *
10)Comparing stringsJust as with numbers, you can use Python's comparison operators to compare
strings:str1 == str2 is True if str1 and str2 are the same sequence of characters, otherwise it is Falsestr1 !=
str2 is True if str1 and str2 are not the same sequence of characters, otherwise it is Falsestr1 in str2 is
True if str1 appears as a substring in str2, otherwise it is Falsestr1 not in str2 is True if str1 does not
appear as a substring in str2, otherwise it is Falsestr1 is True if str1 is lexicographically less than (i.e.
would appear earlier in the dictionary than) str2, otherwise it is FalseSimilarly we have str1 , str1 > str2,
and str1 >= str2print('A' == 'A')
print('A' == "A") # Whether you define literals with ' or " does not matterprint('fish' in 'selfishness')
```

```
print('fine' in 'selfishness') # Substrings have to be contiguous
print('A' Applying functions to stringsY...
```

Slide ID: 361922 | **Index:** 90 | **Title:** Working with booleans | **Status:** unseen

Boolean objects are the simplest type of objects in Python, but often they play the most significant role in determining the path that a program takes (as you will see). There are two boolean objects: True and False. They most commonly appear as the result of the comparison operations we have seen for the other types, for example 3 will evaluate to True; and 1 + 1 == 3 will evaluate to False:

```
print(3 > 2) # True
print(1 + 1 == 3) # False
```

You can build complex boolean expressions by combining simple boolean expressions with the logical operators not, and, and or:

- not x is true if x is false, otherwise it is false.
- x and y is true if x is true and y is true, otherwise it is false.
- x or y is true if x is true or y is true or both, otherwise it is false.

```
print(not True) # False
```

```
print(not False) # True
```

```
print(not 3 > 2) # True
```

```
print(True and False) # False
```

```
print(False and True) # False
```

```
print(False and False) # False
```

```
print((3 > 2) and True) # True
```

```
print(True or False) # True
```

```
print(False or True) # True
```

```
print(False or False) # False
```

```
print((3 > 2) and guess == 42) # True
```

```
print("Your number is between 2 and 5?")
```

Chaining comparisonsIt is possible to combine comparisons into a chain:

```
x = 12
2 < x < 5
```

Rather than this

Short circuitingThe operators or and and are said to be short circuiting operators, because they only evaluate their second expression if they need to. To illustrate, the program below does not generate an error, even though the variable var has not been defined. That's because Python doesn't even look at that variable. It doesn't need to, because the first True is enough to make the whole expression true.

```
print(True or var)
```

Similarly, the program below does not generate an error, even though the variable var has not been defined. That's because Python doesn't need to look at it - the first False is enough to make the whole expression false.

```
print(False and var)
```

If you swap the order in either case, you get an error:

```
print(var and False)
```

Slide ID: 361923 | **Index:** 91 | **Title:** Controlling program flow | **Status:** unseen

In a simple program, Python executes the statements of the program one-by-one in sequence, from start to finish. However, you will often want your program to deviate from this:

- Conditional execution. You might want to execute a certain statement only if a certain condition is true.
- Loops. You might want to loop through a block of code multiple times.
- Handling exceptions. You might want to run a block of code in a "cautious" way, so that if an error arises you can deal with it without your whole program stopping. We will look at these one-by-one in the next few slides.

Slide ID: 361924 | **Index:** 92 | **Title:** If statements | **Status:** unseen

You will often want Python to execute a certain statement only if a certain condition is true. For this you can use an if statement. The following program uses an if statement:

```
number = int(input('What is your favourite number? '))
```

```
if number == 42:
```

```
    print('That is my favourite number too!')
```

```
print('Good bye')
```

If the user enters the number 42 then Python executes line 3, otherwise it skips line 3 and goes straight to line 4. SyntaxThe syntax for an if statement is:

```
if condition:
    # code to execute if condition is true
```


There are two parts to this statement - the part between `if` and `:` is called the header of the statement; the rest is called the body of the statement. The body of the statement is a block of statements. Notice that the body contains one or more statements (as many as you like), so an `if` statement contains other statements as part of it. Because of this we call it a compound statement. Also notice that the body is indented. This is required - if you don't use indentation then Python will issue an error: `if True:`

`print('Hello')` You can use either the tab character or the space character to create the indentation, but you must use the same character for each line, and the same number of those characters, otherwise Python will issue an error. Each of the following will cause an error: `if True:`


```
print('Hello') # Indentation using a tab
```

```
print('Hello') # Indentation using 4 spaces - Error
```

`if True:`

```
print('Hello') # Indentation using 4 spaces
```

```
print('Hello') # Indentation using 5 spaces - Error
```

It is standard to use 4 spaces for indentation. You can set this to be the default behaviour in the code editor for `ed` by selecting "Soft tabs" under the settings menu (icon: ) in the top right of an editor window. The statement block after `if` must contain at least one statement. It is common use the `pass` statement as a placeholder for unfinished code - it is a statement that does nothing: `if True:`

```
pass
```

There must be at least one statement in the body. Conditions Any boolean expression can be used between the `if` and the `:` `if True:`

```
print('The condition is true')
```

```
if 2 > 1:
```

```
print('The condition is true')
```

```
if 2 > 1 and 2 <= 2 or 2 > 1:
```

```
print('The condition is true')
```

```
if 'cat' == 'cat':
```

```
print('The condition is true')
```

Be careful with variables! If you introduce a variable inside the block of an `if` statement then that variable will only be defined if the block is executed. This might cause you some unexpected errors. For example: `if False:`

```
x = 1
```

```
print(x)
```

Error - `x` does not have a value. Since line 2 is not executed the variable `x` does not get assigned a value, so when it is used in line 3 Python issues an error. Else clauses You can add an `else` clause to an `if` statement, to tell Python what to do if the condition of the `if` statement is false: `number = int(input('What is your favourite number? '))`

```
if number == 42:
```

```
print('That is my favourite number!')
```

```
else:
```

```
print('That is not my favourite number.')
```

Grade example with multiple conditions # we assume the user gives a valid mark between 0 and 100 inclusive

```
mark = float(input("Please give your mark: "))
```

```
if mark == 50
```

```
if mark == 65
```

```
if mark == 75
```

```
if mark == 85
```

```
print('HD')
```

Elif clauses You can add `elif` clauses (short for 'else if') to chain together multiple conditions: `number = int(input('What is your favourite number? '))`

```
if number == 42:
```

```
print('That is my favourite number!')
```

```
elif number == 21:
```

```
print('That is my second favourite number')
```

```
else:
```

```
print('That is neither of my favourite numbers.')
```

Grade example with `elif` # we assume the user gives a valid mark between 0 and 100 inclusive

```
mark = float(input("Please give your mark: "))
```

```

if mark = 50
print("PS")
elif mark = 65
print("CR")
elif mark =75
print("DN")
else: # mark is >=85
print('HD')
Abbreviations
If you only have one statement in an if body then you can put it on the same line as the header. The same applies to elif and else. Note that you still need the colon:
number = int(input('What is your favourite number? '))
if number == 42: print('That is my favourite number too!')
elif number == 21: prin...

```

Slide ID: 361925 | **Index:** 93 | **Title:** While statements | **Status:** unseen

Sometimes you might want to repeat a set of statements for as long as a certain condition is true. For this you can use a while statement. Here's a program that uses a while statement to print the first 10 positive integers:

```
n = 1
```

while n When Python gets to line 2 it evaluates the condition after while. If the condition is true then it executes the statement block below, in lines 3-4, and then returns to line 2 again. If the condition is false then it skips the block and goes straight to line 5. You could achieve the same effect by using 10 different print statements, but using a while statement is more elegant and less repetitive. And if you don't know in advance how many integers to print, for example if you want to ask the user, then it might be impossible to use just print statements. The while block can contain any statement(s) you like, including if statements and other while statements. For example, here's a program that prints the even numbers between 1 and 10. It uses an if statement inside the while loop:

```
n = 1
```

while n Here is an example of a while loop used to iterate through a string, counting the number of times 'e' occurs in it.

```
string = 'The quick brown fox jumped over the lazy dog'
occurrences = 0
i = 0
```

while i Continuing You can use a continue statement to skip to the next iteration of a loop:

```
i = 0
```

while i When the value of i gets to 5 the continue statement is executed, and Python jumps directly back to line 2 and continues. The number 5 does not get printed, but 6 - 10 do. Why might you use continue? It can help to keep your code from getting to many levels of indentation. We will see examples of this.

Breaking You can use a break statement to break out of a loop entirely:

```
i = 0
```

while i When the value of i gets to 5 the break statement is executed, and Python jumps directly to line 7. The number 5 does not get printed, and nor do 6 - 10. Note that if the loop is nested inside another loop, the break statement terminates only the inner loop.

Keeping a program running When you run the program below it stops after it gets and prints a name. To run it again you have to click 'Run' again:

```
name = input('What is your name? ')
```

```
print('Hello', name)
It can be convenient to have the program keep running - getting it to start again automatically after it does its thing. You can get it to do this by adding a while loop, with a condition that always evaluates to true:
while True:
name = input('What is your name? ')
print('Hello', name)
Now the program will keep running, until you click 'Stop'. If you want to get a bit fancier, you could get your program to stop when the user enters a certain value, such as 'q'. Remember to let the user know that they can do this:
while True:
name = input('What is your name? (Enter q to quit) ')
if name == 'q':
break
```

```
print('Hello', name)
Nested while loops example
# Write a program to display the following:
# * * * * *
```

```
# * * * * *  
# * * * * *
```

```
i = 1  
while i
```

Slide ID: 361926 | **Index:** 94 | **Title:** Handling exceptions | **Status:** unseen

If your code interacts with the outside world, you may encounter unexpected circumstances. Perhaps a file that you are trying to open doesn't exist, or when you try to save the user's data you find that the disk is full, or perhaps the user enters a non integer value when you are expecting an integer, or maybe you divided by zero. Exceptions are a mechanism for dealing with the unwanted or unexpected situations. The approach in Python is known as structured exception handling. This means that if Python encounters an exception in a block of code, it will search for an exception handler enclosing that block. If none exists, it will look for an exception handler enclosing that outer block, and so on. It is important to understand that exception handling should be used to handle exceptions that are recoverable. If your program cannot handle the exception or recover from it, it should not even try. Instead it should let the exception terminate your program so the user can deal with it instead. For many cases of exceptions, there is no remedy. Try and except You can handle exceptions by using a try statement. Consider the following program, which asks the user to enter a number and then returns the square of that number:

```
num = input("Please enter a number: ")  
result = float(num) ** 2  
print("The square of the number is", result)
```

If the user enters a string of letters instead of a number then an error occurs and the program very ungracefully ends. Python complained with a `ValueError`. To handle this, we'd write a `try ...except ...` block around the code that could potentially raise an exception.

```
num = input("Please enter a number: ")  
try:
```

```
    result = float(num) ** 2  
    print("The square of the number is", result)  
except:
```

```
    print("You did not enter a number.")
```

In the above code, when control reaches line 2, an exception is raised. Python finds the nearest exception handler and moves control to line 3, which proceeds into the block at line 4. If there was no exception on line 2, then the code in the except block would not be executed. If you don't want to do anything with the exception then you can use the pass statement:

```
num = input("Please enter a number: ")  
try:
```

```
    result = float(num) ** 2  
    print("The square of the number is", result)  
except:
```

```
    pass
```

When an exception occurs, there is information about the exception contained in a special `Exception` object. You can assign that object to a variable and then display that information to the user:

```
try:  
    int('string')  
except Exception as err:  
    print('Exception:', err)
```

Else if you have some code that you want to execute only if the try block succeeds then you can add an else clause:

```
x = 'Hello'
```

```
try:  
    x = int(x)  
except:  
    print('The conversion was not successful.')  
else:  
    print('The conversion was successful.')
```

So far we have seen programs that interact with the external environment via the `input()` and `print()` functions. Another way they can interact is by reading from and writing to files. This is very simple in Python, which is why Python is a popular tool for working with files. This is what you will typically want to do: Open the file, Read from the file, or write to the file, Close the file. Opening a file To work with a file you must first open it. You do so using the `open()` function: `f = open('MyData', 'w')` The function expects a file as the first argument. This can be given as an absolute or relative filename. If given as a relative filename, it is relative to the directory that Python was executed from - in Ed this is always the same directory as the program. You can also supply a mode as the second argument. This indicates whether the file is to be opened for reading (i.e. input) or writing (i.e. output). If you don't specify the mode, Python assumes that you want to open the file for reading. The available options for the mode are: 'r' - Open the file for reading. `open` will throw an exception if the file does not exist. 'w' - Open the file for writing. If the file exists, the contents are completely overwritten. If the file does not exist, it will be created. 'x' - Open the file for writing. If the file already exists this will throw an exception. If the file does not exist, it will be created. 'a' - Open the file for appending. The file is opened at the end and any writes to the file will append to the end. If the file does not exist it is created. This option is useful for adding information to a file - for example a log file. The `open()` function returns a file object (also called a file handle) that represents the file that you have opened. It is this object that allows you to perform operations on the underlying file itself. Reading from and writing to a file The `read()` method of a file object returns a string that contains the entire contents of the file. The `write()` method takes a string and adds it to the file. # Open a file for writing

```
file = open('myfile', 'w')
```

```
# Write to the file
```

```
# If you want a newline anywhere you have to add it, using \n
file.write('Line 1: Some text.\n')
```

```
# Write some more to the file
```

```
file.write('Line 2: Some more text.')
```

```
# Now open the file for reading
```

```
file = open('myfile', 'r')
```

```
# Print the contents
```

```
print(file.read())
```

Closing a file You can close a file by using its `close()` method. It is important to close a file after you are done working with it, to free up resources back to the system. A program can only have a limited number of files open while it is running. If a running program reaches this limit, it will receive a "Too many open files" error when it attempts to open more files. So the program above should look like this

```
instead: file = open('myfile', 'w')
```

```
file.write('Line 1: Some text.\n')
```

```
file.write('Line 2: Some more text.')
```

```
file.close()
```

```
file = open('myfile', 'r')
```

```
print(file.read())
```

```
file.close()
```

Using a `with` block It's a good idea to work with a file inside a `with` statement. This ensures that the file is always closed after use, even if an error occurs inside your program. As soon as control exits the `with` statement the file will be automatically closed. Here is the previous example written with `with` statements:

```
# Open a file for writing
```

```
# The file is automatically closed after the with statement
```

```
with open('myfile', 'w') as file:
```

```
file.write('Line 1: Some text.\n')
```

```
file.write('Line 2: Some more text.')
```

```
# Open the file again for reading
# The file is automatically closed after the with statement
with open('myfile', 'r') as file:
    print(file.read())
```

See Files Example in Ed Workspaces

Slide ID: 361928 | **Index:** 97 | **Title:** Further reading | **Status:** unseen

You might find the following helpful: The Python Tutorial at w3schools.com
Numbers in Python <https://realpython.com/python-numbers/>

Lesson 8: Week 1 - From Problem Description to Python Program Example

Lesson ID: 53301 | Created: 2024-05-27T01:25:36.720507+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 361930 | Index: 1 | Title: From Problem Description to Python Program Example | Status: unseen

Slide ID: 361931 | Index: 2 | Title: Python Program Example | Status: unseen

Lesson 9: Week 4 - Notes 3 The Towers of Hanoi

Lesson ID: 53313 | Created: 2024-05-27T01:25:38.498815+10:00 | State: active | Status: unattempted | Slide Count: 8

Slides

Slide ID: 361987 | Index: 1 | Title: The towers of Hanoi | Status: unseen

Slide ID: 361988 | Index: 2 | Title: Useful links about the towers of Hanoi | Status: unseen

Tower of Hanoi - Wikipediahttps://en.wikipedia.org/wiki/Tower_of_Hanoi
Tower of Hanoi - Game<https://www.mathsisfun.com/games/towerofhanoi.html>

Slide ID: 361989 | Index: 3 | Title: The towers of Hanoi | Status: unseen

Slide ID: 361990 | Index: 4 | Title: recursive_hanoi.py | Status: unseen

Slide ID: 361991 | Index: 5 | Title: iterative_hanoi.py | Status: unseen

Slide ID: 361992 | Index: 6 | Title: The towers of Hanoi I | Status: unseen

Slide ID: 361993 | Index: 7 | Title: The towers of Hanoi II | Status: unseen

Slide ID: 361994 | Index: 8 | Title: The towers of Hanoi III | Status: unseen

Lesson 10: Week 5 - Overview

Lesson ID: 53315 | Created: 2024-05-27T01:25:39.145437+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 361999 | Index: 9 | Title: Week 5 Tuesday To Do List | Status: unseen

Week 5 Tuesday To Do ListAdmin/TipsQuiz 2 marks and sample solution releasedQuiz 3 worth 4 marks is due Week 5 Thursday 27/6 @ 9pmAssignment 1 worth 13 marks is due Week 7 Monday 8/7 @ 10amContentClassesUseful LinksPython help() function in Python:<https://www.geeksforgeeks.org/help-function-in-python/><https://www.tutorialsteacher.com/python/help-method>Difference between Method and Function in Python<https://www.geeksforgeeks.org/difference-method-function-python/>

Slide ID: 362000 | Index: 11 | Title: Week 5 Thursday To Do List | Status: unseen

Week 5 Thursday To Do ListAdmin/TipsQuiz 4 worth 4 marks will be released today @ 7.15pmQuiz 3 worth 4 marks is due today @ 9pmAssignment 1 worth 13 marks is due Week 7 Monday 8/7 @ 10amContentContinue Classes lesson from Class attributes slideWeek 5 Notes 6 Eratosthenes' Sieveimprecision of floating point calculationint() vs round(), rounding half to evenstring formattingrange([first],last,[step]) functioniterables vs iterators, iter()chain() function from itertoolslibrarytimeit() function returns a time the execution takes in seconds as a floatWeek 5 Notes 7 Euler's SieveMatplotlib library - PlottingGlobal variableUseful LinksSieve of Eratostheneshttps://en.wikipedia.org/wiki/Sieve_of_EratosthenesSieve of Euler<https://programmingpraxis.com/2011/02/25/sieve-of-euler/>

Lesson 11: PEP 8 - Style Guide for Python Code

Lesson ID: 53295 | **Created:** 2024-05-27T01:25:36.304315+10:00 | **First Viewed:** 2025-09-18T14:07:55.271355+10:00 | **Last Viewed Slide ID:** 361909 | **State:** active | **Status:** completed | **Slide Count:** 1

Slides

Slide ID: 361909 | **Index:** 3 | **Title:** PEP 8 - Style Guide for Python Code | **Status:** completed

<https://www.python.org/dev/peps/pep-0008/>

Lesson 12: Week 3 - Notes 2 The Monty Hall Problem (optional)

Lesson ID: 53308 | Created: 2024-05-27T01:25:37.572103+10:00 | First Viewed: 2025-09-18T14:41:43.088835+10:00 | Last Viewed Slide ID: 361969 | State: active | Status: attempted | Slide Count: 5

Slides

Slide ID: 361969 | Index: 2 | Title: The Monty Hall problem | Status: completed

Slide ID: 361970 | Index: 3 | Title: Useful links about Monty Hall problem | Status: unseen

Monty Hall problemhttps://en.wikipedia.org/wiki/Monty_Hall_problemThe Monty Hall Problem (from Khan Academy)<https://www.khanacademy.org/partner-content/wi-phi/wiphi-metaphysics-epistemology/probability-philosophy/v/the-monty-hall-problem>Monty Hall Problem Express Explanation<https://www.youtube.com/watch?v=C4vRTzsv4os>Monty Hall Problem - Numberphile<https://www.youtube.com/watch?v=4Lb-6rxZxx0>

Slide ID: 361971 | Index: 4 | Title: The Monty Hall problem | Status: unseen

Slide ID: 361972 | Index: 5 | Title: Monty Hall | Status: unseen

Slide ID: 361973 | Index: 6 | Title: The Monty Hall problem | Status: unseen

Lesson 13: Week 5 - Notes 6 Eratosthenes' Sieve

Lesson ID: 53318 | Created: 2024-05-27T01:25:39.437716+10:00 | State: active | Status: unattempted | Slide Count: 6

Slides

Slide ID: 362012 | Index: 3 | Title: Eratosthenes' sieve | Status: unseen

Slide ID: 362013 | Index: 4 | Title: Eratosthenes' sieve | Status: unseen

Slide ID: 362014 | Index: 5 | Title: eratosthenes_sieve_v1.py | Status: unseen

Slide ID: 362015 | Index: 6 | Title: eratosthenes_sieve_v2.py | Status: unseen

Slide ID: 362016 | Index: 7 | Title: Eratosthenes' sieve I | Status: unseen

Slide ID: 362017 | Index: 8 | Title: Eratosthenes' sieve II | Status: unseen

Lesson 14: Week 7 - Notes 9 US Social Security Data on Given Names

Lesson ID: 53322 | Created: 2024-05-27T01:25:40.538883+10:00 | State: active | Status: unattempted | Slide Count: 6

Slides

Slide ID: 362031 | Index: 1 | Title: US Social Security data on given names | Status: unseen

Slide ID: 362032 | Index: 2 | Title: US Social Security data on given names | Status: unseen

Slide ID: 362033 | Index: 3 | Title: names_per_gender.py | Status: unseen

Slide ID: 362034 | Index: 4 | Title: names_revivals.py | Status: unseen

Slide ID: 362035 | Index: 7 | Title: US Social Security data on given names I | Status: unseen

Slide ID: 362036 | Index: 8 | Title: US Social Security data on given names II | Status: unseen

Lesson 15: Week 8 - Notes 10 K-means Clustering

Lesson ID: 53323 | Created: 2024-05-27T01:25:41.980321+10:00 | State: active | Status: unattempted | Slide Count: 4

Slides

Slide ID: 362037 | Index: 1 | Title: K-means clustering | Status: unseen

Slide ID: 362038 | Index: 2 | Title: K-means clustering | Status: unseen

Slide ID: 362039 | Index: 3 | Title: k_means_clustering.py | Status: unseen

Slide ID: 362040 | Index: 4 | Title: K-means clustering | Status: unseen

Lesson 16: Week 8 - Notes 12 The Game of Life

Lesson ID: 53326 | Created: 2024-05-27T01:25:42.658018+10:00 | State: active | Status: unattempted | Slide Count: 8

Slides

Slide ID: 362047 | Index: 1 | Title: The game of life | Status: unseen

Slide ID: 362048 | Index: 2 | Title: Useful links about game of life | Status: unseen

Conway's Game of Lifehttps://en.wikipedia.org/wiki/Conway%27s_Game_of_LifeGun (cellular automaton)[https://en.wikipedia.org/wiki/Gun_\(cellular_automaton\)](https://en.wikipedia.org/wiki/Gun_(cellular_automaton))

Slide ID: 362049 | Index: 3 | Title: The game of life | Status: unseen

Slide ID: 362050 | Index: 4 | Title: game_of_life.py | Status: unseen

Slide ID: 362051 | Index: 5 | Title: The game of life I | Status: unseen

Slide ID: 362052 | Index: 6 | Title: The game of life II | Status: unseen

Slide ID: 362053 | Index: 7 | Title: The game of life III | Status: unseen

Slide ID: 362054 | Index: 8 | Title: The game of live IV | Status: unseen

Lesson 17: Week 1 - Overview

Lesson ID: 53297 | **Created:** 2024-05-27T01:25:36.374378+10:00 | **First Viewed:** 2025-09-25T11:16:16.736448+10:00 | **Last Viewed Slide ID:** 361911 | **State:** active | **Status:** attempted | **Slide Count:** 3

Slides

Slide ID: 361911 | **Index:** 2 | **Title:** Week 1 - Introduction | **Status:** completed

Slide ID: 364311 | **Index:** 9 | **Title:** Week 1 Tuesday To Do List | **Status:** unseen

Week 1 Tuesday To Do ListAdmin/TipsCheck Lectures and Tutorials Timetable since some locations have changedPython 3 Cheat Sheet - allowed in final exam (as a soft copy)You can generate a PDF version of "Week 1 - Python Programming Fundamentals" by clicking on "..." on the top right corner then "Preview Lesson" then "Download PDF"ContentGo through Week 1 - Introduction PPT slidesStart "Week 1 - Python Programming Fundamentals"

Slide ID: 361912 | **Index:** 10 | **Title:** Week 1 Thursday To Do List | **Status:** unseen

Week 1 Thursday To Do ListAdmin/TipsEd Python EditorCTRL + / to comment/uncomment a selected codeCTRL +] to add one indentation for a selected codeCTRL + [to remove one indentation for a selected codeReset to Scaffold if accidentally deleting lesson/practice/quiz/assignment files. However, save your work first (if any).ContentContinue "Week 1 - Python Programming Fundamentals" with "Importing modules" slidelf statements including match new Python statementWhile statementsShow the flowchart - problem solving from requirements to Python programShow how to run Turing machine simulatorHandling exceptions (if time permits)Working with files (if time permits)Useful LinksWinSCP is an open source free SFTP client, FTP client, WebDAV client, S3 client and SCP client and file manager for Windows. Its main function is file transfer between a local and a remote computer. Download it using the link below:<https://winscp.net/eng/download.php>Numbers in Python<https://realpython.com/python-numbers/>

Slide ID: 361913 | **Index:** 11 | **Title:** More Examples | **Status:** unseen

Lesson 18: Week 10 - Review

Lesson ID: 53298 | Created: 2024-05-27T01:25:36.526112+10:00 | State: active | Status: unattempted | Slide Count: 1

Slides

Slide ID: 361914 | Index: 1 | Title: Week 10 - Review | Status: unseen

Lesson 19: Week 5 - Classes

Lesson ID: 53316 | Created: 2024-05-27T01:25:39.241589+10:00 | State: active | Status: unattempted | Slide Count: 15

Slides

Slide ID: 362001 | Index: 8 | Title: Classes | Status: unseen

There is an important and powerful technique, and another cornerstone of good programming - defining and using classes. You have learned about Python objects and their attributes, and you have been working with them a lot. You have also learned that objects are of different types, and that the type of an object determines what attributes it has, and thus what you can do with it. You might already have found yourself wishing there were other types. Wouldn't it be nice, for example, to have a "passage" type, for large slabs of text, with methods such as `num_paragraphs()`, `num_sentences()`, `num_words()`, `average_word_length()`, and so on? Well, you can actually define your own types of objects, and give them whatever methods you like. Brilliant! You will learn how to do this this week.

Slide ID: 374419 | Index: 69 | Title: What is OOP? | Status: unseen

OOP stands for Object-Oriented Programming. Procedural programming is about writing functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods. Object-oriented programming (OOP) has several advantages over procedural programming: OOP provides a clear structure of the programs OOP makes the code easier to maintain and modify OOP makes it possible to create full reusable applications with less code and shorter development time Python is an object oriented programming language.

Slide ID: 374426 | Index: 70 | Title: What are Classes and Objects? | Status: unseen

Classes and objects are the two main aspects of object-oriented programming. Look at the following two examples to see the difference between class and objects: Therefore, a class is a template for objects, and an object is an instance of a class. When the individual objects are created, they inherit all the data variables and methods from the class.

Slide ID: 362002 | Index: 71 | Title: Defining classes and objects | Status: unseen

You have been working extensively with objects of various types: integers, strings, lists, functions, and so on. Here are some examples: The integer 10 is an object of type `int` The string 'Hello' is an object of type `str` The list `[1, 2, 3]` is an object of type `list` The function `lambda x: x + 10` is an object of type `function` You can confirm these for yourself using the `type()` function: `print(type(10))`

```
print(type('Hello'))
```

```
print(type([1, 2, 3]))
```

```
print(type(lambda x: x + 10))
```

Types are also called classes. That is why you see the word "class" in the results when you run the code above. And objects are also called instances - they are instances of the class that is their type. So we have two equivalent ways of saying the same thing: The integer 10 is an object of type `int` The integer 10 is an instance of the class `int` You can actually define your own classes, and create instances of those classes, and this is a very useful thing to do. You specify the attributes (that is, data and methods) that each object of that class should have. A class is a "blueprint" for creating objects. Defining a class You can define a class using a class statement, which has the following form: `class` :

For example: `class Person:`

`pass` It is conventional to use Capital Case when naming classes (sometimes called Pascal Case). You need to have at least one statement in the body of the class definition - we have just used `pass` above. Creating instances

Once you have defined a class you can create instances of it. You do so by calling the class as if it were a function. In the following code we create two instances of the class `Person`

and assign them to variables p1 and p2:

```
class Person:
    pass
```

```
p1 = Person()
```

```
p2 = Person()
```

You can confirm that p1 and p2 are instances of this class by using type() or isinstance():

```
class Person:
    pass
```

```
p1 = Person()
```

```
p2 = Person()
```

```
print(type(p1), type(p2))
```

```
print(isinstance(p1, Person), isinstance(p2, Person))
```

Setting and getting attributes

Having created these instances you can give them attributes:

```
class Person:
    pass
```

```
p1 = Person()
```

```
p2 = Person()
```

```
p1.first_name = 'Brad'
```

```
p1.last_name = 'Pitt'
```

```
p1.full_name = 'Brad Pitt'
```

```
p2.first_name = 'Angelina'
```

```
p2.last_name = 'Jolie'
```

```
p2.full_name = 'Angelina Jolie'
```

```
print(p1.first_name, p1.last_name, p1.full_name)
```

```
print(p2.first_name, p2.last_name, p2.full_name)
```

The examples above are classes and objects in their simplest form, and are not really useful in real life applications. You will typically want to set attributes of an object as soon as you create it, so there is a special way to do this, using an __init__() method (called constructor). Defining an __init__() method

You can specify what happens when an instance is created by defining a special method __init__() in the class definition. The first parameter must always be named self and represents the newly created object. Any other parameters to the method can be added in the usual way.

```
class Person:
```

```
    def __init__(self, first_name, last_name):
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
        self.full_name = first_name + ' ' + last_name
```

```
p1 = Person('Brad', 'Pitt')
```

```
p2 = Person('Angelina', 'Jolie')
```

```
print(p1.first_name, p1.last_name, p1.full_name)
```

```
print(p2.first_name, p2.last_name, p2.full_name)
```

In the code above, arguments are passed to the class's __init__() method. The self argument is automatically provided by Python. The code in the method adds attributes (first_name, last_name, and full_name) to the newly created object.

Defining other methods

You can also define methods in the class block, which can then be called on an object created from the class, as methods of the object. The functions you define must have self as the first parameter and Python will automatically set it to the object on which the method is called.

```
class Person:
```

```
    def __init__(self, first_name, last_name):
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
    def full_name(self): # Define a method called "full_name"
```

```
        return self.first_name + ' ' + self.last_name
```

```
def reverse_na...
```

Slide ID: 362003 | **Index:** 72 | **Title:** Other special methods | **Status:** unseen

The `__init__()` method is an example of a special instance method. It is also called a dunder method (double underscore method). You do not need to call it explicitly on an object, because Python calls it automatically whenever it needs to create the object, that is, an instance of the class. There are many other special instance methods that you can define. We will look at some examples of the most helpful ones. `__str__()` Consider what happens when you print an instance of the Person class:

```
class Person:
```

```
def __init__(self, first_name, last_name):
```

```
self.first_name = first_name
```

```
self.last_name = last_name
```

```
person = Person('Brad', 'Pitt')
```

```
print(person.__str__())
```

```
# print(person) # same effect as print(person.__str__())
```

It is not very informative - when Python prints the instance, it converts the instance into a string, and the default way it does so is just to give some general information. You can override this default behaviour and specify what string to produce. You do this by adding another special method, `__str__()`.

```
class Person:
```

```
def __init__(self, first_name, last_name):
```

```
self.first_name = first_name
```

```
self.last_name = last_name
```

```
def __str__(self):
```

```
return 'Person: ' + self.first_name + ' ' + self.last_name
```

```
person = Person('Brad', 'Pitt')
```

```
print(person)
```

Now you get a more informative result when you print a person. `__eq__()` You can define how to compare two instances of your class for equality. Let's add age to the class Person and add an `__eq__()` special method, which checks whether two people are equal if they have the same age:

```
class Person:
```

```
def __init__(self, first_name, last_name, age):
```

```
self.first_name = first_name
```

```
self.last_name = last_name
```

```
self.age = age
```

```
def __eq__(self, other):
```

```
if isinstance(other, Person):
```

```
return self.age == other.age
```

```
return False
```

```
john = Person('John', 'Citizen', 25)
```

```
jane = Person('Jane', 'Doe', 25)
```

```
mary = Person('Mary', 'Smith', 27)
```

```
print(john == mary) # False
```

```
print(john == jane) # True
```

Let's switch the example to a "Square" class. Let's define a class of squares, with a side attribute and an `area()` method.

```
class Square:
```

```
def __init__(self, side):
```

```
self.side = side
```

```
def area(self):
```

```
return self.side ** 2
```

```
sq = Square(10)
```

print(sq.area()) Let's add an `__eq__()` special method, which checks whether two squares are equal. Let's consider them equal when they have the same side length. Notice that we need to have a second parameter for the method, to represent the object being compared with.

```
class Square:
```

```
def __init__(self, side):
```

```
self.side = side
```

```
def __eq__(self, other):
```

```
return self.side == other.side
```

```
def area(self):
```

```
return self.side ** 2
```

```
sq1 = Square(10)
```

```
sq2 = Square(11)
```

```
sq3 = Square(10)
```

```
print(sq1 == sq2) # False
```

```
print(sq1 != sq3) # False
```

```
print(sq1 == sq3) # True
```

```
__ne__(), __lt__(), __le__(), __gt__(), and __ge__() Similarly, we can specify how
```

```
to compare two squares using !=, , , >, and >=:
```

```
class Square:
```

```
def __init__(self, side):
```

```
self.side = side
```

```
def __eq__(self, other):
```

```
return self.side == other.side
```

```
def __ne__(self, other):
```

```
return self.side != other.side
```

```
def __lt__(self, other):
```

```
return self.side < other.side
```

```
def __ge__(self, other):
```

```
return self.side >= other.side
```

```
sq1 = Square(10)
```

```
sq2 = Square(11)
```

```
print(sq1 != sq2) # True
```

```
print(sq1 < sq2) # False
```

```
print(sq1 >= sq2) # False
```

Others There are many other special methods that you can define, to specify how the objects of your class should behave when operated on, including:

Operator	Method
----------	--------

+	object.__add__(self, other)
---	-----------------------------

-	object.__sub__(self, other)
---	-----------------------------

*	object.__mul__(self, other)
---	-----------------------------

/	object.__div__(self, other)
---	-----------------------------

%	object.__mod__(self, other)
---	-----------------------------

**	object.__pow__(self, other)
----	-----------------------------

&	object.__and__(self, other)
---	-----------------------------

^	object.__xor__(self, other)
---	-----------------------------

	object.__or__(self, other)
--	----------------------------

+=	object.__iadd__(self, other) ("i" for in place)
----	---

-=	object.__isub__(self, other)
----	------------------------------

*=	object.__imul__(self, other)
----	------------------------------

/=	object.__idiv__(s...
----	----------------------

The attributes you have defined so far are called instance attributes - they are attributes of instances of the class, not of the class itself. Sometimes you might want to add attributes to a class itself, rather than to instances of the class. These are called class attributes. If the attribute is a method then you can make it a class method by omitting the self parameter from the method definition. class Square:

```
def __init__(self, side):
    self.side = side
```

```
def area(self): # self parameter - this is an instance method
    return self.side ** 2
```

```
def calculate_area(side): # No self parameter - this is a class method
    return side ** 2
```

```
sq = Square(10) # Create an instance
print(sq.area()) # Invoke an instance method
print(Square.calculate_area(20)) # Invoke a class method
```

The method can be invoked by using the class name and the method name. It provides a way of organising methods that are related to the class, but do not belong on the instances themselves. If the attribute is a data then you can make it a class data by defining it outside the constructor. It will be then shared by all class instances. It represents a characteristic of the entire class rather than individual objects. Class fields have a single value shared by all instances. Hence changing the value impacts all instances equally as shown in the example below:

```
class Square:
```

```
    nbSquares = 0
```

```
    def __init__(self, side):
        self.side = side
        Square.nbSquares += 1
```

```
    def area(self): # self parameter - this is an instance method
        return self.side ** 2
```

```
    def calculate_area(side): # No self parameter - this is a class method
        return side ** 2
```

```
sq = Square(10) # Create an instance
print(sq.area()) # Invoke an instance method
print(Square.calculate_area(20)) # Invoke a class method
print(Square.nbSquares) # Outputs 1
print(sq.nbSquares) # Outputs 1
sq2 = Square(10) # Create another instance
print(Square.nbSquares) # Outputs 2
```

You might find yourself working with lengths. There are many different units in which lengths are measured, including: SI (Système International) units: mm, cm, m, km Imperial/US Customary units: in, ft, yd, mi Let's define a class Length to help us work with lengths of various units. Let's define it so that: We can create a Length object by supplying a number and a unit (let's stick to the units above). We can get the length of a length object in whatever units we like. We can add a length object to another. We can print a length object in an informative way. We'll need the following conversions: 1mm = 1/1000m 1cm = 1/100m 1km = 1000m 1yd = 0.9144m 1ft = 1/3yd 1in = 1/12ft 1mi = 1760yd Here's one way to define the class:

```
class
```

Length:

```
"""A class to help work with lengths in various units"""
```

```
def __init__(self, number, unit='m'):
    # Convert and store length as self.metres
    # SI units
    if unit == 'mm': self.metres = number/1000
    elif unit == 'cm': self.metres = number/100
    elif unit == 'm': self.metres = number
    elif unit == 'km': self.metres = number*1000
    # Imperial/US Customary units
    elif unit == 'in': self.metres = (number/36)*0.9144
    elif unit == 'ft': self.metres = (number/3)*0.9144
    elif unit == 'yd': self.metres = (number)*0.9144
    elif unit == 'mi': self.metres = (number*1760)*0.9144
    # Unit not recognised
    else: raise Exception("Unit not recognised")
```

```
def to(self, unit, dp=None):
    # Convert self.metres to unit
    # SI units
    if unit == 'mm': number = self.metres*1000
    elif unit == 'cm': number = self.metres*100
    elif unit == 'm': number = self.metres
    elif unit == 'km': number = self.metres/1000
    # Imperial/US Customary units
    elif unit == 'in': number = (self.metres*36)/0.9144
    elif unit == 'ft': number = (self.metres*3)/0.9144
    elif unit == 'yd': number = (self.metres)/0.9144
    elif unit == 'mi': number = (self.metres/1760)/0.9144
    else: raise Exception("Unit not recognised")
    if dp is not None: number = round(number, dp)
    return f"{number}{unit}"
```

```
def __str__(self):
    return f"Length: {self.metres}m"
```

```
def __add__(self, other):
    return Length(self.metres + other.metres)
```

```
# Try it out
print(Length(6, 'ft').to('cm'))
print(Length(6, 'ft').to('cm', 1))
print(Length(6, 'ft').to('cm', 0))
print(Length(172.5, 'cm').to('ft'))
print(Length(6, 'ft') + Length(2.5, 'm'))
print((Length(6, 'ft') + Length(2.5, 'm')).to('yd'))
```

Slide ID: 362005 | **Index:** 160 | **Title:** Attributes can be class instances | **Status:** unseen

There is nothing stopping you from having an attribute of an object being an instance of another class, or perhaps even the same class. Consider the following modified definition of the Person class:

```
class Person:
    def __init__(self, first_name, last_name, boss = None):
```

```
self.first_name = first_name
self.last_name = last_name
self.boss = boss
```

```
def full_name(self):
    return self.first_name + ' ' + self.last_name
```

```
def reverse_name(self):
    return self.last_name + ', ' + self.first_name
```

We have added a new attribute, boss, which is set when a person is created. It can be set to any object. That means it can be set to a person. Here is an example:

```
class Person:
    def __init__(self, first_name, last_name, boss = None):
        self.first_name = first_name
        self.last_name = last_name
        self.boss = boss
```

```
def full_name(self):
    return self.first_name + ' ' + self.last_name
```

```
def reverse_name(self):
    return self.last_name + ', ' + self.first_name
```

```
basil = Person('Basil', 'Fawltly')
polly = Person('Polly', 'Sherman', basil) # Set Polly's boss to the Person object basil
print(polly.boss.full_name())
```

Notice that we can print the full name of polly's boss using the expression polly.boss.full_name(). You can also write the __str__() special method to cater for that as shown below:

```
class Person:
    def __init__(self, first_name, last_name, boss = None):
        self.first_name = first_name
        self.last_name = last_name
        self.boss = boss
```

```
def full_name(self):
    return self.first_name + ' ' + self.last_name
```

```
def reverse_name(self):
    return self.last_name + ', ' + self.first_name
```

```
def __str__(self):
    if self.boss == None:
        return self.first_name + " " + self.last_name
    return self.first_name + " " + self.last_name + " " + str(self.boss)
# or return self.first_name + " " + self.last_name + " " + self.boss.__str__()
```

```
basil = Person('Basil', 'Fawltly')
polly = Person('Polly', 'Sherman', basil) # Set Polly's boss to the Person object basil
print(basil) # outputs Basil Fawltly
print(polly) # outputs Polly Sherman Basil Fawltly
```

Here is another example of a Course class that has Textbook and Instructor as instance attributes:

```
class Textbook:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn

    def __str__(self):
        return f'{self.title} by {self.author}'

class Instructor:
    def __init__(self, name, department):
        self.name = name
        self.department = department

    def __str__(self):
        return f'{self.name} from {self.department} department'

class Course:
    def __init__(self, course_code, course_name, textbook, instructor):
        self.course_code = course_code
        self.course_name = course_name
        self.textbook = textbook
        self.instructor = instructor

    def __str__(self):
        return f'Course Code: {self.course_code}\nCourse Name: {self.course_name}\nTextbook: {self.textbook}\nInstructor: {self.instructor}'

# Example usage:

# Creating instances of Textbook and Instructor
textbook1 = Textbook("Python Programming", "John Smith", "978-0134852045")
instructor1 = Instructor("Jane Doe", "Computer Science")

# Creating a Course instance using the Textbook and Instructor instances
course1 = Course("COMP1001", "Introduction to Python", textbook1, instructor1)
# Printing out information about the course
print(course1)
The code will output the following:
Course Code: COMP1001
Course Name: Introduction to Python
Textbook: Python Programming by John Smith
Instructor: Jane Doe from Computer Science department
```

Slide ID: 362007 | **Index:** 162 | **Title:** Class inheritance | **Status:** unseen

Sometimes you might want to define a class as a subclass of another class. Suppose you've defined a Person class as before:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def full_name(self):
```



```
return self.first_name + ' ' + self.last_name
```

```
def reverse_name(self):
```

```
return self.last_name + ', ' + self.first_name
```

Suppose you are especially interested in a certain type of person - your employees. You have special data about them that you'd like to keep, and special methods that you'd like to use to manipulate that data. Suppose you decide to create an Employee class for them.

The `full_name()` and `reverse_name()` methods of the Person class are quite handy, and they apply equally well to employees. So you would probably want to add them to the definition of the Employee class. But then you would just be duplicating code, which is a bad idea. Fortunately, you do not have to. You can specify, as part of your definition of Employee, that it is a subclass of Person meaning that instances of Employee class are also instances of Person class (an employee is just a special kind of person). When you do that, every instance of Employee automatically inherits all of the attributes defined in Person. To make Employee a subclass (or child class) of Person (which is then called the parent class, or superclass), you just add Person in brackets after the class name. Then you can start creating instances of Employee, which have all the attributes (including the methods), of instances of Person:

```
class Person:
```

```
def __init__(self, first_name, last_name):
```

```
self.first_name = first_name
```

```
self.last_name = last_name
```

```
def full_name(self):
```

```
return self.first_name + ' ' + self.last_name
```

```
def reverse_name(self):
```

```
return self.last_name + ', ' + self.first_name
```

```
class Employee(Person): # Add Person in brackets, to make it a subclass of Person
```

```
pass
```

```
x = Employee('John', 'Smith')
```

```
print(x.full_name())
```

Now you can start adding your special attributes to the Employee class. Instances of Employee will get these attributes, in addition to the attributes they get from Person.

```
class Person:
```

```
def __init__(self, first_name, last_name):
```

```
self.first_name = first_name
```

```
self.last_name = last_name
```

```
def full_name(self):
```

```
return self.first_name + ' ' + self.last_name
```

```
def reverse_name(self):
```

```
return self.last_name + ', ' + self.first_name
```

```
class Employee(Person):
```

```
role = None
```

```
def full_name(self):
```

```
return self.first_name + ' ' + self.last_name + ', ' + self.role
```

```
x = Employee('John', 'Smith')
```

```
x.role = 'Director'
```

```
print(x.full_name())
```

```
print(x.reverse_name())
```

Notice that we can override methods of the parent class in the child class. We've

overridden full_name() method, by defining a different version of it in the child class, in which it includes the employee's role. But we've not overridden reverse_name(). We usually define data attributes using the constructor method, as we did for the class Person previously, as shown in the improved example below:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
```

```
    def full_name(self):
        return self.first_name + ' ' + self.last_name
    def __str__(self):
        return self.first_name + ' ' + self.last_name
```

```
    def reverse_name(self):
        return self.last_name + ', ' + self.first_name
```

```
class Employee(Person):
    def __init__(self, first_name, last_name, role):
        super().__init__(first_name, last_name)
        self.role = role
```

```
    def full_name(self):
        return self.first_name + ' ' + self.last_name + ', ' + self.role
    def __str__(self):
        return self.first_name + ' ' + self.last_name + ', ' + self.role
```

```
x = Employee('John', 'Smith', 'Director')
print(x).full_name()
print(x.reverse_name())
y = Person('Jane', 'Doe')
print(y).full_name()
```

Note the use of the super() function to access methods of the parent class.

Slide ID: 374437 | **Index:** 165 | **Title:** Class Polymorphism | **Status:** unseen

Polymorphism means many forms, and in programming it refers to functions/methods/operators with the same name that can be executed on many objects or classes. Polymorphism is often used in class methods, where we can have multiple classes with the same method name. For example, say we have three classes: Car, Boat, and Plane, and they all have a method called move():

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
```

```
    def move(self):
        print("Drive!")
```

```
class Boat:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
```

```
    def move(self):
        print("Sail!")
```

```
class Plane:
def __init__(self, brand, model):
self.brand = brand
self.model = model
```

```
def move(self):
print("Fly!")
```

```
car1 = Car("Ford", "Mustang") #Create a Car class
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class
plane1 = Plane("Boeing", "747") #Create a Plane class
```

```
for x in (car1, boat1, plane1):
x.move()
```

Note the for loop at the end. Because of polymorphism, we can execute the same method for all three classes.

Slide ID: 374438 | **Index:** 166 | **Title:** Inheritance Class Polymorphism | **Status:** unseen

What about classes with child classes with the same name? Can we use polymorphism there? Yes. If we use the example from the previous slide (Class Polymorphism) and make a parent class called Vehicle, and make Car, Boat, and Plane child classes of Vehicle, the child classes inherit the Vehicle methods, but can override them:

```
class Vehicle:
def __init__(self, brand, model):
self.brand = brand
self.model = model
```

```
def move(self):
print("Move!")
```

```
class Car(Vehicle):
pass
```

```
class Boat(Vehicle):
def move(self):
print("Sail!")
```

```
class Plane(Vehicle):
def move(self):
print("Fly!")
```

```
car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object
```

```
for x in (car1, boat1, plane1):
print(x.brand)
print(x.model)
x.move()
```

print()

Child classes inherit the attributes and methods from the parent class. In the example above you can see that the Car class is empty, but it inherits brand, model, and move() from Vehicle. The Boat and Plane classes also inherit brand, model, and move() from Vehicle, but they both override the move() method. Because of polymorphism, we can execute the same method for all classes.

When you define a function, or define a class, or create a module, there is a special kind of comment that you can add, called a docstring. Functions

Here is an example of a docstring being used in a function definition:

```
def full_name(first_name, last_name):  
    """Returns 'first_name last_name'"""
```

```
    return (first_name + ' ' + last_name).strip()
```

A function docstring is just a string literal, but: It must be triple quoted (triple-single or triple-double) (PEP8 style guide recommends triple-double) It must be the first line of the function body Why use a docstring? Why not just comments, as usual? Because Python recognises docstrings and uses them to help document your code. Notice what happens when you ask Python for help about the function:

```
def full_name(first_name, last_name):
```

```
    """Returns 'first_name last_name'"""
```

```
    return (first_name + ' ' + last_name).strip()
```

Classes Here is an example of docstrings being used inside a class definition:

```
class Person:  
    """Represents a person"""
```

```
    def __init__(self, first_name, last_name):
```

```
        """This is a constructor"""
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
    def full_name(self):
```

```
        """Returns 'first_name last_name'"""
```

```
        return self.first_name + ' ' + self.last_name
```

```
    def reverse_name(self):
```

```
        """Returns 'last_name, first_name'"""
```

```
        return self.last_name + ', ' + self.first_name
```

Each function definition inside the class definition can have a docstring, as per functions in general. In addition, the class definition itself can have a docstring. The rules are the same: It must be triple quoted (triple-single or triple-double) It must be the first line of the class

body And the purpose is the same:

```
class Person:  
    """Represents a person"""
```

```
    def __init__(self, first_name, last_name):
```

```
        """This is a constructor"""
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
    def full_name(self):
```

```
        """Returns 'first_name last_name'"""
```

```
        return self.first_name + ' ' + self.last_name
```

```
    def reverse_name(self):
```

```
        """Returns 'last_name, first_name'"""
```

```
        return self.last_name + ', ' + self.first_name
```

Modules You can add a docstring to any module that you create, too. The rules are the same, and the purpose is the same. Suppose you have a module called "people.py" in which you have a bunch of function and class definitions to help work with people. Here's how the start of your module might look:

```
"""A collection of functions and classes to help work with people"""
```

```
class Person:
```

```
    """Represents a person"""
```

```
    def __init__(self, first_name, last_name):
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

```
    def full_name(self):
```

```
        """Returns 'first_name last_name'"""
```

```
return self.first_name + ' ' + self.last_name
def reverse_name(self):
    """Returns 'last_name, first_name'"""
    return self.last_name + ', ' + self.first_name
```

```
def add_two_numbers(num1, num2):
    """Returns the sum of two given numbers"""
```

return num1 + num2

Anyone who imports your module and runs `help(people)` (assuming they haven't given it an alias) will see the information in your docstring. You can try it with one of the modules we have been importing. Rather than running `help(math)`, which will give us a lot of information, we can run `print(math.__doc__)` - this will show us just the docstring of the module:

```
import math
```

```
print(math.__doc__)
input("Press to see the help about the module")
help(math)
```

Slide ID: 362009 | **Index:** 169 | **Title:** Documenting your modules - Example | **Status:** unseen

Slide ID: 362010 | **Index:** 170 | **Title:** Further reading | **Status:** unseen

You might find the following helpful: [The Python Tutorial at w3schools.com](https://www.w3schools.com/python/)

Lesson 20: Week 10 - Searching and Sorting

Lesson ID: 53333 | **Created:** 2024-05-27T01:25:44.68356+10:00 | **First Viewed:** 2025-07-16T15:46:51.107512+10:00 | **Last Viewed Slide ID:** 362090 | **State:** active | **Status:** unattempted | **Slide Count:** 13

Slides

Slide ID: 362081 | **Index:** 2 | **Title:** Useful links about Searching and Sorting | **Status:** seen

Binary and Linear Search Visualization<https://www.cs.usfca.edu/~galles/visualization/Search.html>Linear Search<https://www.geeksforgeeks.org/linear-search/Binary> Search<https://www.geeksforgeeks.org/binary-search/Selection> Sort<https://www.geeksforgeeks.org/selection-sort/Bubble> Sort<https://www.geeksforgeeks.org/bubble-sort/Insertion> Sort<https://www.geeksforgeeks.org/insertion-sort/Quick> Sort<https://www.geeksforgeeks.org/quick-sort/A> tour of the top 5 sorting algorithms with Python code<https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa0288915> Sorting Algorithms in 6 Minutes<https://www.youtube.com/watch?v=kPRA0W1kECgA> Bubble Sorting Algorithm animated example<https://www.youtube.com/watch?v=9I2oOAr2okY>The Quick Sort<https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html>Sorting, searching and algorithm analysishttps://python-textbok.readthedocs.io/en/1.0/Sorting_and_Searching_Algorithms.htmlPython Programming Examples on Searching and Sorting<https://www.sanfoundry.com/python-programming-examples-searching-sorting/A> tour of the top 5 sorting algorithms with Python code<https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa02889>Sorting and Searching in Python<https://code.tutsplus.com/tutorials/sorting-and-searching-in-python--cms-25668>Batcher odd-even mergesorthttps://en.wikipedia.org/wiki/Batcher_odd-even_mergesort

Slide ID: 362089 | **Index:** 3 | **Title:** linear_search.py | **Status:** seen

Slide ID: 362084 | **Index:** 4 | **Title:** binary_search.py | **Status:** seen

Slide ID: 362085 | **Index:** 5 | **Title:** binary_search_recursive.py | **Status:** seen

Slide ID: 362082 | **Index:** 6 | **Title:** sorting.py | **Status:** seen

Slide ID: 362086 | **Index:** 7 | **Title:** bubble_sort.py | **Status:** seen

Slide ID: 362092 | **Index:** 8 | **Title:** selection_sort.py | **Status:** seen

Slide ID: 362088 | **Index:** 9 | **Title:** insertion_sort.py | **Status:** seen

Slide ID: 362093 | **Index:** 10 | **Title:** shell_sort.py | **Status:** seen

Slide ID: 362090 | **Index:** 11 | **Title:** merge_sort.py | **Status:** seen

Slide ID: 362091 | **Index:** 12 | **Title:** quick_sort.py | **Status:** seen

Slide ID: 362087 | **Index:** 13 | **Title:** heap_sort.py | **Status:** seen

Slide ID: 362083 | **Index:** 14 | **Title:** batcher_sort.py | **Status:** seen

Lesson 21: The Babylonian method for computing square roots

Lesson ID: 53356 | Created: 2024-05-27T01:25:56.072082+10:00 | State: active | Status: unattempted | Slide Count: 4

Slides

Slide ID: 362180 | Index: 1 | Title: The Babylonian method for computing square roots | Status: unseen

Slide ID: 362181 | Index: 2 | Title: The Babylonian method for computing square roots | Status: unseen

Slide ID: 362182 | Index: 3 | Title: sqrt_approximation.py | Status: unseen

Slide ID: 362183 | Index: 5 | Title: The Babylonian method for computing square roots | Status: unseen

Lesson 22: The Vigenere cipher

Lesson ID: 53357 | Created: 2024-05-27T01:25:56.317547+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 384765 | Index: 1 | Title: The Vigenere cipher | Status: unseen

Slide ID: 362184 | Index: 2 | Title: The Vigenere cipher | Status: unseen

Lesson 23: Nash equilibrium

Lesson ID: 53358 | Created: 2024-05-27T01:25:56.567805+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 384764 | Index: 1 | Title: Nash equilibrium | Status: unseen

Slide ID: 362185 | Index: 2 | Title: Nash equilibrium | Status: unseen

Lesson 24: Continued fractions

Lesson ID: 53362 | Created: 2024-05-27T01:25:57.339493+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 384760 | Index: 1 | Title: Continued fractions | Status: unseen

Slide ID: 362189 | Index: 2 | Title: Continued fractions | Status: unseen

Lesson 25: Fractals

Lesson ID: 53363 | Created: 2024-05-27T01:25:57.512953+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 384759 | Index: 1 | Title: Fractals | Status: unseen

Slide ID: 362190 | Index: 2 | Title: Fractals | Status: unseen

Lesson 26: Week 1 - Notes 1 Turing Machines (optional)

Lesson ID: 53302 | Created: 2024-05-27T01:25:36.918745+10:00 | State: active | Status: unattempted | Slide Count: 6

Slides

Slide ID: 361932 | Index: 1 | Title: Turing machines | Status: unseen

Slide ID: 361933 | Index: 6 | Title: Useful link about Turing machines | Status: unseen

https://en.wikipedia.org/wiki/Turing_machine

Slide ID: 361934 | Index: 7 | Title: Turing machines | Status: unseen

Slide ID: 361935 | Index: 8 | Title: Turing Machine Simulator | Status: unseen

Slide ID: 361936 | Index: 9 | Title: Turing machines I | Status: unseen

Slide ID: 361937 | Index: 10 | Title: Turing machines II | Status: unseen

Lesson 27: Week 2 - Overview

Lesson ID: 53303 | **Created:** 2024-05-27T01:25:37.284146+10:00 | **First Viewed:** 2025-09-18T14:08:06.188986+10:00 | **Last Viewed Slide ID:** 361938 | **State:** active | **Status:** attempted | **Slide Count:** 2

Slides

Slide ID: 361938 | **Index:** 9 | **Title:** Week 2 Tuesday To Do List | **Status:** completed

Week 2 Tuesday To Do List Admin/Tips Quiz 1 worth 4 marks will be released Week 2 Thursday and due Week 3 Thursday 9pm (you will have one week to do it) Week 3 Monday is a public holiday. Students missing Week 3 Monday tutorial are strongly advised to attend one of the four online tutorials in Week 3 to catch up on the missed Week 3 Monday tutorial. Content Continue showing how to run Turing machine simulator Importing modules slide revisited (added extra examples) Continue "Week 1 - Python Programming Fundamentals" with "Controlling program flow" slide if statements, body/block, indentation, elif, and match while statements, continue, and break Exceptions, try, except, and else Working with files, open(), close(), with(), Show the flowchart - problem solving from requirements, example.py rewritten using match statement Go through "Week 2 - Lists, Tuples, Sets, and Dictionaries"

Slide ID: 361939 | **Index:** 11 | **Title:** Week 2 Thursday To Do List | **Status:** unseen

Week 2 Thursday To Do List Admin/Tips Quiz 1 worth 4 marks will be released today @ 7.15pm Online Tutorials recordings in BB Collaborate Plagiarism and Academic Integrity Reminder (see today's announcement on Ed) Assignment 1 worth 13 marks will be released Week 3 Tuesday Content Start "Week 2 - Lists, Tuples, Sets, and Dictionaries" lesson Collections Creating a collection Inspecting a collection Selecting elements Looping through a collection for loops break and continue The range() function Adding elements Adding to a list: append(), insert(), extend(), and using + operator No adding to tuples since they are immutable Adding to a set: add() and update() Adding to a dictionary by specifying a value for a new key or updating it Removing elements Removing from a list: del, set slice to empty list, pop(), remove(), and clear() No removing from tuples since they are immutable Removing from a set: remove(), discard(), and clear() Removing from a dictionary: del, pop(), and clear() Modifying elements Modifying list elements: assignment and slice No tuples modification since they are immutable Modifying set elements: can not be changed, but remove then add Modifying dictionary elements: using assignment similarly to list elements Sorting elements sort() method (lists only and in-place) sorted() method (applies to all and returns a list) Joining elements using join() Special string operations Strings as tuples Splitting strings Special set operations: union, intersection, difference, symmetric difference, comparing sets Comprehensions Files as lists: readlines(), writelines(), reading CSV files Dates and times Useful Links ASCII Table <https://www.ascii-code.com/> Python List sort() Method <https://www.programiz.com/python-programming/methods/list/sort> Python sorted() Method <https://www.programiz.com/python-programming/methods/built-in/sorted>

Lesson 28: Week 2 - Lists, Tuples, Sets, and Dictionaries

Lesson ID: 53304 | Created: 2024-05-27T01:25:37.330126+10:00 | State: active | Status: unattempted | Slide Count: 17

Slides

Slide ID: 361940 | Index: 112 | Title: Collections | Status: unseen

So far we have worked mostly with individual objects - individual numbers, individual strings, and so on. But it is very common, when programming, to work with whole collections of objects - collections of numbers, collections of strings, even mixed collections of objects of different types. As programming languages have developed, collections have proven to be a cornerstone of working with data, and Python has some of the most powerful techniques for manipulating collections among modern programming languages. Python provides four built-in types of collection, each useful in their own way: Lists, Tuples, Sets, Dictionaries.

Lists A list is an ordered collection of objects (possibly empty). The objects can be of any type, and they can be repeated. **Tuples** A tuple is like a list except that it is immutable, which means that objects cannot be added to a tuple, they cannot be removed from a tuple, and they cannot be reordered within a tuple. If it is important that the collection does not change then you should use a tuple, even if the careful use of a list would achieve the same thing. By using a tuple you guarantee that it won't change, and you also signal your intentions more clearly to anyone who reads your code. **Sets** A set is an unordered collection of objects (possibly empty), each of which is unique (cannot be repeated). If it is important that the collection does not contain the same object twice then you should use a set, even if the careful use of a list would achieve the same thing. By using a set you guarantee that there won't be duplicates, and you also signal your intentions more clearly to anyone who reads your code. **Dictionaries** A dictionary is an ordered collection of key-value pairs. In addition, the keys must be unique. You can think of a dictionary as a mapping from a set of keys to some values. You will learn more about what this means as you work your way through this week's slides. In earlier versions of Python dictionaries were not ordered. They have been ordered since version 3.7.

Slide ID: 361941 | Index: 113 | Title: Creating a collection | Status: unseen

You can create a collection in a variety of ways.

List literals A list literal is a sequence of literals, each of which refers to an object, separated by commas, and surrounded by square brackets. For example: `x = [2, 4, 6, 8]`

```
x = ['cat', 'mouse', 'cat', 'mouse']
x = [1, 3.14, 'a', True]
```

`x = []` Note the last literal above - it is the literal for the empty list. Also note that the same object can be included more than once in a list. The expressions that you use within square brackets need not be literals - they could be variables or other complex expressions. For example: `y = 45`

```
x = [2*23, 4-1, abs(-6), y]
```

`print(x)` But, strictly speaking, if they are not literals then the whole thing does not count as a list literal - it's only a list literal if all of the expressions in the square brackets are themselves literals.

Tuple literals A tuple literal is like a list literal, except you use round brackets rather than square brackets: `x = (2, 4, 6, 8)`

```
x = ('cat', 'mouse', 'dog')
x = (1, 3.14, 'a', True)
```

`x = ()` Note the last literal again - it is the literal for the empty tuple. In some contexts you can drop the round brackets around a tuple. We have already seen an example of this - variable unpacking. For example, line 1 below is just shorthand for line 2: `x, y = 1, 2`

```
(x, y) = (1, 2)
```

You should be careful when doing this, because it sometimes causes errors, and it can make your code less readable.

Set literals A set literal is also like a list literal, except you use curly brackets rather than square brackets: `x = {2, 4, 6, 8}`

```
x = {'cat', 'mouse', 'dog'}
```

`x = {1, 3.14, 'a', True}`What about the empty set? You might expect that you could use `{}`, but unfortunately that refers to the empty dictionary (see below). If you want an empty set you have to use `set()` (more on this below).The items in a set are unique, so what happens if you try to include an object more than once?

Try it:`x = {'cat', 'mouse', 'dog', 'cat', 'cat'}`

`print(x)`As you can see, Python silently ignores all but one occurrence of the object.Dictionary literalsA

dictionary literal uses curly brackets, like set literals, but with key-value pairs separated by commas, and each key and value separated by a colon.`x = {1: 'cat', 2: 'dog', 3: 'mouse'}`

`x = {'A': 0, 'B': 1, 'E': 2, 'M': 3}`

`x = {}`Note the last literal again - it is the literal for the empty dictionary.Although you'll mostly use integers, floats, and strings for the keys of a dictionary, they can also be booleans or tuples:`x = {True: 1, False: 0}`

`print(x)`

`x = {(0, 0): 'bottom left', (1, 1): 'top right'}`

`print(x)`They cannot be lists or sets or dictionaries. Try it:`try: x = {[1, 2]: 'First list', [3, 4]: 'Second list'}
except Exception as e: print(e)`

`try: x = {{1, 2}: 'First set', {3, 4}: 'Second set'}`

`except Exception as e: print(e)`

`try: x = {{1: 2}: 'First dictionary', {3: 4}: 'Second dictionary'}`

`except Exception as e: print(e)` The problem with lists, sets and dictionaries is that they are mutable, which means that Python cannot hash them (which means, roughly, giving them a unique unchanging value).The keys do not all have to be the same type, and neither do the values:`x = {1: 'one', 'two': 2, 3.176: False}`

`print(x)`However, mixing types can lead to confusion, and it is good programming practice to keep the

types the same.In a dictionary, the values can be duplicated but the keys must be unique. So what

happens if you use the same key twice? Try it:`x = {'A': 0, 'B': 1, 'A': 2, 'B': 3}`

`print(x)`As you can see, if there are multiple items with the same key then Python silently ignores all but the

last one.Collections of collectionsThe items in a collection can be objects of any type. In particular, they

can be collections. So it's possible to create collections of collections. For example:# A list of lists

`x = [[1, 2, 3], [3, 4, 5], [6, 7, 8]]`

`print(x)`

A set of tuples

`x = {(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')}`

`print(x)`Using other collectionsYou can also use the `list()`, `tuple()`, `set()`, and `dict()` functions to create a collection o...

Slide ID: 361942 | **Index:** 114 | **Title:** Inspecting a collection | **Status:** unseen

There are a variety of ways in which you can inspect a collection.Number of itemsYou can get the number of items in a collection using Python's `len()` function:`print(len([1, 2, 3])) # List`

`print(len((1, 2, 3))) # Tuple`

`print(len({1, 2, 3})) # Set`

`print(len({'a': 1, 'b': 2, 'c': 3})) # Dictionary`

`print(len([])) # Empty list`Notice that if you have a collection of collections, `len()` only counts the number of items at the top level:`numbers = [[1, 2, 3], [3, 4, 5], [5, 6, 7]]`

`print(len(numbers))`Existence of an itemYou can check whether a collection contains an item using the `in` keyword. Note that in the case of dictionaries it is the keys that are checked.`print(1 in [1, 2, 3]) # True`

`print('a' in [1, 2, 3]) # False`

`print(1 in (1, 2, 3)) # True`

`print('a' in (1, 2, 3)) # False`

`print(1 in {1, 2, 3}) # True`

`print('a' in {1, 2, 3}) # False`

```

print(1 in {'1': 'a', 2: 'b', 3: 'c'}) # True - keys are checked
print('a' in {'1': 'a', 2: 'b', 3: 'c'}) # False - keys are checked
You can check whether an item is not in a collection either by using in and checking whether False is returned, or by using not in and checking whether True is returned:
print('a' in [1, 2, 3]) # False
print('a' not in [1, 2, 3]) # True
You can use the index() method to find the index of the first occurrence of a value in a list or tuple. If the value is not found then Python raises an error.
x = [1, 3, 7, 8, 7, 5, 4, 6, 8, 5, 7]
print(x.index(7)) # Only the first occurrence is found
Frequency of an item
If you want to know how many times an item occurs in a list or tuple you can use the count() method (sets and dictionaries do not have this method):
x = [1, 1, 2, 2, 2]
print(x.count(1))
x = (1, 1, 2, 2, 2)
print(x.count(1))
print(x.count('a'))
You could also use this to check whether a list or tuple contains a certain item (if not, the count will be zero).
Minimum item
You can find the minimum item in a collection by using the min() function. In the case of dictionaries it compares their keys, rather than their values.
print(min([1, 2, 3])) # List
print(min((1, 2, 3))) # Tuple
print(min({1, 2, 3})) # Set
print(min({'a': 10, 'b': 5, 'c': 1})) # Dictionary - compares the keys
Maximum item
You can find the maximum item in a collection by using the max() function. Again, in the case of dictionaries it compares their keys.
print(max([1, 2, 3])) # List
print(max((1, 2, 3))) # Tuple
print(max({1, 2, 3})) # Set
print(max({'a': 10, 'b': 5, 'c': 1})) # Dictionary - compares the keys
Sum of the items
You can find the sum of the items in a collection by using the sum() function. In the case of dictionaries it sums their keys (so there will be an error if the keys cannot be summed, such as when they are letters).
print(sum([1, 2, 3])) # List
print(sum((1, 2, 3))) # Tuple
print(sum({1, 2, 3})) # Set
print(sum({'1': 'a', 2: 'b', 3: 'c'})) # Dictionary - sums the keys
Checking if all items are true
You can check whether all the items in a collection evaluate to True using the all function. As you probably expect by now, in the case of dictionaries it checks the keys, not the values.
print(all([True, True]))
print(all([True, False]))
print(all(('a', 'b')))
print(all(("a", 'b')))
print(all({1, 2}))
print(all({0, 1}))
print(all({'1': 'a', 2: 'b', 3: 'c'}))
print(all({'0': 'a', 1: 'b', 1: 'c'}))
Checking if any items are true
You can check whether any of the items in a collection evaluate to True using the any function. Again, in the case of dictionaries it checks the keys, not the values.
print(any([True, False]))
print(any([False, False]))
print(any(("a", 'b')))
print(any(("a", 0)))
print(any({1, 0}))
print(any({0, 0}))
print(any({'0': 'a', 1: 'b', 2: 'c'}))
print(any({'0': 'a', 1: 'b', False: 'c'}))

```

Slide ID: 361943 | **Index:** 115 | **Title:** Selecting elements | **Status:** unseen

Lists and tuples The items of a list or tuple are ordered, so each one has a position in the list. An item's position is also called its index. The indexes are integers, starting from 0. So, the first item has index 0, the

second item has index 1, and so on. You can select items in a list or tuples by using the indexing operator

```
[]:x = ['a', 'b', 'c', 'd', 'e']
```

```
print(x[0])
```

print(x[3]) Indexes can also be negative. The last item has index -1, the second last item has index -2, and

so on. x = ['a', 'b', 'c', 'd', 'e']

```
print(x[-1])
```

```
print(x[-2])
```

 You can also use the indexing operator to get a slice of a list or tuple. The syntax for slicing is

[start:end], where start is the start index and end is the end index. The cuts are made just before the items

at the start and end indexes, so the slice includes the start item but not the end item. x = ['a', 'b', 'c', 'd', 'e']

```
print(x[0:3])
```

 # Prints items with indexes 0, 1, 2 Notice that x[0:3] returns items with index 0 to 2, rather than

0 to 3 as you might expect - this can be very confusing! You can also use negative indices when slicing: x =

```
['a', 'b', 'c', 'd', 'e']
```

```
print(x[1:-1])
```

 If you don't specify a start index then it is assumed to be zero. If you don't specify an end index

then it is assumed to be the length of the list (an invalid index, but as it is excluded - this is fine and is the

way to include the last element of the list). If you don't specify either then the whole list is returned. x = ['a',

```
'b', 'c', 'd', 'e']
```

```
print(x[:3])
```

```
print(x[1:])
```

```
print(x[:])
```

 Python allows you to add a third parameter to the slice to control the step. This parameter allows

you to easily extract every nth element from the list. The syntax for slicing with a step is [start:end:step],

where step is an integer. x = ['a', 'b', 'c', 'd', 'e']

```
print(x[::2])
```

 # Every second element

```
print(x[::3])
```

 # Every third element

```
print(x[::-1])
```

 # Every element in reverse

```
print(x[::-2])
```

 # Every second element in reverse Getting a random item It can be useful to get a random item

of a collection. For lists and tuples you can use the choice() function of the random module: import random

```
print(random.choice([1, 2, 3, 4, 5]))
```

```
print(random.choice((1, 2, 3, 4, 5)))
```

 Sets The items in a set are unordered, which means that they have no

index. If you try to refer to an item in a set by using the indexing operator (square brackets notation) you

will get an error: x = {'cat', 'mouse', 'dog'}

```
print(x[0])
```

 # Error Dictionaries You can access the elements of a dictionary also using the indexing operator,

but in this case the indices are the keys of the dictionary, which are not necessarily integers. scores = {

```
'Alice': 0,
```

```
'Bob': 1,
```

```
'Eve': 2,
```

```
'Mallory': 3,
```

```
}
```

```
print(scores['Alice'])
```

 If there is no element with the given index then an error will occur: scores = {

```
'Alice': 0,
```

```
'Bob': 1,
```

```
'Eve': 2,
```

```
'Mallory': 3,
```

```
}
```

```
print(scores['Steve'])
```

 To allow for this you can use the get() method instead, which allows you to specify a

default value in case there is no matching index: scores = {

```
'Alice': 0,
```

```
'Bob': 1,
```

```
'Eve': 2,
```

```
'Mallory': 3,
```

```
}
```

```
print(scores.get('Steve', 'There is no such index'))
```

 Note that lists and tuples do not have a get() method.

For loopsIt is very common to loop through the items in a collection one-by-one. A good way to do so is to use a for loop, which is custom-made for this kind of thing:vowels = ['a', 'e', 'i', 'o', 'u']

```
for v in vowels:
```

```
    print(v)You can use this technique not just with lists but with tuples, sets and dictionaries too:for x in (1, 2, 3): # Tuple
```

```
    print(x)
```

```
for x in {1, 2, 3}: # Set
```

```
    print(x)
```

```
for x in {1:'a', 2:'b', 3:'c'}: # Dictionary
```

```
    print(x)Keep in mind that because sets are not ordered there is no guarantee in what order their items will be looped through. Try running the following piece of code a few times:for x in {'a', 'e', 'i', 'o', 'u'}:
```

```
    print(x)Also notice that when you loop through a dictionary it is the keys that get looped through:for x in {1: 'a', 2: 'e', 3: 'i', 4: 'o', 5: 'u'}:
```

```
    print(x)VariationsSometimes when you loop through a list you will want to use the index of an item as well as its value. You can do this by using the enumerate() function, which returns a collection of key-value pairs for you to loop through:vowels = ['a', 'e', 'i', 'o', 'u']
```

```
for index, vowel in enumerate(vowels):
```

```
    print('The vowel at index', index, 'is', vowel)You can do a similar thing with dictionaries by using the items() method of a dictionary:scores = {'Alice': 0, 'Bob': 1, 'Eve': 2, 'Mallory': 3}
```

```
for key, value in scores.items():
```

```
    print(f'{key} scored {value}')If you want to loop through just the values of a dictionary you can use the values() method:scores = {'Alice': 0, 'Bob': 1, 'Eve': 2, 'Mallory': 3}
```

```
for x in scores.values():
```

```
    print(x)Break and continueIn Week 1 you learned about using break and continue in a while loop. They can also be used in for loops.vowels = ['a', 'e', 'i', 'o', 'u']
```

```
print('Everything before o:')
```

```
for v in vowels:
```

```
    if v == 'o':
```

```
        break
```

```
    print(v)
```

```
print()
```

```
print('Everything except o:')
```

```
for v in vowels:
```

```
    if v == 'o':
```

```
        continue
```

```
    print(v)Nested loopsIf you are looping through a collection whose items are themselves collections then you might want to use nested loops.Suppose that you have a list of lists of numbers, and you want to add up all the numbers. You could do this using nested loops:lists = [[1, 2, 3], [3, 4, 5], [5, 6, 7]]
```

```
total = 0
```

```
for lst in lists:
```

```
    for num in lst:
```

```
        total += num
```

```
print(total)
```

Slide ID: 361953 | **Index:** 117 | **Title:** The range function | **Status:** unseen

A very common thing to do when programming is to loop through a sequence of numbers. Python's range() function is a very useful way of creating the numbers to loop through.for i in range(10):

```
    print(i)Note that range(n) returns n integers, from 0 to n-1. The number n is not included.Here's how you might use it:vowels = ['a', 'e', 'i', 'o', 'u']
```

```
for i in range(len(vowels)):
```

```
    print('The vowel at index', i, 'is', vowels[i])Although, you might find it more convenient in this case to use
```

```

the enumerate() function:vowels = ['a', 'e', 'i', 'o', 'u']
for i, value in enumerate(vowels):
    print('The vowel at index', i, 'is', value)
What range returns
It might seem like the range function returns these numbers as a list, the list [0, 1, ..., 10]. But actually it doesn't. It returns a special type of object called a range. You see this by checking its type:x = range(10)
print(type(x))
A range object is a method for generating each number as required, but not until it is required.If you'd like to use range() to get a list of numbers, you can just apply the list() function to it:x = list(range(10))
print(type(x))
print(x)
In a similar way, you could get a tuple or set of numbers.
Customising range
You can specify a starting value:for i in range(3, 10):
    print(i)
You can specify a step:for i in range(0, 10, 2):
    print(i)
You can work backwards by making the step negative:for i in range(10, 0, -2):
    print(i)
Nesting ranges
You might find yourself using range in nested for loops:for i in range(1, 11):
    for j in range(1, 11):
        print(f'{i} times {j} is {i*j}')
It has become conventional to use i, j, and k as loop counters.for i in range(2):
    for j in range(2):
        for k in range(2):
            print(i, j, k)

```

Slide ID: 361944 | **Index:** 118 | **Title:** Adding elements | **Status:** unseen

Adding to a list
You can add an item to the end of a list by using the append() method:x = ['a', 'b', 'c', 'd', 'e']
x.append('f')
print(x)
Notice that the append() method modifies the list in-place - you do not have to assign the result back to the variable. This is different from the string methods we saw last week, which do not modify strings in place, but return new values:x = "Hello"
x.upper() # x not changed
print(x)
x = x.upper() # x changed
print(x)
You can insert an item at a specified index by using the insert() method:x = ['a', 'b', 'c', 'd', 'e']
x.insert(2, 'x') # Insert at index 2
print(x)
You can achieve the same thing by replacing the empty slice from 2 to 2:x = ['a', 'b', 'c', 'd', 'e']
x[2:2] = 'x' # Insert at index 2
print(x)
You can extend a list with the items from another list by using the extend() method:x = ['a', 'b', 'c', 'd', 'e']
y = ['x', 'y', 'z']
x.extend(y)
print(x)
Notice that the extend() method also modifies the list in-place - you do not have to assign the result back to the variable.Also notice that extending by a list is different from appending a list. Compare the result of extending (above) with the result of appending (below):x = ['a', 'b', 'c', 'd', 'e']
y = ['x', 'y', 'z']
x.append(y)
print(x)
You can also extend a list with the items from another list using the + operator:x = ['a', 'b', 'c', 'd', 'e']
y = ['x', 'y', 'z']
x = x + y
print(x)
Notice that the + operator does not modify the list in-place - you have to assign the result back to the variable.
Adding to a tuple
Because tuples are immutable you cannot add items to them.
Adding to a set
You can add a single item to a set by using the set's add() method. If the item is already in the set, Python quietly ignores the request.letters = {'a', 'b'}
letters.add('c')
print(letters)
letters.add('c') # No error, just not added

```

print(letters)You can add multiple items to a set by using the set's update() method:vowels = set()
vowels.update('a', 'e')
print(vowels)Adding to a dictionaryYou can add an item to a dictionary by specifying a value for a new key.
If the key already exists, the value for that key will be updated.x = {1: 'a', 2: 'b'}
x[3] = 'c' # Item added
print(x)
x[3] = 'd' # Item updated
print(x)

```

Slide ID: 361945 | **Index:** 119 | **Title:** Removing elements | **Status:** unseen

```

Removing from a listYou can remove an item at a specific index using a del statement:x = ['a', 'b', 'c', 'd', 'e']
del x[1]
print(x)You can do the same with a slice of the list:x = ['a', 'b', 'c', 'd', 'e']
del x[1:3]
print(x)You can also remove a slice by setting it to the empty list:x = ['a', 'b', 'c', 'd', 'e']
x[1:3] = [] # Items removed
print(x)This doesn't work for individual items:x = ['a', 'b', 'c', 'd', 'e']
x[0] = [] # Item not removed - replaced by the empty list
print(x)You can also remove an item at a specific index using the pop() method. If pop() isn't given an
index then the last item will be removed. The removed element is returned.x = ['a', 'b', 'c', 'd', 'e']
print(x.pop(2)) # Removes and returns item at index 2
print(x)
print(x.pop()) # Removes and returns the last item
print(x)You can remove the first item with a given value by using the remove() method:x = ['a', 'b', 'c', 'd', 'e', 'c']
x.remove('c') # Only first occurrence is removed
print(x)You can remove all items from a list by using the clear() method:x = ['a', 'b', 'c', 'd', 'e']
x.clear() # All items removed
print(x)Removing from a tupleBecause tuples are immutable you cannot remove items from
them.Removing from a setBecause a set is not ordered you cannot remove elements by index, but you
can remove them by value, using the set's remove() or discard() methods. If the item is not in the set then
using remove() will cause an error, but using discard() will not:vowels = {'a', 'e', 'i', 'o', 'u'}
vowels.remove('a') # Removes 'a'
print(vowels)
vowels.discard('f') # No error
print(vowels)
vowels.remove('f') # ErrorYou can remove all items from a set by using the set's clear() method:vowels = {'a', 'e', 'i', 'o', 'u'}
vowels.clear() # Remove all items
print(vowels)Removing from a dictionaryYou can remove an element of a dictionary by key using del or
pop():scores = {'A': 0, 'B': 1, 'E': 2, 'M': 3}
del scores['B'] # Remove item whose key is 'B'
print(scores)
scores.pop('M') # Remove item whose key is 'M', and return its value
print(scores)You can remove all elements using clear():scores = {'A': 0, 'B': 1, 'E': 2, 'M': 3}
scores.clear() # Remove all items
print(scores)

```

Slide ID: 361946 | **Index:** 120 | **Title:** Modifying elements | **Status:** unseen

Modifying list elements You can change the value of an item in a list by referring to it and then assigning it a new value:

```
x = ['a', 'b', 'c', 'd', 'e']
```

```
x[0] = 'z' # Assign a new value
```

```
print(x) You can also assign to a slice with a sequence: x = ['a', 'b', 'c', 'd', 'e']
```

```
x[0:1] = ['x', 'y'] # Assign new values
```

```
print(x) Modifying tuple elements Since tuples are immutable, you cannot change which objects are in the tuple: x = (0, 2, 3)
```

```
x[0] = 1 # Error - cannot change which objects are in the tuple But if one of those objects is itself mutable, then you can change the nature of that object. Suppose the first element of a tuple is a list, for example. You can change the elements of this list, even though the list is part of a tuple. That's because you're not changing which objects are in the tuple - you're just changing the nature of one of those objects. It's not a good idea to do this, though - it goes against the spirit of using tuples to signal that your data should not change.
```

Modifying set elements You can't change an element of a set, but you remove it and then add a different element:

```
x = {1, 2, 3}
```

```
x.remove(3)
```

```
x.add(4)
```

```
print(x) Modifying dictionary elements Updating an item in a dictionary is similar to updating an item in a list.
```

```
scores = {
```

```
'Alice': 0,
```

```
'Bob': 1,
```

```
'Eve': 2,
```

```
'Mallory': 3,
```

```
}
```

```
scores['Bob'] = 900
```

```
scores['Alice'] += 1
```

```
print(scores)
```

Slide ID: 361947 | Index: 121 | Title: Sorting elements | Status: unseen

Sorting a list You can sort the elements of a list by using the `sort()` method, which orders the items by comparing their values using the operator. Note that this method sorts the list in-place - it does not return a new list.

```
x = [1, 5, 4, 2, 3]
```

```
x.sort()
```

```
print(x)
```

```
x = ['c', 'a', 'e', 'b', 'd']
```

```
x.sort()
```

```
print(x) You can sort the elements in descending order by passing the keyword argument reverse = True: x = [1, 5, 4, 2, 3]
```

```
x.sort(reverse = True)
```

```
print(x) You could also achieve this by sorting them in ascending order and then reversing the list, using the reverse() method: x = [1, 5, 4, 2, 3]
```

```
x.sort()
```

```
x.reverse()
```

```
print(x) Note that reversing a list is not the same thing as sorting it in descending order: x = [1, 5, 4, 2, 3]
```

```
x.sort(reverse = True)
```

```
print(x)
```

```
x = [1, 5, 4, 2, 3]
```

```
x.reverse()
```

```
print(x) Also note that if you have a list of lists the reverse() method only reverses the topmost level of lists: numbers = [[1, 2, 3], [3, 4, 5], [5, 6, 7]]
```

```
numbers.reverse()
```

```
print(numbers) Sorting uppercase and lowercase Somewhat surprisingly, Python sorts all uppercase
```

characters before all lowercase characters. So 'B' comes before 'a'. Try it: `x = ['a', 'A', 'b', 'B', 'c', 'C']`

```
x.sort()
```

`print(x)` The reason for this is that Python sorts characters according to their ASCII numeric value, and the ASCII values of uppercase characters are lower than the ASCII values of lowercase characters. You can see this, by using the `ord()` function: `print(ord('a'))`

```
print(ord('A'))
```

```
print(ord('b'))
```

```
print(ord('B'))
```

```
print(ord('c'))
```

`print(ord('C'))` Sorting with a function Suppose you have a list of words and you want to sort them by length.

If you use the bare `sort()` method then you will get the wrong result - it will sort them alphabetically: `x =`

```
['dog', 'chicken', 'mouse', 'horse', 'goat', 'donkey']
```

```
x.sort()
```

`print(x)` In this case you need to specify a key, which is a function that returns, for each item in the list, the value that you'd like to sort the item by. In this case we'd like to sort items by length, so we can use the

`len()` function: `x = ['dog', 'chicken', 'mouse', 'horse', 'goat', 'donkey']`

```
x.sort(key = len)
```

`print(x)` Later you will learn how to define your own functions. This will allow you to do even more

sophisticated sorting. What about tuples, sets and dictionaries? Since tuples are immutable they cannot be

sorted, so tuples do not have a `sort()` method. Since the elements in a set are not ordered it doesn't make

sense to sort them (you cannot be guaranteed of getting them back in any particular order), so sets do not

have a `sort()` method. Although the elements of a dictionary are ordered, they are always in insertion order

- the order in which they were inserted into the dictionary. So it doesn't make sense to sort a dictionary,

and dictionaries do not have a `sort()` method. You can, however, create a sorted list from these things,

using Python's `sorted()` function. The sorted function The `sorted()` function takes a collection and returns a

list of its items sorted in ascending order. It leaves the original collection unchanged (so it can be used on

tuples). Note that in the case of a dictionary it returns the keys of the dictionary, sorted. `x = (1, 5, 4, 2, 3)` #

Tuple

```
print(sorted(x)) # Returns a new list, sorted
```

```
x = {1, 5, 4, 2, 3} # Set
```

```
print(sorted(x)) # Returns a new list, sorted
```

```
x = {1:'f', 5:'a', 4:'b', 2:'d', 3:'c'} # Dictionary
```

```
print(sorted(x)) # Returns a new list, from the keys, sorted
```

If you want the resulting list in descending order, pass the argument `reverse = True`: `x = (1, 5, 4, 2, 3)`

```
print(sorted(x, reverse = True))
```

```
x = {1, 5, 4, 2, 3} # Set
```

```
print(sorted(x, reverse = True))
```

```
x = {1:'f', 5:'a', 4:'b', 2:'d', 3:'c'}
```

```
print(sorted(x, reverse = True))
```

Slide ID: 361948 | **Index:** 122 | **Title:** Joining elements | **Status:** unseen

You can join the elements of a collection into a string, as long as they are themselves strings. The method by which you do this is somewhat counterintuitive: `x = ['a', 'b', 'c', 'd', 'e']`

```
s = ','.join(x)
```

`print(s)` To join the elements of the list into a string using commas to separate the items, you call the `join()` method on the comma string, and supply the list as an argument. You can use whatever separator you

want: `x = ['a', 'b', 'c', 'd', 'e']`

```
print(' '.join(x))
```

```

print(' '.join(x))
print('--'.join(x))
print(' then '.join(x)) This technique works for tuples, sets and dictionaries too (in the case of dictionaries it
is the keys that get joined, and they must be strings): x = ('a', 'b', 'c', 'd', 'e') # Tuple
print('-'.join(x))

x = {'a', 'b', 'c', 'd', 'e'} # Set
print('-'.join(x))

x = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5} # Dictionary
print('-'.join(x))

```

Slide ID: 361949 | **Index:** 123 | **Title:** Special string operations | **Status:** unseen

Strings as tuples In Python we can think of a string as a tuple of characters. This means that you can access the characters in a string in the same ways that you can access the items in a tuple, using the [] indexing operator. You can select individual characters this way, and you can select slices of characters, which are substrings: x = 'Hello'

```

print(x[1]) # Individual character
print(x[1:3]) # Slice
print(x[2:-1]) # Slice But you cannot reassign individual characters: x = 'Hello'
x[1] = 'J'
print(x) # Error Splitting strings One of the most useful methods available for string objects is the split()
method. This breaks a string into a list of substrings that are separated by a delimiter. y = 'The cat sat on
the mat'
print(y.split(' ')) # Split using ' ' as the delimiter

```

```

z = '12:30:45'
print(z.split(':')) # Split using ':' as the delimiter Note that the delimiter is not included in the substrings.

```

Slide ID: 361950 | **Index:** 124 | **Title:** Special set operations | **Status:** unseen

There are special operations that you can perform on sets. Union The union of two sets is the set of items that belong to either or both sets: You can get the union of two sets using either the | operator or the union() method: evens = {2, 4, 6, 8}

```

primes = {2, 3, 5, 7}
print(evens | primes)
print(evens.union(primes))
print(primes.union(evens)) Intersection The intersection of two sets is the set of items that belong to both
sets: You can get the intersection of two sets using either the & operator or the intersection() method: evens
= {2, 4, 6, 8}
primes = {2, 3, 5, 7}
print(evens & primes)
print(evens.intersection(primes))
print(primes.intersection(evens)) Difference The difference between set A and set B is the set of items that
belong to A but not B: You can get the difference between two sets using either the - operator or the
difference() method: evens = {2, 4, 6, 8}
primes = {2, 3, 5, 7}
print(evens - primes)
print(evens.difference(primes))
print(primes - evens)
print(primes.difference(evens)) Symmetric difference The symmetric difference between set A and set B is
the set of items that belong to A but not B, or to B but not A. You can get the symmetric difference between

```

```

two sets using the symmetric_difference() method:evens = {2, 4, 6, 8}
primes = {2, 3, 5, 7}
print(evens.symmetric_difference(primes))
print(primes.symmetric_difference(evens))Comparing setsTwo sets are disjoint when they have no
elements in common.You can check whether two sets are disjoint using the isdisjoint() method:evens = {2,
4, 6, 8}
primes = {2, 3, 5, 7}
print(evens.isdisjoint(primes))
print(primes.isdisjoint(evens))A set A is a subset of a set B when every element of A is also an element of
B. We also say, in this case, that B is a superset of A.You can check whether one set is a subset of
another using either the operator or the issubset() method:A = {2, 3}
B = {2, 3, 6, 7}
C = {4, 5, 6, 7}
print(A.issubset(B))
print(A Similarly, you can check whether one set is a superset of another using either the >= operator or
the issuperset() method:A = {2, 3}
B = {2, 3, 6, 7}
C = {4, 5, 6, 7}
print(B.issuperset(A))
print(B >= A)
print(C.issuperset(A))
print(C >= A)

```

Slide ID: 361952 | **Index:** 125 | **Title:** Comprehensions | **Status:** unseen

List comprehensionsSuppose you have a list of words and you'd like a list of their lengths. You could get it by using a for loop:words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
lengths = []
for word in words:
lengths.append(len(word))
print(lengths)Python provides a more elegant way to do this - a list comprehension:words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
lengths = [len(word) for word in words]
print(lengths)The expression [len(word) for word in words] is the list comprehension. Note that it is an expression, rather than a block of statements.You can add a condition to the comprehension. Suppose you only want to include words that are more than three letters long. Then you could use the following:words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
lengths = [len(word) for word in words if len(word) > 3]
print(lengths)This is a very useful way of filtering out elements from a list. Here is another example:nums = [2, 12, 4, 2, 9, 10, 11, 1, 15, 23]
nums = [x for x in nums if x >= 10]
print(nums)In a previous example we used nested for loops to sum the numbers in a list of lists of numbers. We can do it using list comprehensions:lists = [[1, 2, 3], [3, 4, 5], [5, 6, 7]]
total = sum([sum(x) for x in lists])
print(total)Set comprehensionsIf you would like a set of lengths rather than a list of lengths then you could use a set comprehension, which is just the same but with curly brackets:words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
unique_lengths = {len(word) for word in words}
print(unique_lengths)Note that duplicate values are automatically removed from the set.Dictionary comprehensionsYou can also create a dictionary using a dictionary comprehension. Here is a slightly different example:words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
words_with_lengths = {word: len(word) for word in words}
print(words_with_lengths)What about a tuple comprehension?Alas, there is no tuple comprehension. You

can use the same sort of construction, but it will give you a generator, not a tuple (more about generators next week).

```
words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
lengths = (len(word) for word in words)
print(lengths)
```

Slide ID: 361954 | **Index:** 126 | **Title:** Files as lists | **Status:** unseen

Just as we can think of a string as a tuple of characters, we can think of a file as a list of lines. Python provides two useful methods that take advantage of this. **Readlines** We saw in Week 1 that you can read the content of a text file into a string, using the `read()` method. The `readlines()` method allows you to read the contents into a list of lines instead. This allows you to use a `for` statement to loop through the lines of a file one-by-one.

Add some lines to a file

```
with open('myfile', 'w') as file:
    file.write('This is the first line\n')
    file.write('This is the second line\n')
    file.write('This is the third line')
```

Inspect the results

```
with open('myfile', 'r') as file:
    lines = file.readlines()
    print(f'The file contains {len(lines)} lines.')
    for line in lines:
```

`print(f'Line: {line}')` Notice the newline character is included in each line. You will typically want to remove that using the `strip()` method of a string. **Writelines** The `writelines()` method lets you write multiple lines to a file, given as a list of strings. Note that each line in the list should be terminated with a newline character (`\n`) otherwise `writelines()` will concatenate the content onto a single line.

```
LINES = [
    'This is the first line\n',
    'This is the second line\n',
    'This is the third line'
]
```

Write these lines to a file

```
with open('myfile', 'w') as file:
    file.writelines(LINES)
```

Inspect the results

```
with open('myfile', 'r') as file:
    print(file.read())
```

Reading CSV files You might sometimes want to read a CSV file. Python provides a convenient way of reading a CSV file and parsing it into a list of lists. You need to import the `csv` module to use this feature of Python.

Create a CSV file to experiment with

```
with open('myfile', 'w') as file:
    file.write('a,b,c\n')
    file.write('d,e,f\n')
    file.write('g,h,i\n')
```

Read the CSV file

```
with open('myfile', 'r') as file:
    lst = list(csv.reader(file))
    print(lst)
```

Slide ID: 361955 | **Index:** 127 | **Title:** Dates and times | **Status:** unseen

This is not related to collections, but now is a good time to talk about Python's facilities for working with dates and times. When working with real-world data you often have to deal with dates and times. Date and times can be tricky to work with, because people format and present them in different ways. Consider, for example, a date written as "02-03-1998". Does this represent 2nd March 1998, or 3rd February 1998? In Australia it would be the former, but in the US it would be the latter. Python has a datetime library that defines datetime, date, and time types. These provide a uniform and comprehensive way to handle dates and times. The datetime type is the most flexible and thus the most commonly used. There is also a timedelta type, which is used to work with durations (i.e., time intervals). To use these you need to import them: from datetime import datetime, date, time, timedelta. You don't need to import them all - just the ones you will be using. Usually your goal will be to convert a string or integer representation of a date or time into a datetime object, apply whatever processing you need to the object, and then use a formatting function to convert it back to a traditional format. Creating a datetime object There are several ways to create a datetime object. You can directly construct it by providing the year, month, and day, and, optionally, the time and timezone.

```
from datetime import datetime
```

```
dt = datetime(  
    year = 1968, month = 6, day = 24,  
    hour = 5, minute = 30, second = 0  
)
```

print(dt) You can also construct it from a string. In this case you need to tell Python what format the string uses:

```
from datetime import datetime
```

```
dt = datetime.strptime('24-06-1968, 05:30:00', '%d-%m-%Y, %H:%M:%S')
```

print(dt) The trickiest part about this is remembering what the formatting codes are. You'll probably find yourself looking them up quite often. Here are the main ones:

%Y Four-digit year (1968)

%y Two-digit year (68)

%B Full month name (June)

%b Abbreviated month name (Jun)

%m Two-digit month number (01-12)

%A Full day name (Monday)

%a Abbreviated day name (Mon)

%d Two-digit day number (01-31)

%H Two-digit hour (00-23)

%I Two-digit hour (00-12)

%M Two-digit minute (00-59)

%S Two-digit second (00-59)

%p AM/PM If you want a datetime object that represents the current date and time you can use the now method:

```
from datetime import datetime
```

```
dt = datetime.now()
```

print(dt) Unix timestamp You can also construct a datetime object from a Unix timestamp. This is one of the most common representations of time. It represents the time as a numerical value - the number of seconds since the Unix epoch, which was at 00:00:00 on Thursday, 1 January 1970 UTC. You can get the current Unix timestamp using the time.time function.

```
from time import time
```

```
print(time()) To create a datetime object from a Unix timestamp you can use
```

```
datetime.fromtimestamp().from datetime import datetime
```

```
x = datetime.fromtimestamp(1565315907)
```

print(x) Extracting the components of a datetime object Once you have a datetime object you can extract its individual components:

```
from datetime import datetime
```

```
dt = datetime(  
    year = 1968, month = 6, day = 24,
```

```
hour = 5, minute = 30, second = 0
)
```

```
print(dt.year)
print(dt.month)
print(dt.day)
print(dt.hour)
```

```
print(dt.minute)
```

```
print(dt.second)You can also extract a date object or a time object from a datetime object:from datetime
import datetime
```

```
dt = datetime(
year = 1968, month = 6, day = 24,
hour = 5, minute = 30, second = 0
)
```

```
print(dt.date())
```

```
print(dt.time())Formatting a datetime object as a stringA very common thing to do is to present a datetime
object in a certain format. You can use the strftime method, to do this. You need to specify the format you
would like, using the same formatting codes as listed above.from datetime import datetime
```

```
dt = datetime(
year = 1968, month = 6, day = 24,
hour = 5, minute = 30, second = 0
)
```

```
print(dt.strftime('%d %B %Y, at %I:%M %p'))Operating on datetime objectsOne of the most useful features
of datetime object...
```

Slide ID: 361956 | **Index:** 128 | **Title:** Further reading | **Status:** unseen

You might find the following helpful:[The Python Tutorial at w3schools.com](https://www.w3schools.com/python/)

Lesson 29: Week 5 - Notes 7 Euler's Sieve

Lesson ID: 53319 | Created: 2024-05-27T01:25:39.813331+10:00 | State: active | Status: unattempted | Slide Count: 5

Slides

Slide ID: 362018 | Index: 3 | Title: Euler's sieve | Status: unseen

Slide ID: 362019 | Index: 4 | Title: Euler's sieve | Status: unseen

Slide ID: 362020 | Index: 5 | Title: euler_sieve_v1.py | Status: unseen

Slide ID: 362021 | Index: 6 | Title: euler_sieve_v2.py | Status: unseen

Slide ID: 362022 | Index: 7 | Title: Euler's sieve | Status: unseen

Lesson 30: Week 7 - Overview

Lesson ID: 53320 | Created: 2024-05-27T01:25:40.175001+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 362023 | Index: 9 | Title: Week 7 Tuesday To Do List | Status: unseen

Week 7 Tuesday To Do List
Admin/Tips Assignment 2 worth 13 marks will be released today @ 7.15pm
Quiz 4 worth 4 marks is due Week 7 Thursday 11/7/24 @ 9pm
Content Notes 6 Eratosthenes' Sieve (cont'd) - 2nd optimised approach
Notes 7 Euler's Sieve Plotting with Matplotlib
Discuss Quiz 4 Discuss Assignment 2
Useful Links Sieve of Euler [https://programmingpraxis.com/2011/02/25/sieve-of-euler/zip\(\)](https://programmingpraxis.com/2011/02/25/sieve-of-euler/zip/)
Function https://www.w3schools.com/python/ref_func_zip.asp zip_longest() from itertools module https://www.geeksforgeeks.org/python-itertools-zip_longest/

Slide ID: 362024 | Index: 11 | Title: Week 7 Thursday To Do List | Status: unseen

Week 7 Thursday To Do List
Admin/Tips Sample Final Exam Questions released
Quiz 5 worth 4 marks will be released today @ 7.15pm
Quiz 4 worth 4 marks is due today @ 9pm
Assignment 2 worth 13 marks is due Week 11 Monday 5/8/24 @ 10am
Content Week 7 Notes 8 Card shuffling Unicode character set List comprehension String join() method List extend() method slices getsizeof() from the sys module Command line arguments
Week 7 Notes 9 US Social Security Data on Given Names Files and folders manipulation csv built-in module os built-in module
Discuss Quiz 4 Discuss Assignment 2 Discuss Quiz 5 Useful Links
Playing cards in Unicode https://en.wikipedia.org/wiki/Playing_cards_in_Unicode defaultdict (same as dict but never raises a KeyError) <https://www.geeksforgeeks.org/defaultdict-in-python/> Python os Module https://www.w3schools.com/python/module_os.asp

Lesson 31: Week 7 - Notes 8 Card Shuffling

Lesson ID: 53321 | **Created:** 2024-05-27T01:25:40.213174+10:00 | **First Viewed:** 2025-09-25T11:15:52.816024+10:00 | **Last Viewed Slide ID:** 362026 | **State:** active | **Status:** attempted | **Slide Count:** 5

Slides

Slide ID: 362026 | **Index:** 2 | **Title:** Card shuffling | **Status:** completed

Slide ID: 362027 | **Index:** 3 | **Title:** Useful link about playing cards in Unicode | **Status:** unseen
Playing cards in Unicodehttps://en.wikipedia.org/wiki/Playing_cards_in_Unicode

Slide ID: 362028 | **Index:** 4 | **Title:** Card shuffling | **Status:** unseen

Slide ID: 362029 | **Index:** 5 | **Title:** card_shuffling.py | **Status:** unseen

Slide ID: 362030 | **Index:** 6 | **Title:** Card shuffling | **Status:** unseen

Lesson 32: Week 10 - Overview

Lesson ID: 53330 | Created: 2024-05-27T01:25:43.93153+10:00 | State: active | Status: unattempted | Slide Count: 3

Slides

Slide ID: 398949 | Index: 9 | Title: Constructive Feedback | Status: unseen

Slide ID: 362068 | Index: 10 | Title: Week 10 Tuesday To Do List | Status: unseen

Week 10 Tuesday To Do ListAdmin/TipsmyExperience SurveyAssignment 2 worth 13 marks is due Week 11 Monday 5/8/24 @ 10amFinal Exam worth 50 marks to be held on Wednesday 21 August 2024Morning (11:15-13:30)Afternoon (13:20-16:10)Session preference form
<https://cgi.cse.unsw.edu.au/~exam/24T2/seating/register.cgi>Closes Midday Week 10 Friday 2nd August 2024Seating allocations together with exact time and location released Mid Week 11 (Study Break)More on Final Exam in our last review class on Thursday Week 10ContentWeek 10 Notes 15 Context Free GrammarsThe Unicode of the alphabet character ϵ is 03B5 (hexadecimal) or 949 (decimal)`chr(949)` or `chr(0x03B5)`Week 10 Notes 16 Three Special Perfect SquaresSearching and SortingUseful LinksContext-free grammarhttps://en.wikipedia.org/wiki/Context-free_grammarContext-sensitive grammarhttps://en.wikipedia.org/wiki/Context-sensitive_grammar

Slide ID: 362069 | Index: 12 | Title: Week 10 Thursday To Do List | Status: unseen

Week 10 Thursday To Do ListAdmin/TipsmyExperience SurveyAssignment 2 worth 13 marks is due Week 11 Monday 5/8/24 @ 10amFinal Exam worth 50 marks to be held on Wednesday 21 August 2024Morning (11:15-13:30)Afternoon (13:20-16:10)Session preference form
<https://cgi.cse.unsw.edu.au/~exam/24T2/seating/register.cgi>Closes Midday Week 10 Friday 2nd August 2024Seating allocations together with exact time and location released Mid Week 11 (Study Break)Final Exam EnvironmentContentSearching and Sorting (cont'd)ReviewSolve Sample Exam Question 8Useful Linksitertools.permutations<https://docs.python.org/3/library/itertools.html#itertools.permutations>

Lesson 33: Week 4 - Overview

Lesson ID: 53310 | Created: 2024-05-27T01:25:37.901929+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 361975 | Index: 9 | Title: Week 4 Tuesday To Do List | Status: unseen

Week 4 Tuesday To Do List
Admin/Tips
How to Use Jupyter Notebook: A Beginner's Tutorial (added to Ed Lessons General)
For an example of exam questions using doctest, see Exam Questions with Doctest
Quiz 1 marks and sample solution released
Quiz 2 worth 4 marks is due
Week 4 Thursday @ 9pm
Assignment 1 worth 13 marks is due
Week 7 Monday @ 10am
Content
Briefly continue discussing Week 3 - Notes 2 The Monty Hall Problem
Modules and Recursion
Week 4 - Notes 3 The Towers of Hanoi
Useful Links
doctest — Test interactive Python examples (will be used for final exam questions)
<https://docs.python.org/3/library/doctest.html>
timeit — Measure execution time of small code snippets
<https://docs.python.org/3/library/timeit.html>

Slide ID: 361976 | Index: 11 | Title: Week 4 Thursday To Do List | Status: unseen

Week 4 Thursday To Do List
Admin/Tips
Quiz 3 worth 4 marks will be released today @ 7.15pm
Quiz 2 worth 4 marks is due today @ 9pm
Assignment 1 worth 13 marks is due
Week 7 Monday @ 10am
Content
Continue Week 4 - Notes 3 The Towers of Hanoi
Week 4 - Notes 4 The Fibonacci Sequence
yield statement
functools module
itertools module
islice() function from itertools module
lru_cache() function from functools module
Week 4 - Notes 5 From Decimal Expansions to Reduced Fractions
Using _ as thousands separator, for instance, 10_000
gcd() (greatest common divisor) function
, //, and % operators
divmod() function
bool() function when condition is not a boolean expression
Unpacking tuple elements into function arguments
Useful Links
When to use yield instead of return in Python?
<https://www.geeksforgeeks.org/use-yield-keyword-instead-return-keyword-python/>
Python next() Function
https://www.w3schools.com/python/ref_func_next.asp
functools module — Higher-order functions and operations on callable objects
<https://docs.python.org/3/library/functools.html>
itertools module — Functions creating iterators for efficient looping
<https://docs.python.org/3/library/itertools.html>

Lesson 34: Week 4 - Modules and Recursion

Lesson ID: 53311 | Created: 2024-05-27T01:25:37.937984+10:00 | State: active | Status: unattempted | Slide Count: 6

Slides

Slide ID: 361977 | Index: 160 | Title: Creating modules | Status: unseen

You might find yourself creating your own functions or classes (to be discussed later) that you would like to use in various programs. Rather than copying and pasting their definitions into each program it is far better to put them into a module, and then import that module whenever you need to. Just as you can put code into functions, you can put functions (and classes) into modules. You can create your own modules by writing Python code and saving it in a file with a ".py" extension. Then you can import these modules into your code whenever you need them. For example, suppose you have a module called "hello.py" which contains the following code:

```
def say_hello():
```

```
    print('Hello world!')
```

You can import and use this module just like any other module (do not include the ".py" part of the file name):

```
from hello import say_hello
```

```
say_hello()
```

Importing quietly Sometimes you might have code in your module that is outside of a function or class. For example, suppose the contents of "hello.py" are this:

```
def say_hello():
```

```
    print('Hello world!')
```

```
    print("Python is fun!")
```

When you import this module into your program, the Python interpreter executes the code in the module, including line 3. This will generate output in your program that you probably don't want: Python is fun!

```
Hello world!
```

Python is fun! is a consequence of line 3 of your module, which was executed when it was imported. It is better if your module is imported quietly. You make sure this happens by modifying your module as follows:

```
def say_hello():
```

```
    print('Hello world!')
```

```
if __name__ == '__main__':
```

```
    print("Python is fun!")
```

When a file executes it contains a special variable `__name__` that indicates whether the file has been executed as a standalone file or as an imported module. In the former case, the value of this variable is `'__main__'`, otherwise it is the name of the module, 'hello'. This gives you a way of checking whether your module was executed as a standalone file or as an imported module, and modifying its behaviour accordingly. In the code above, line 4 does the check, and `print("Python is fun!")` is considered only if the module is being executed as a standalone file. When you import the module, it will not be executed.

Slide ID: 361978 | Index: 161 | Title: Modules Example | Status: unseen

Slide ID: 361979 | Index: 162 | Title: Recursion | Status: unseen

You can get a function to call itself. This is a very powerful technique known as recursion. A function that calls itself is said to be a recursive function. Suppose you want to create a function that calculates, for a given number n , the factorial of n , which is $1 \times 2 \times 3 \times \dots \times n$. For example, the factorial of 6 is $1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$. You can do this without recursion, using a loop:

```
def factorial(n):
```

```
    result = 1
```

```
    for x in range(1, n+1):
```

```
        result = result * x
```

```
    return result
```

```
number = input('Enter a number: ')
```

```
number = int(number)
```

print(f'The factorial of {number} is {factorial(number)}.')Alternatively, you can do it with recursion.The trick is to notice that the factorial of n can be obtained by multiplying n by the factorial of (n-1).Hence, we can do this:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1) # Recursion - the function calls itself
```

```
number = input('Enter a number: ')
```

```
number = int(number)
```

print(f'The factorial of {number} is {factorial(number)}.')You need to be careful, when defining a recursive function, that it does not keep calling itself forever. That is why the recursive function above checks the value of n and decides what to do accordingly. There are two cases: n = 1. If n is 1 then the function doesn't call itself - it just returns 1. This is sometimes called the base case. Otherwise. If n is not 1 then the function calls itself. This is sometimes called the recursive case. The base case is important - it stops the function from calling itself forever. Notice what happens without it:

```
def factorial(n):
    return n * factorial(n-1)
```

```
number = input('Enter a number: ')
```

```
number = int(number)
```

print(f'The factorial of {number} is {factorial(number)}.')It is very common for a recursive function to distinguish these two cases - the base case and the recursive case. When you define a recursive function, check that you have your cases covered.

Slide ID: 361980 | **Index:** 163 | **Title:** More Recursive Examples | **Status:** unseen

Adding Two Numbers

```
def add(a,b):
    if b == 0:
        return a
    return add(a,b-1) + 1
```

print(add(4,9))Multiply by 4

```
def mult4(n):
    if n == 1:
        return 4
    return mult4(n-1) + 4
```

```
print(mult4(1))
```

print(mult4(5))Factorial - Better Version

```
def fact(n):
    if n
```

Slide ID: 361981 | **Index:** 164 | **Title:** Fibonacci Iterative, Recursive, and Memoise Versions | **Status:** unseen

Slide ID: 361982 | **Index:** 165 | **Title:** Further reading | **Status:** unseen

You might find the following helpful: [The Python Tutorial at w3schools.com](https://www.w3schools.com/python/)

Lesson 35: Week 4 - Notes 4 The Fibonacci Sequence

Lesson ID: 53312 | Created: 2024-05-27T01:25:38.214967+10:00 | State: active | Status: unattempted | Slide Count: 4

Slides

Slide ID: 361983 | Index: 1 | Title: The Fibonacci sequence | Status: unseen

Slide ID: 361984 | Index: 2 | Title: The Fibonacci sequence | Status: unseen

Slide ID: 361985 | Index: 3 | Title: fibonacci.py | Status: unseen

Slide ID: 361986 | Index: 4 | Title: The Fibonacci sequence | Status: unseen

Lesson 36: Cryptography

Lesson ID: 53360 | Created: 2024-05-27T01:25:56.984162+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 384762 | Index: 1 | Title: Cryptography | Status: unseen

Slide ID: 362187 | Index: 2 | Title: Cryptography | Status: unseen

Lesson 37: Week 9 - Notes 13 Quadratic Equations

Lesson ID: 53328 | Created: 2024-05-27T01:25:43.024866+10:00 | State: active | Status: unattempted | Slide Count: 7

Slides

- Slide ID: 362057 | Index: 1 | Title: Week 9 - Quadratic equations | Status: unseen
- Slide ID: 362058 | Index: 2 | Title: Week 9 - Quadratic equations | Status: unseen
- Slide ID: 362059 | Index: 3 | Title: quadratic_equation_v1.py | Status: unseen
- Slide ID: 362060 | Index: 4 | Title: quadratic_equation_v2.py | Status: unseen
- Slide ID: 362061 | Index: 5 | Title: quadratic_equation_v3.py | Status: unseen
- Slide ID: 362062 | Index: 6 | Title: quadratic_equation_v4.py | Status: unseen
- Slide ID: 362063 | Index: 7 | Title: quadratic_equation_v5.py | Status: unseen

Lesson 38: Week 10 - Notes 15 Context Free Grammars

Lesson ID: 53331 | Created: 2024-05-27T01:25:43.971614+10:00 | State: active | Status: unattempted | Slide Count: 5

Slides

Slide ID: 362070 | Index: 1 | Title: Week 10 - Context free grammars | Status: unseen

Slide ID: 362071 | Index: 2 | Title: Useful links about context-free grammars | Status: unseen

Context-free grammarhttps://en.wikipedia.org/wiki/Context-free_grammarContext-sensitive grammarhttps://en.wikipedia.org/wiki/Context-sensitive_grammar

Slide ID: 362072 | Index: 3 | Title: Week 10 - Context free grammar | Status: unseen

Slide ID: 362073 | Index: 4 | Title: CFG Example | Status: unseen

$S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$ is same as $S \rightarrow aSaS \rightarrow bSbS \rightarrow aS \rightarrow bS \rightarrow \epsilon$ S is the starting symbol and ϵ is the empty word
Does this word 'abbabba' belong to the language of the grammar above?
 $S \rightarrow aSa \rightarrow abSba \rightarrow abbSbba \rightarrow abbabba$ YES
abab? NO
elle, radar, level, and refer are examples of palindrome

Slide ID: 362074 | Index: 5 | Title: context_free_grammar.py | Status: unseen

Lesson 39: Week 9

Lesson ID: 53340 | Created: 2024-05-27T01:25:50.005439+10:00 | State: active | Status: unattempted | Slide Count: 5

Slides

Slide ID: 362134 | Index: 2 | Title: Exercise 1: Canonical coin systems | Status: unseen

Write a program that prompts the user for an amount, and outputs the minimal number of coins needed to yield that amount, as well as the detail of how many coins of each value are used. The available coins have a face value which is one of $\$1$, $\$2$, $\$5$, $\$10$, $\$20$, $\$50$, and $\$100$. Insert your code into `canonical_coin_systems.py`. Here are examples of interactions: `$ python3 canonical_coin_systems.py`
Input the desired amount: 10

1 banknote is needed.

The detail is:

$\$10$: 1

`$ python3 canonical_coin_systems.py`

Input the desired amount: 739

12 banknotes are needed

The detail is:

$\$100$: 7

$\$20$: 1

$\$10$: 1

$\$5$: 1

$\$2$: 2

`$ python3 canonical_coin_systems.py`

Input the desired amount: 35642

359 banknotes are needed

The detail is:

$\$100$: 356

$\$20$: 2

$\$2$: 1

Slide ID: 362135 | Index: 3 | Title: Exercise 2: Unit fractions | Status: unseen

Let N and D be two strictly positive integers with $\frac{N}{D} = \frac{1}{d_1} + \frac{1}{d_2} + \dots + \frac{1}{d_k}$. There are actually infinitely many such representations. Indeed, since $1 = \frac{1}{2} + \frac{1}{3} + \frac{1}{6}$ if $\frac{N}{D} = \frac{1}{d_1} + \frac{1}{d_2} + \dots + \frac{1}{d_k}$ then also $\frac{N}{D} = \frac{1}{d_1} + \frac{1}{d_2} + \dots + \frac{1}{d_{k-1}} + \frac{1}{2d_k} + \frac{1}{3d_k} + \frac{1}{6d_k}$. One particular representation is obtained by a method proposed by Fibonacci, in the form of a greedy algorithm. Suppose that N/D cannot be simplified, that is, N and D have no other common factor but 1. If $N=1$ then we are done, so suppose otherwise. Let d_1 be the smallest integer such that $\frac{N}{D}$ can be written as $\frac{1}{d_1} + f_1$, with f_1 necessarily strictly positive by assumption. Looking for the smallest d_1 is what makes the algorithm greedy. Of course, d_1 is equal to $D \div N + 1$. By the choice of d_1 , $\frac{1}{d_1-1} > \frac{N}{D}$, hence $D > N(d_1-1)$, hence $N > Nd_1 - D$. Since f_1 is equal to $\frac{N}{D} - \frac{1}{d_1} = \frac{Nd_1 - D}{Dd_1}$, it follows that $\frac{N}{D}$ can be written as $\frac{1}{d_1} + \frac{N_1}{D_1}$ with N_1 then the same argument allows one to greedily

find $d_2 > d_1$ such that for some strictly positive integers N_2 and D_2 , $\frac{N}{D}$ can be written as $\frac{1}{d_1} + \frac{1}{d_2} + \frac{N_2}{D_2}$ with $N_2 \leq N$ then the same argument allows one to greedily find $d_3 > d_2$ such that for some strictly positive integers N_3 and D_3 , $\frac{N}{D}$ can be written as $\frac{1}{d_1} + \frac{1}{d_2} + \frac{1}{d_3} + \frac{N_3}{D_3}$ with $N_3 \leq N$. The number of summands in the sum of unit fractions given by Fibonacci's method is not always minimal: it is sometimes possible to decompose $\frac{N}{D}$ as sum of unit fractions with fewer summands. For instance, Fibonacci's method yields $\frac{4}{17} = \frac{1}{5} + \frac{1}{29} + \frac{1}{1233} + \frac{1}{3039345}$ whereas $\frac{4}{17}$ can be written as a sum of 3 unit fractions, actually in 4 possible ways: $\frac{4}{17} = \frac{1}{5} + \frac{1}{30} + \frac{1}{510}$, $\frac{4}{17} = \frac{1}{5} + \frac{1}{34} + \frac{1}{170}$, $\frac{4}{17} = \frac{1}{6} + \frac{1}{15} + \frac{1}{510}$, $\frac{4}{17} = \frac{1}{6} + \frac{1}{17} + \frac{1}{102}$. Complete the program `unit_fractions.py` so as to have the functionality of the two functions: `fibonacci_decomposition(N, D)`, that takes two strictly positive integers N and D as arguments, and writes N/D as a sum of unit fractions following Fibonacci method, plus an integer in case $N \geq D$ (in a unique way); `shortest_length_decompositions(N, D)`, that also takes two strictly positive integers N and D as arguments, and writes N/D as a sum of unit fractions with a minimal number of summands, plus an integer in case $N \geq D$ (in possibly many ways). Here are possible interactions: `>>> from unit_fractions import *`

```
>>> fibonacci_decomposition(1, 521)
1/521 = 1/521
>>> fibonacci_decomposition(521, 521)
521/521 = 1
>>> fibonacci_decomposition(521, 1050)
521/1050 = 1/3 + 1/7 + 1/50
>>> fibonacci_decomposition(1050, 521)
1050/521 = 2 + 1/66 + 1/4913 + 1/33787684 + 1/2854018941421956
>>> fibonacci_decomposition(6, 7)
6/7 = 1/2 + 1/3 + 1/42
>>> shortest_length_decompositions(6, 7)
6/7 = 1/2 + 1/3 + 1/42
>>> fibonacci_decomposition(8, 11)
8/11 = 1/2 + 1/5 + 1/37 + 1/4070
>>> shortest_length_decompositions(8, 11)
8/11 = 1/2 + 1/5 + 1/37 + 1/4070
8/11 = 1/2 + 1/5 + 1/38 + 1/1045
8/11 = 1/2 + 1/5 + 1/40 + 1/440
8/11 = 1/2 + 1/5 + 1/44 + 1/220
8/11 = 1/2 + 1/5 + 1/45 + 1/198
8/11 = 1/2 + 1/5 + 1/55 + 1/110
8/11 = 1/2 + 1/5 + 1/70 + 1/77
8/11 = 1/2 + 1/6 + 1/17 + 1/561
8/11 = 1/2 + 1/6 + 1/18 + 1/198
8/11 = 1/2 + 1/6 + 1/21 + 1/77
8/11 = 1/2 + 1/6 + 1/22 + 1/66
8/11 = 1/2 + 1/7 + 1/12 + 1/924
8/11 = 1/2 + 1/7 + 1/14 + 1/77
8/11 = 1/2 + 1/8 + 1/10 + 1/440
8/11 = 1/2 + 1/8 + 1/11 + 1/88
8/11 = 1/3 + 1/4 + 1/7 + 1/924
>>> fibonacci_decomposition(4, 17)
4/17 = 1/5 + ...
```


We consider Diophantine equations of the form $ax+by=c$ with a and b both not equal to 0. We will represent such an equation as a string of the form $ax+by=c$ or $ax-by=c$ where a and c are nonzero integer literals (not preceded by $+$ in case they are positive) and where b is a strictly positive integer literal (not preceded by $+$), possibly with spaces anywhere at the beginning, at the end, and around the $+$, $-$ and $=$ characters. The equation $ax+by=c$ has a solution iff c is a multiple of $\gcd(a,b)$. In case c is indeed a multiple of $\gcd(a,b)$, then $ax+by=c$ has infinitely many solutions, namely, all pairs (x,y) of the form $\left(x_0 + \frac{\mathrm{lcm}(a,b)}{a}n, y_0 - \frac{\mathrm{lcm}(a,b)}{b}n\right)$ for arbitrary integers n , where $\mathrm{lcm}(a,b)$ denotes the least common multiplier of a and b , and where (x_0,y_0) is a solution to the equation. That particular solution can be derived from the extended Euclidian algorithm, that yields not only $\gcd(a,b)$ but also a pair of Bézout coefficients, namely, two integers x and y with $ax+by=\gcd(a,b)$. To normalise the representation of the solutions, we rewrite the equation above as $\left(x_0 + \frac{\mathrm{lcm}(a,b)}{a}n, y_0 - \mathrm{sign}(a)\frac{\mathrm{lcm}(a,b)}{b}n\right)$ where $\mathrm{sign}(a)$ is 1 if a is positive and -1 if a is negative, and we impose that the pair (x_0,y_0) is such that x_0 is nonnegative and minimal. Write a Python program `diophantine_equation.py` that defines a function `diophantine()` that prints out whether the equation provided as argument has a solution, and in case it does, prints out the normalised representation of its solutions. The output reproduces the equation nicely formatted, that is, with a single space around the $+$, $-$ and $=$ characters. As for the representation of the solutions, it is also nicely formatted, omitting x_0 or y_0 when they are equal to 0, and omitting 1 as a factor of n .

Press the Run or Mark buttons for possible interactions: `>>> diophantine('1x + 1y = 0')`

`1x + 1y = 0` has as solutions all pairs of the form

`(n, -n)` with n an arbitrary integer.

`>>> diophantine('-1x + 1y = 0')`

`-1x + 1y = 0` has as solutions all pairs of the form

`(n, n)` with n an arbitrary integer.

`>>> diophantine('1x - 1y = 0')`

`1x - 1y = 0` has as solutions all pairs of the form

`(n, n)` with n an arbitrary integer.

`>>> diophantine('-1x - 1y = 0')`

`-1x - 1y = 0` has as solutions all pairs of the form

`(n, -n)` with n an arbitrary integer.

`>>> diophantine('1x + 1y = -1')`

`1x + 1y = -1` has as solutions all pairs of the form

`(n, -1 - n)` with n an arbitrary integer.

`>>> diophantine('-1x + 1y = 1')`

`-1x + 1y = 1` has as solutions all pairs of the form

`(n, 1 + n)` with n an arbitrary integer.

`>>> diophantine('4x + 6y = 9')`

`4x + 6y = 9` has no solution.

`>>> diophantine('4x + 6y = 10')`

`4x + 6y = 10` has as solutions all pairs of the form

`(1 + 3n, 1 - 2n)` with n an arbitrary integer.

`>>> diophantine('71x+83y=2')`

`71x + 83y = 2` has as solutions all pairs of the form

`(69 + 83n, -59 - 71n)` with n an arbitrary integer.

`>>> diophantine(' 782 x + 253 y = 92')`

`782x + 253y = 92` has as solutions all pairs of the form

`(4 + 11n, -12 - 34n)` with n an arbitrary integer.

`>>> diophantine('-123x -456y = 78')`

`-123x - 456y = 78` has as solutions all pairs of the form

`(118 + 152n, -32 - 41n)` with n an arbitrary integer.

`>>> diophantine('-321x +654y = -87')`

`-321x + 654y = -87` has as solutions all pairs of the form

$(149 + 218n, 73 + 107n)$ with n an arbitrary integer.

Slide ID: 378432 | Index: 6 | Title: Exercise 4: Fibonacci codes | Status: unseen

Recall that the Fibonacci sequence $(F_n)_{n \geq 0}$ is defined by the equations: $F_0 = 0$, $F_1 = 1$ and for all $n > 0$, $F_n = F_{n+1} + F_{n-2}$. It can be shown that every strictly positive integer N can be uniquely coded as a string σ of 0's and 1's ending with 1, so of the form $b_2 b_3 \dots b_k$ with $k \geq 2$ and $b_k = 1$, such that N is the sum of all F_i 's, $2 \leq i \leq k$, with $b_i = 1$. For instance, $11 = 3 + 8 = F_4 + F_6$, hence 11 is coded by 00101. Moreover: there are no two successive occurrences of 1 in σ ; F_k is the largest Fibonacci number that fits in N , and if j is the largest integer in $\{2, \dots, k-1\}$ such that $b_j = 1$ then F_j is the largest Fibonacci number that fits in $N - F_k$, and if i is the largest integer in $\{2, \dots, j-1\}$ such that $b_i = 1$ then F_i is the largest Fibonacci number that fits in $N - F_k - F_j$ Also, every string of 0's and 1's ending in 1 and having no two successive occurrences of 1's is a code of a strictly positive integer according to this coding scheme. For instance: There is only one string of 0's and 1's of length 1 ending in 1 and having no two successive occurrences of 1's; it is 1, and it codes 1. There is only one string of 0's and 1's of length 2 ending in 1 and having no two successive occurrences of 1's; it is 01, and it codes 2. The strings of 0's and 1's of length 3 ending in 1 and having no two successive occurrences of 1's are 001 and 101 and they code 3 and 4, respectively. The strings of 0's and 1's of length 4 ending in 1 and having no two successive occurrences of 1's are 0001, 1001 and 0101 and they code 5, 6 and 7, respectively. The strings of 0's and 1's of length 5 ending in 1 and having no two successive occurrences of 1's are 00001, 10001, 01001, 00101 and 10101 and they code 8, 9, 10, 11 and 12, respectively.... The Fibonacci code of N adds 1 at the end of σ ; the resulting string then ends in two 1's, therefore marking the end of the code, and allowing one to let one string code a finite sequence of strictly positive integers. For instance, 00101100111011 codes $(11, 3, 4)$. Implement the two functions in the stub, one that takes one argument N meant to be a strictly positive integer and returns its Fibonacci code, and one that takes one argument σ meant to be a string consisting of 0's and 1's, returns 0 if σ cannot be a Fibonacci code, and otherwise returns the integer N such that σ is the Fibonacci code of N . Here are possible interactions:

```
...
>>> from fibonacci_codes import *
>>> encode(1)
'11'
>>> encode(2)
'011'
>>> encode(3)
'0011'
>>> encode(4)
'1011'
>>> encode(8)
'000011'
>>> encode(11)
'001011'
>>> encode(12)
'101011'
>>> encode(14)
'1000011'
>>> decode('1')
0
>>> decode('01')
0
>>> decode('100011011')
0
```

```

>>> decode('11')
1
>>> decode('011')
2
>>> decode('0011')
3
>>> decode('1011')
4
>>> decode('000011')
8
>>> decode('001011')
11
>>> decode('1000011')
14

```

Slide ID: 381773 | **Index:** 7 | **Title:** Exercise 5: Change making | **Status:** unseen

Write a program `change_making.py` that prompts the user for the face values of coins and their associated quantities as well as for an amount, and if possible, outputs the minimal number of coins needed to match that amount, as well as the detail of how many coins of each type value are used. The face values and associated quantities should be input as a dictionary. You might find the `literal_eval()` function from the `ast` module to be useful. A solution is output from smallest face value to largest face value. If a solution is represented as a list of pairs of the form (coin face value, number of coins) ordered from smallest to largest face value, then the solutions themselves are output in lexicographical order (for sequences of pairs). All face values for a given solution are right aligned. Insert your code into `change_making.py`. Here are

examples of interactions: `$ python3 change_making.py`

Input a dictionary whose keys represent coin face values
with as value for a given key the number of coins
that are available for the corresponding face value:

```
{2: 100, 50: 100}
```

Input the desired amount: 99

There is no solution.

`$ python3 change_making.py`

Input a dictionary whose keys represent coin face values
with as value for a given key the number of coins
that are available for the corresponding face value:

```
{1: 30, 20: 30, 50: 30}
```

Input the desired amount: 60

There is a unique solution:

```
$20: 3
```

`$ python3 change_making.py`

Input a dictionary whose keys represent coin face values
with as value for a given key the number of coins
that are available for the corresponding face value:

```
{1: 100, 2: 5, 3: 4, 10: 5, 20: 4, 30: 1}
```

Input the desired amount: 107

There are 2 solutions:

\$1: 1
\$3: 2
\$10: 1
\$20: 3
\$30: 1

\$2: 2
\$3: 1
\$10: 1
\$20: 3
\$30: 1

\$ python3 change_making.py
Input a dictionary whose keys represent coins face values
with as value for a given key the number of coins
that are available for the corresponding face value:
{1: 7, 2: 5, 3: 4, 4: 3, 5: 2}
Input the desired amount: 29

There are 4 solutions:

\$1: 1
\$3: 2
\$4: 3
\$5: 2

\$2: 1
\$3: 3
\$4: 2
\$5: 2

\$2: 2
\$3: 1
\$4: 3
\$5: 2

\$3: 4
\$4: 3
\$5: 1

\$ python3 change_making.py
Input a dictionary whose keys represent coins face values
with as value for a given key the number of coins
that are available for the corresponding face value:
{11:34, 12:34, 13: 234, 17:44, 18:54, 19: 3}
Input the desired amount: 3422

There are 8 solutions:

\$11: 1
\$12: 4
\$13: 122
\$17: 44
\$18: 54

\$19: 3

\$11: 1

\$13: 127

\$17: 43

\$18: 54

\$19: 3

\$11: 2

\$12: 2

\$13: 123

\$17: 44

\$18: 54

\$19: 3

\$11: 3

\$13: 124

\$17: 44

\$18: 54

\$19: 3

\$12: 1

\$13: 127

\$17: 44

\$18: 53

\$19: 3

\$12: 2

\$13: 126

\$17: 43

\$18: 54

\$19: 3

\$12: 6

\$13: 121

\$17: 44

\$18: 54

\$19: 3

\$13: 128

\$17: 44

\$18: 54

\$19: 2

The natural approach makes use of the linear programming technique exemplified in the computation of the Levenshtein distance between two words discussed in Week 9 lectures.

Lesson 40: Sample Exam Questions

Lesson ID: 53351 | **Created:** 2024-05-27T01:25:53.761254+10:00 | **First Viewed:** 2025-09-18T14:16:04.500291+10:00 | **Last Viewed Slide ID:** 362160 | **State:** active | **Status:** attempted | **Slide Count:** 9

Slides

Slide ID: 362154 | **Index:** 10 | **Title:** Python 3 Cheat Sheet | **Status:** completed

Slide ID: 362155 | **Index:** 11 | **Title:** Question 1 | **Status:** unseen

Complete the code in the function that, given a list L of random non negative whole numbers, decomposes L into a list R of increasing sequences and with consecutive duplicates removed. Here are some execution examples: >>> f(0, 0, 10)

Here is L: []

The decomposition of L into increasing sequences, with consecutive duplicates removed, is:

[]

>>> f(0, 1, 10)

Here is L: [6]

The decomposition of L into increasing sequences, with consecutive duplicates removed, is:

[[6]]

>>> f(0, 2, 10)

Here is L: [6, 6]

The decomposition of L into increasing sequences, with consecutive duplicates removed, is:

[[6]]

>>> f(0, 3, 10)

Here is L: [6, 6, 0]

The decomposition of L into increasing sequences, with consecutive duplicates removed, is:

[[6], [0]]

>>> f(0, 4, 10)

Here is L: [6, 6, 0, 4]

The decomposition of L into increasing sequences, with consecutive duplicates removed, is:

[[6], [0, 4]]

>>> f(0, 5, 10)

Here is L: [6, 6, 0, 4, 8]

The decomposition of L into increasing sequences, with consecutive duplicates removed, is:

[[6], [0, 4, 8]]

>>> f(0, 6, 10)

Here is L: [6, 6, 0, 4, 8, 7]

The decomposition of L into increasing sequences, with consecutive duplicates removed, is:

```
[[6], [0, 4, 8], [7]]
```

```
>>> f(0, 7, 10)
```

Here is L: [6, 6, 0, 4, 8, 7, 6]

The decomposition of L into increasing sequences,
with consecutive duplicates removed, is:

```
[[6], [0, 4, 8], [7], [6]]
```

```
>>> f(3, 10, 6)
```

Here is L: [1, 4, 4, 1, 2, 4, 3, 5, 4, 0]

The decomposition of L into increasing sequences,
with consecutive duplicates removed, is:

```
[[1, 4], [1, 2, 4], [3, 5], [4], [0]]
```

```
>>> f(3, 15, 8)
```

Here is L: [3, 8, 2, 5, 7, 1, 0, 7, 4, 8, 3, 3, 7, 8, 8]

The decomposition of L into increasing sequences,
with consecutive duplicates removed, is:

```
[[3, 8], [2, 5, 7], [1], [0, 7], [4, 8], [3, 7, 8]]
```

Slide ID: 362156 | **Index:** 12 | **Title:** Question 2 | **Status:** unseen

Write a function that accepts a strictly positive integer and displays its binary representation as well as the number of times the value 1 appears in its binary representation. You might find the function `bin()`

useful. Here are some execution examples: `>>> f(1)`

1 in binary reads as: 1.

Only one bit is set to 1 in the binary representation of 1.

```
>>> f(2)
```

2 in binary reads as: 10.

Only one bit is set to 1 in the binary representation of 2.

```
>>> f(3)
```

3 in binary reads as: 11.

2 bits are set to 1 in the binary representation of 3.

```
>>> f(7)
```

7 in binary reads as: 111.

3 bits are set to 1 in the binary representation of 7.

```
>>> f(2314)
```

2314 in binary reads as: 100100001010.

4 bits are set to 1 in the binary representation of 2314.

```
>>> f(9871)
```

9871 in binary reads as: 10011010001111.

8 bits are set to 1 in the binary representation of 9871.

Slide ID: 362157 | **Index:** 13 | **Title:** Question 3 | **Status:** seen

Write a function that accepts a strictly positive integer greater or equal to 2 and "not too large" and displays its decomposition into prime factors. Here are some execution examples: `>>> f(2)`

The decomposition of 2 into prime factors reads:

```

2 = 2
>>> f(3)
The decomposition of 3 into prime factors reads:
3 = 3
>>> f(4)
The decomposition of 4 into prime factors reads:
4 = 2^2
>>> f(5)
The decomposition of 5 into prime factors reads:
5 = 5
>>> f(6)
The decomposition of 6 into prime factors reads:
6 = 2 x 3
>>> f(8)
The decomposition of 8 into prime factors reads:
8 = 2^3
>>> f(10)
The decomposition of 10 into prime factors reads:
10 = 2 x 5
>>> f(15)
The decomposition of 15 into prime factors reads:
15 = 3 x 5
>>> f(100)
The decomposition of 100 into prime factors reads:
100 = 2^2 x 5^2
>>> f(5432)
The decomposition of 5432 into prime factors reads:
5432 = 2^3 x 7 x 97
>>> f(45103)
The decomposition of 45103 into prime factors reads:
45103 = 23 x 37 x 53
>>> f(45100)
The decomposition of 45100 into prime factors reads:
45100 = 2^2 x 5^2 x 11 x 41

```

Slide ID: 362158 | **Index:** 14 | **Title:** Question 4 | **Status:** unseen

Will be tested with a at least equal to 2 and b at most equal to 10_000_000. Here are some execution examples: >>> f(2, 2)

There is a unique prime number between 2 and 2.

```
>>> f(2, 3)
```

There are 2 prime numbers between 2 and 3.

```
>>> f(2, 5)
```

There are 3 prime numbers between 2 and 5.

```
>>> f(4, 4)
```

There is no prime number between 4 and 4.

```
>>> f(14, 16)
```

There is no prime number between 14 and 16.


```
>>> f(3, 20)
```

There are 7 prime numbers between 3 and 20.

```
>>> f(100, 800)
```

There are 114 prime numbers between 100 and 800.

```
>>> f(123, 456789)
```

There are 38194 prime numbers between 123 and 456789.

Slide ID: 362159 | **Index:** 15 | **Title:** Question 5 | **Status:** unseen

Write a function that accepts a year between 1913 and 2013 inclusive and displays the maximum inflation during that year and the month(s) in which it was achieved. You might find the `reader()` function of the `csv` module useful, but you can also use the `split()` method of the `str` class. Make use of the attached `cpiai.csv` file. Here are some execution examples:

```
>>> f(1914)
```

In 1914, maximum inflation was: 2.0

It was achieved in the following months: Aug

```
>>> f(1922)
```

In 1922, maximum inflation was: 0.6

It was achieved in the following months: Jul, Oct, Nov, Dec

```
>>> f(1995)
```

In 1995, maximum inflation was: 0.4

It was achieved in the following months: Jan, Feb

```
>>> f(2013)
```

In 2013, maximum inflation was: 0.82

It was achieved in the following months: Feb

Slide ID: 362160 | **Index:** 16 | **Title:** Question 6 | **Status:** seen

You might find the `zip()` function useful, though you can also do without it. Here are some execution examples:

```
>>> f(0, 2, 2)
```

Here is the square:

```
1 1
```

```
0 1
```

It is not a good square because it contains duplicates, namely: 1

```
>>> f(0, 3, 5)
```

Here is the square:

```
3 3 0
```

```
2 4 3
```

```
3 2 3
```

It is not a good square because it contains duplicates, namely: 2 3

```
>>> f(0, 6, 50)
```

Here is the square:

```
24 48 26 2 16 32
```

```
31 25 19 30 22 37
```

```
13 32 8 18 8 48
```

```
6 39 16 34 45 38
```

9 19 6 46 4 43
21 30 35 6 22 27

It is not a good square because it contains duplicates, namely: 6 8 16 19 22 30 32 48

```
>>> f(0, 2, 50)
```

Here is the square:

24 48

26 2

It is a good square.

Ordering the elements from left to right column, from top to bottom, yields:

2 26

24 48

```
>>> f(0, 3, 100)
```

Here is the square:

49 97 53

5 33 65

62 51 38

It is a good square.

Ordering the elements from left to right column, from top to bottom, yields:

5 49 62

33 51 65

38 53 97

```
>>> f(0, 6, 5000)
```

Here is the square:

3155 3445 331 2121 4188 3980

3317 2484 3904 2933 4779 1789

4134 1140 2308 1144 776 2052

4362 4930 1203 2540 809 604

2704 3867 4585 824 2898 3556

2590 1675 4526 3907 3626 4270

It is a good square.

Ordering the elements from left to right column, from top to bottom, yields:

331 1144 2308 2933 3867 4270

604 1203 2484 3155 3904 4362

776 1675 2540 3317 3907 4526

809 1789 2590 3445 3980 4585

824 2052 2704 3556 4134 4779

1140 2121 2898 3626 4188 4930

Slide ID: 362161 | **Index:** 17 | **Title:** Question 7 | **Status:** seen

Write a function that accepts a strictly positive integer called height and displays a triangle shape of numbers starting from 0 and of height height. Use only digits from 0 to 9 to construct the shape as per the examples below: >>> f(1)

0

```
>>> f(2)
```

0

123

```
>>> f(3)
```

0

```

123
45678
>>> f(4)
0
123
45678
9012345
>>> f(5)
0
123
45678
9012345
678901234
>>> f(6)
0
123
45678
9012345
678901234
56789012345
>>> f(20)
0
123
45678
9012345
678901234
56789012345
6789012345678
901234567890123
45678901234567890
1234567890123456789
012345678901234567890
12345678901234567890123
4567890123456789012345678
901234567890123456789012345
67890123456789012345678901234
5678901234567890123456789012345
678901234567890123456789012345678
90123456789012345678901234567890123
4567890123456789012345678901234567890
123456789012345678901234567890123456789

```

Slide ID: 362162 | **Index:** 18 | **Title:** Question 8 | **Status:** seen

Write a function that accepts a string of DISTINCT UPPERCASE letters only called letters and displays all pairs of words using all (distinct) letters in letters. Please note that the words need to be valid. Use the provided dictionary.txt to check the validity of words. Here are some execution examples: >>> f('ABCDEFGH')

There is no solution

>>> f('GRIHWSNYP')

The pairs of words using all (distinct) letters in "GRIHWSNYP" are:

('SPRING', 'WHY')

>>> f('ONESIX')

The pairs of words using all (distinct) letters in "ONESIX" are:

('ION', 'SEX')

('ONE', 'SIX')

>>> f('UTAROFSMN')

The pairs of words using all (distinct) letters in "UTAROFSMN" are:

('AFT', 'MOURNS')

('ANT', 'FORUMS')

('ANTS', 'FORUM')

('ARM', 'FOUNTS')

('ARMS', 'FOUNT')

('AUNT', 'FORMS')

('AUNTS', 'FORM')

('AUNTS', 'FROM')

('FAN', 'TUMORS')

('FANS', 'TUMOR')

('FAR', 'MOUNTS')

('FARM', 'SNOUT')

('FARMS', 'UNTO')

('FAST', 'MOURN')

('FAT', 'MOURNS')

('FATS', 'MOURN')

('FAUN', 'STORM')

('FAUN', 'STROM')

('FAUST', 'MORN')

('FAUST', 'NORM')

('FOAM', 'TURNS')

('FOAMS', 'RUNT')

('FOAMS', 'TURN')

('FORMAT', 'SUN')

('FORUM', 'STAN')

('FORUMS', 'NAT')

('FORUMS', 'TAN')

('FOUNT', 'MARS')

('FOUNT', 'RAMS')

('FOUNTS', 'RAM')

('FUR', 'MATSON')

('MASON', 'TURF')

('MOANS', 'TURF')

Lesson 41: ASCII art

Lesson ID: 53354 | Created: 2024-05-27T01:25:55.567405+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 362173 | Index: 1 | Title: ASCII art | Status: unseen

Slide ID: 362174 | Index: 2 | Title: ASCII art | Status: unseen

Lesson 42: Elementary cellular automata

Lesson ID: 53355 | Created: 2024-05-27T01:25:55.76284+10:00 | State: active | Status: unattempted | Slide Count: 5

Slides

Slide ID: 362175 | Index: 1 | Title: Elementary cellular automata | Status: unseen

Slide ID: 362176 | Index: 2 | Title: Elementary cellular automata | Status: unseen

Slide ID: 362177 | Index: 3 | Title: Elementary cellular automata | Status: unseen

Slide ID: 362178 | Index: 4 | Title: Elementary cellular automata I | Status: unseen

Slide ID: 362179 | Index: 5 | Title: Elementary cellular automata II | Status: unseen

Lesson 43: Discrete probability distributions

Lesson ID: 53361 | Created: 2024-05-27T01:25:57.15921+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 384761 | Index: 1 | Title: Discrete probability distributions | Status: unseen

Slide ID: 362188 | Index: 2 | Title: Discrete probability distributions | Status: unseen

Lesson 44: Week 2

Lesson ID: 53334 | Created: 2024-05-27T01:25:45.710299+10:00 | State: active | Status: unattempted | Slide Count: 8

Slides

Slide ID: 362094 | Index: 1 | Title: Exercise 1: Greetings | Status: unseen

Write a program that asks the user for their name, and then prints "Hello, !". Note that represents the content of the variable name. Example Enter your name: Ed
Hello, Ed! Hint: You'll want to use the input() and print() functions for this challenge.
message = input('Message: ')
print(message)

Slide ID: 362095 | Index: 2 | Title: Exercise 2: BMI calculator | Status: unseen

Write a program that calculates the user's BMI (body mass index). The formula is: $\text{BMI} = \frac{\text{weight in kilograms}}{(\text{height in meters})^2}$ Example What is your weight in kg? 70
What is your height in m? 1.82
Your BMI is 21.1

Slide ID: 362096 | Index: 3 | Title: Exercise 3: Odd/even checker | Status: unseen

Write a program that asks the user for an integer number and then displays whether the number is odd or even. Example 1: Enter a number: 23 The number 23 is odd. Example 2: Enter a number: 10 The number 10 is even.

Slide ID: 362097 | Index: 4 | Title: Exercise 4: Temperature conversions | Status: unseen

Study the program fahrenheit_to_celsius.py and run it in the Terminal window, executing "python fahrenheit_to_celsius.py". Then complete the program celsius_to_fahrenheit.py that displays a conversion table from Celsius degrees to Fahrenheit degrees, with the former ranging from 0 to 100 in steps of 10; run it and check your solution with the Run and Mark (or Submit) buttons, respectively. See commands_and_expected_outputs.txt for expected output.

Slide ID: 367843 | Index: 5 | Title: Exercise 5: Tax calculator | Status: unseen

Write a program that asks the user for her income and then displays the estimated amount of tax on her income. Use the Australian Tax Office's 2022-23 calculation method as shown below:
0 – \$18,200: Nil

\$18,201 – \$45,000: 19c for each \$1 over \$18,200

\$45,001 – \$120,000: \$5,092 plus 32.5c for each \$1 over \$45,000

\$120,001 – \$180,000: \$29,467 plus 37c for each \$1 over \$120,000

\$180,001 and over: \$51,667 plus 45c for each \$1 over \$180,000

You may assume that the user enters a valid income, which is an integer greater than or equal to zero (that is, no decimal places), so you don't need to check for invalid input. Your program only needs to ask for one income, so you don't need it to keep asking for further incomes. You can round the tax to the nearest integer. See ATO's Simple tax calculator. Examples: What was your income in 2022-23? 76000

The estimated tax on your income is \$15167 What was your income in 2022-23? 125000

The estimated tax on your income is \$31317

To help you check your code, here are some sample incomes and the tax your program should calculate:
\$0 income -> \$0 tax

\$10,000 income -> \$0 tax
\$25,000 income -> \$1,292 tax
\$45,000 income -> \$5,092 tax
\$120,001 income -> \$29,467 tax
\$120,002 income -> \$29,468 tax
\$140,000 income -> \$36,867 tax
\$250,000 income -> \$83,167 tax

Slide ID: 367844 | **Index:** 87 | **Title:** Exercise 6: Word count | **Status:** unseen

Write a program that asks the user for some text and then says how many words the text contains. You may assume that words are separated by a space. Example: What is your text? The quick brown fox jumped over the lazy dog
Your text contains 9 words.

Slide ID: 367845 | **Index:** 91 | **Title:** Exercise 7: Acronym maker | **Status:** unseen

Write a program that asks the user for a multi-word name and then returns the corresponding acronym. Example: What is the name? World Health Organisation
Its acronym is WHO.

Slide ID: 367846 | **Index:** 95 | **Title:** Exercise 8: 24 hour time converter | **Status:** unseen

Write a program that converts 24 hr times to 12 hr times. Examples: What is the time? 0637
0637 is 6:37 am What is the time? 1423
1423 is 2:23 pm What is the time? 1200
1200 is 12:00 pm What is the time? 2400
2400 is 12:00 am

Lesson 45: Week 3

Lesson ID: 53335 | **Created:** 2024-05-27T01:25:46.425696+10:00 | **State:** active | **Status:** unattempted | **Slide Count:** 8

Slides

Slide ID: 362102 | **Index:** 87 | **Title:** Exercise 1: Word frequency | **Status:** unseen

In the workspace on the right is a file called "text.txt" which contains a piece of text (it's the opening passage of Charles Darwin's *On the Origin of Species*). Write a program that reads the text in the file and lists each unique word, along with its frequency (i.e., how many times it occurs). Example: (These numbers might not be correct - they're just a guide to the sort of output you should generate.)

we: 3

look: 1

to: 9

the: 11

... etc. Your program should work not only for this file but also for variations. You may assume the following about the text in a file: First. Words are separated by whitespace. In the sample text given all of the whitespace is just a single space. But sometimes it might be multiple spaces, or tabs, or line breaks, etc. Make sure your program can handle all of these different kinds of whitespace. You will find the string method `split` very helpful. Second. The text might contain punctuation marks (as the sample text does). Make sure that you don't include punctuation marks in words. You could remove them from the text altogether. It will be good enough if your program can handle the following punctuation marks: `. ? ! , ; : () [] { } "`

Don't worry about dashes and single quote marks:- '

These are tricky, because sometimes they are used as parts of words (e.g., hyphenated words, such as "sub-variety", or contractions, such as "aren't") and sometimes they are used not as part of words (e.g., as dashes or quote marks). If you feel like a challenge then you could get your program to deal with them correctly, but you're not expected to.

Third. Words might occur both with a capital first letter and without a capital first letter (e.g., the sample text contains both "When" and "when"). You should consider these to be the same word. You could make all words lowercase, turning "When" into "when". Or you could make them all upper case, turning both "When" and "when" into "WHEN". It's up to you.

Checking your work

Note that you can add your own files to the workspace on the right. So you could create your own text file, called, for example, "my_text.txt", put whatever text you like in that file, and then check your program by getting it to read that file instead of the sample file. You could add something like the following text, which contains things your program should be able to handle:

```
Hello, world, hello!
```

World: hello?

GOODBYE.

A nice thing about this is that you know what answers you should get: "hello" occurs three times, "world" occurs twice, and "goodbye" occurs once. So, if you're converting words to lowercase then your output should be something like this:

world: 2

goodbye: 1Optional extraIf you're feeling up to it, get the words to appear in alphabetical order. Even better, get them to appear in order of frequency, from the most frequent down to the least frequent. Again, you're not expected to.

Slide ID: 362104 | Index: 91 | Title: Exercise 2: Vowel stripper | Status: unseen

It is sometimes said that English text is still fairly easy to read even if you remove all of the vowels. To test this, write a program that asks the user for a sentence, and then prints the sentence with all of the vowels removed. Example: What is your sentence? The quick brown fox jumped over the lazy dog

Here it is without vowels: Th qck brwn fx jmpd vr th lzy dg

It might work best if you only remove vowels from inside words (i.e., not from the beginnings or ends of words). Write an improved version that deals with that.

Slide ID: 362105 | **Index:** 92 | **Title:** Exercise 3: Fibonacci lister | **Status:** unseen

The Fibonacci numbers are a famous sequence of numbers that goes as follows: 0, 1, 1, 2, 3, 5, 8, 13, ... The rule for generating the sequence is this: The sequence starts with 0, 1. The next number is the sum of the previous two numbers. Write a program that prints as many Fibonacci numbers as the user would like. Get the numbers to appear on the same line, separated by commas. Example: How many Fibonacci numbers would you like? 10
0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Slide ID: 362107 | **Index:** 95 | **Title:** Exercise 4: ISBN validator | **Status:** unseen

An ISBN (International Standard Book Number) is a 10 character string assigned to every commercial book before 2007. Each character is a digit between 0 and 9, but the last character might also be 'X'. Write a program that asks the user for an ISBN and determines whether it is valid or not. The check for validity goes as follows: Multiply each of the first 9 digits by its position. The positions go from 1 to 9. Add up the 9 resulting products. Divide this sum by 11, and get the remainder, which is a number between 0 and 10. If the remainder is 10, the last character should be the letter 'X'. Otherwise, the last character should be the remainder (a single digit). Examples: Enter ISBN: 1503290565
1503290565 is valid Enter ISBN: 938007834X
938007834X is valid
Enter ISBN: 2222222224
2222222224 is invalid

Slide ID: 367847 | **Index:** 102 | **Title:** Exercise 5: Max element and span in a list | **Status:** unseen

Study the program `max_in_list.py` and run it in the Terminal window, executing `"python max_in_list.py"`. Then complete the program `span.py` that prompts the user for a seed for the random number generator, and for a strictly positive number, `nb_of_elements`, generates a list of `nb_of_elements` random integers between 0 and 99, prints out the list, computes the difference between the largest and smallest values in the list without using the built-ins `min()` and `max()`, prints it out, and check that the result is correct using the built-ins; run it and check your solution with the Run and Mark (or Submit) buttons, respectively. See `commands_and_expected_outputs.txt` for expected outputs and sample inputs.

Slide ID: 367848 | **Index:** 103 | **Title:** Exercise 6: Classifying elements in a list | **Status:** unseen

The operators `/`, `//` and `%` are used for floating point division, integer division, and remainder, respectively. Study the program `modulo_4.py` and run it in the Terminal window, executing `"python modulo_4.py"`. Then complete program `intervals.py` that prompts the user for a strictly positive integer, `nb_of_elements`, generates a list of `nb_of_elements` random integers between 0 and 19, prints out the list, computes the number of elements strictly less than 5, 10, 15 and 20, and prints those out; run it and check your solution with the Run and Mark (or Submit) buttons, respectively. See `commands_and_expected_outputs.txt` for expected outputs and sample inputs.

Slide ID: 367849 | **Index:** 104 | **Title:** Exercise 7: Mean, median, and standard deviation | **Status:** unseen

Complete the program `mean_median_standard_deviation.py` that prompts the user for a strictly positive integer, `nb_of_elements`, generates a list of `nb_of_elements` random integers between -50 and 50, prints out the list, computes the mean, the median and the standard deviation in two ways, that is, using or not the functions from the statistics module, and prints them out. To compute the median, the easiest way is to first sort the list with the built-in `sort()` method. See `commands_and_expected_outputs.txt` for expected outputs and sample inputs.

A number is perfect if it is equal to the sum of its divisors, itself excluded. For instance, the divisors of 28 distinct from 28 are 1, 2, 4, 7 and 14, and $1+2+4+7+14=28$, hence 28 is perfect. Insert your code into `perfect.py`. The program prompts the user for an integer N. If the input is incorrect then the program outputs an error message and exits. Otherwise the program outputs all perfect numbers at most equal to N. Implement a naive solution, of quadratic complexity, so it can deal with small values of N only. Execute your program and check your outputs against the expected outputs with the Run and Mark (or Submit) buttons, respectively. See Perfect number and List of perfect numbers (from Wikipedia) for more details.

Lesson 46: Week 4

Lesson ID: 53336 | Created: 2024-05-27T01:25:47.394393+10:00 | State: active | Status: unattempted | Slide Count: 8

Slides

Slide ID: 362111 | Index: 2 | Title: Exercise 1: Outlier remover | Status: unseen

When you calculate the mean of a collection of numbers the result can be adversely affected by outliers - values that are extreme, either extremely small or extremely large. Write a program in which you define a function that takes a list of numbers as argument and returns their mean, but when it calculates the mean it ignores the smallest number and the largest number. Give your function an informative name. Illustrate the use of the function by applying it to some example lists. Some things to be careful of: Make sure your function doesn't change the list that it receives. Make sure your function can handle lists that have fewer than 3 elements. Make sure your function can handle lists that contain non-numeric values.

Slide ID: 362113 | Index: 4 | Title: Exercise 2: Memoizer | Status: unseen

Suppose you have a function that takes a value and returns a value but the calculation it uses consumes a lot of time and resources. If the function always returns the same value when given the same value, then it can be a good idea to get the function to remember the results of its calculations. Then, if the function is passed a value that it has already had, it can simply recall the result of the calculation rather than doing the calculation again. This technique is called memoization. Write a program in which you define a function that finds the cube of a number. Get it to use memoization. Memoization is not really needed in this case, because the calculation is not time or resource intensive, but it will illustrate the principles. Here is one way to proceed: Define your function as normal. Add an attribute to the function, whose value is a dictionary. When the function is called with a number it can check this dictionary to see if it has a result saved. If it does, it can return this saved result. Otherwise, it can calculate the result, add it to the dictionary, and return the result.

Slide ID: 362114 | Index: 5 | Title: Exercise 3: List mapper | Status: unseen

Write a program in which you define a function `map()`, which takes a function and a list and applies the function to each element of the list, returning the results as a list. Examples: `x = map(abs, [-2, 4, -6, -8])`
`print(x)`
`[2, 4, 6, 8]`

Slide ID: 362115 | Index: 6 | Title: Exercise 4: Roman numerals | Status: unseen

Write a program that prints out the decimal value of a Roman numeral. Your program should accept the Roman numeral from the command line arguments. Click on Terminal to activate the terminal. You may assume the Roman numeral is in the "standard" form, i.e., any digits involving 4 and 9 will always appear in the subtractive form. Sample interactions:

```
python roman_numerals.py II
2
python roman_numerals.py IV
4
python roman_numerals.py IX
9
python roman_numerals.py XIX
19
python roman_numerals.py XX
20
python roman_numerals.py MDCCLXXVI
1776
python roman_numerals.py MMXIX
```

2019 Hints: Use a loop to iterate through the Roman numeral to figure out their value. Use a list of tuples to store the string characters and their respective values. Compare the characters from the input to this list. Use a while loop so you can manually control the indices.

Slide ID: 362116 | **Index:** 7 | **Title:** Exercise 5: Finding particular sequences of prime numbers | **Status:** unseen

Insert your code into `consecutive_primes.py` to find all sequences of 6 consecutive prime 5-digit numbers, say (a, b, c, d, e, f) , with $b = a + 2$, $c = b + 4$, $d = c + 6$, $e = d + 8$, and $f = e + 10$. a , b , c , d , e , and f are therefore all 5-digit prime numbers and no number between a and b , between b and c , between c and d , between d and e , and between e and f is prime. If you are stuck, but only when you are stuck, then use `consecutive_primes_scaffold.py`.

Slide ID: 362117 | **Index:** 8 | **Title:** Exercise 6: Special products | **Status:** unseen

Insert your code into `special_products.py` to find all triples of positive integers (i, j, k) such that i , j , and k are two digit numbers, no digit occurs more than once in i , j , and k , and the set of digits that occur in i , j , or k is equal to the set of digits that occur in the product of i , j , and k . If you are stuck, but only when you are stuck, then use `special_products_scaffold_1.py`. If you are still stuck, but only when you are still stuck, then use `special_products_scaffold_2.py`.

Slide ID: 362118 | **Index:** 9 | **Title:** Exercise 7: Finding particular sequences of triples of the form $(n, n+1, n+2)$ | **Status:** unseen

Write a program called `special_triples.py` that finds all triples of consecutive positive three-digit integers each of which is the sum of two squares, that is, all triples of the form $(n, n+1, n+2)$ such that: n , $n+1$ and $n+2$ are integers at least equal to 100 and at most equal to 999; each of n , $n+1$ and $n+2$ is of the form $a^2 + b^2$. Hint: As we are not constrained by memory space for this problem, we might use a list that stores an integer for all indexes n in $[100, 999]$, equal to 1 in case n is the sum of two squares, and to 0 otherwise. Then it is just a matter of finding three consecutive 1's in the list. This idea can be refined (by not storing 1s, but suitable nonzero values) to not only know that some number is of the form $a^2 + b^2$, but also know such a pair (a, b) . If an integer n is of the form $a^2 + b^2$, then the decomposition is not necessarily unique. We want each decomposition that is output to be the minimal one w.r.t. the natural ordering of pairs of integers (that is, (a, b)). If you are stuck, but only when you are stuck, then use `special_triples_scaffold.py`.

Slide ID: 367850 | **Index:** 99 | **Title:** Exercise 8: Number of trailing 0s in a factorial | **Status:** unseen

To illustrate, $15!$, the factorial of 15, is equal to 1307674368000, hence has 3 trailing 0s. There are at least three methods to compute the number of trailing 0s in the factorial of a number N at least equal to 5: Divide $N!$ by 10 for as long as it yields no remainder. Note that for a positive integer x , $x // 10$ "removes" the rightmost digit from x , that digit being equal to $x \% 10$. Convert $N!$ into a string and find the rightmost occurrence of a character different to 0. A Google search, or executing `dir(str)` at the python prompt, suggests which string method to use. Note that negative indexes (-1 being the index of the last character in a string, -2 the index of the penultimate character in a string, etc.) is particularly convenient here. Python computes such huge numbers as $1000!$, either iteratively multiplying all numbers from 1 up to 1000 or using `factorial()` from the `math` module (executing `import math` and then `dir(math)` at the python prompt confirms that this function is available), and the first two methods work for such numbers, but there is a much better method that operates on N rather than $N!$, hence that does not suffer the limitations of the first two, and is very efficient. The number of trailing 0s in $N!$ is equal to the number of times $N!$ is a multiple of 10, so to the number of times $N!$ is a multiple of 2×5 . It is easy to verify that $N!$ has at least as many multiples of 2 as multiples of 5. Hence the number of trailing 0s in $N!$ is equal to the number of times $N!$ is a multiple of 5 which is equal to the number of times 5 occurs in the prime decompositions of 1, 2, ..., $N-1$ and N which is equal to the number of times 5 occurs at least once in the prime decompositions of 1, 2, ..., $N-1$ and N , plus the number of times 5 occurs at least twice in the prime decompositions of 1, 2, ..., $N-1$ and N , plus the number of times 5 occurs at least thrice in the prime decompositions of 1, 2, ..., $N-1$ and N ... which is equal to the number of multiples of 5 at most equal to N , plus the number of multiples of 5^2 at most equal to N , plus the number of multiples of 5^3 at most equal to N ... Insert your code into `trailing_0s_in_factorials.py` so that the program prompts the user for a non-negative integer N . If the input

is incorrect then the program outputs an error message and exits. Otherwise the program computes $5!$ three times, using the three methods just described. See sample outputs for details on input and output. If you are stuck, but only when you are stuck, then use `trailing_0s_in_factorial_scaffold_1.py`. If you are still stuck, but only when you are still stuck, then use `trailing_0s_in_factorial_scaffold_2.py`.

Lesson 47: Quiz 3

Lesson ID: 53345 | Created: 2024-05-27T01:25:52.680081+10:00 | State: scheduled | Status: unattempted
| Slide Count: 1

Slides

Slide ID: 362147 | Index: 6 | Title: Quiz 3 | Status: unseen

See PDF file below and stub. This quiz is worth 4 marks. Marking Representation of the integer in base 3 1.25 marks Corresponding sequence of arrows 1.25 marks Sequence of arrows nicely displayed 1.50 marks ----- Total 4.00 marks Quiz 3 is due Week 5 Thursday 27 June 2024 @ 9.00pm (Sydney time). Please note that late submission with 5% penalty per day is allowed up to 3 days from the due date, that is, any late submission after Week 5 Sunday 30 June 2024 @ 9.00pm will be discarded. Please make sure not to change the filename quiz_3.py while submitting by clicking on [Mark] button in Ed. It is your responsibility to check that your submission did go through properly using Submissions link in Ed otherwise your mark will be zero for Quiz 3.

Lesson 48: Week 5

Lesson ID: 53337 | Created: 2024-05-27T01:25:48.176957+10:00 | State: active | Status: unattempted | Slide Count: 8

Slides

Slide ID: 372911 | Index: 2 | Title: Exercise 1: Fibonacci maker | Status: unseen

Recall the Fibonacci numbers from practice exercise Week 3 Exercise 3: Fibonacci lister: 0, 1, 1, 2, 3, 5, 8, 13, ... Write a program that returns a requested Fibonacci number, this time using a recursive function. Examples: Which Fibonacci number would you like? 1
It is 0. Which Fibonacci number would you like? 8
It is 13. Which Fibonacci number would you like? 12
It is 89.

Slide ID: 362119 | Index: 3 | Title: Exercise 2: Prime factoriser | Status: unseen

Write a program that asks the user for a number and then factorises the number into primes. Examples: Enter a number: 345
 $345 = 3 \times 5 \times 23$
Enter a number: 612
 $612 = 2 \times 2 \times 3 \times 3 \times 17$
Enter a number: 127
 $127 = 127$

Slide ID: 373641 | Index: 4 | Title: Exercise 3: Longest sequence of consecutive letters | Status: unseen

Write a program `longest_sequence.py` that prompts the user for a string `w` of lowercase letters and outputs the longest sequence of consecutive letters that occur in `w`, but with possibly other letters in between, starting as close as possible to the beginning of `w`. Insert your code into `longest_sequence.py`. If you are stuck, but only when you are stuck, then use `longest_sequence_scaffold.py`. Examples: Please input a string of lowercase letters: a
The solution is: a
Please input a string of lowercase letters: abceefgh
The solution is: efgh
Please input a string of lowercase letters: abcefg
The solution is: abc
Please input a string of lowercase letters: ablccecmdnneoffpg
The solution is: abcdefg
Please input a string of lowercase letters: abcdiivjwkaalbmmbz
The solution is: ijklm
Please input a string of lowercase letters: abcpqrstuvwxbcbcddeffghijklrst
The solution is: abcdefghijkl

Slide ID: 362120 | Index: 5 | Title: Exercise 4: A triangle of characters | Status: unseen

Write a program `characters_triangle.py` that gets a strictly positive integer `N` as input and outputs a triangle of height `N`. For instance, when `N = 5`, the triangle looks like this: Two built-in functions are useful for this exercise: `ord()` returns the integer that encodes the character provided as argument; `chr()` returns the character encoded by the integer provided as argument. For instance: `>>> ord('A')`
65

```
>>> chr(65)
'A'
```

Consecutive uppercase letters are encoded by consecutive integers. For instance: `>>> ord('A'), ord('B'),`

ord('C')

(65, 66, 67)

Insert your code into `characters_triangle.py`. If you are stuck, but only when you are stuck, then use `characters_triangle_scaffold_1.py`.

Slide ID: 362121 | **Index:** 6 | **Title:** Exercise 5: Pascal triangle | **Status:** unseen

Write a program `pascal_triangle.py` that prompts the user for a number N and prints out the first $N + 1$ lines of Pascal triangle, making sure the numbers are nicely aligned, as illustrated below for $N = 3, 7$ and 11 respectively: Insert your code into `pascal_triangle.py` If you are stuck, but only when you are stuck, then use `pascal_triangle_scaffold_1.py`.

Slide ID: 362122 | **Index:** 7 | **Title:** Exercise 6: Hasse diagrams | **Status:** unseen

Let a strictly positive integer n be given. Let D be the set of divisors of n . Let k be the number of prime divisors of n (that is, the number of prime numbers in D). The members of D can be arranged as the vertices of a solid in a k -dimensional space as illustrated below for $n = 12$ (in which case $D = \{1, 2, 3, 4, 6, 12\}$ and $k = 2$) and for $n = 30$ (in which case $D = \{1, 2, 3, 5, 6, 10, 15, 30\}$ and $k = 3$). Each of the solids' vertices is associated with two collections of nodes: those "directly below" it, and those "directly above" it. In particular, the prime divisors of n are "directly above" 1, and no vertex is below 1; n has exactly k vertices "directly below" it, and no vertex is above n . This suggests considering a dictionary whose keys are the members of D (inserted from smallest to largest), and as value for a given key d , the pair of ordered lists of members of D "directly below" d and "directly above" d , respectively. The solids exhibit k distinct "edge directions", one for each prime divisor of n , defining a partition of the solids' edges. One can represent this partition as a dictionary whose keys are the prime divisors of n (inserted from smallest to largest), and as value for a given key p , the ordered list of ordered pairs of members of D that make up the endpoints of the edges whose "direction" is associated with p . The program `hasse_diagram.py` defines a function `make_hasse_diagram()` that returns a named tuple `HasseDiagram` with three attributes: `factors`, for a dictionary whose keys are the members of D , and as value for a given key d (1 excepted), a string that represents the prime decomposition of d , using x for multiplication and $^$ for exponentiation, displaying only exponents greater than 1; `vertices`, for the first dictionary previously defined; `edges`, for the second dictionary previously defined. Replace `pass` in `hasse_diagram.py` with your code. Except for `namedtuple`, `hasse_diagram.py` imports a number of classes and functions from various modules that are used in the solution, but that other good solutions will make no use of.

Slide ID: 362123 | **Index:** 8 | **Title:** Exercise 7: Encoding pairs of integers as natural numbers | **Status:** unseen

Complete the program `plane_encoding.py` that implements a function `encode(a, b)` and a function `decode(n)` for the one-to-one mapping from the set of pairs of integers onto the set of natural numbers, that can be graphically described as follows: That is, starting from the point $(0, 0)$ of the plane, we move to $(1, 0)$ and then spiral counterclockwise: `encode(0,0)` returns 0 and `decode(0)` returns $(0,0)$ `encode(1,0)` returns 1 and `decode(1)` returns $(1,0)$ `encode(1,1)` returns 2 and `decode(2)` returns $(1,1)$ `encode(0,1)` returns 3 and `decode(3)` returns $(0,1)$ `encode(-1,1)` returns 4 and `decode(4)` returns $(-1,1)$ `encode(-1,0)` returns 5 and `decode(5)` returns $(-1,0)$ `encode(-1,-1)` returns 6 and `decode(6)` returns $(-1,-1)$ `encode(0,-1)` returns 7 and `decode(7)` returns $(0,-1)$ `encode(1,-1)` returns 8 and `decode(8)` returns $(1,-1)$ `encode(2,-1)` returns 9 and `decode(9)` returns $(2,-1)$. . .

Slide ID: 370980 | **Index:** 103 | **Title:** Exercise 8: Decoding a multiplication | **Status:** unseen

We want to decode all multiplications of the form such that the sum of all digits in all 4 columns is constant. Insert your code into `decoded_multiplication.py`. There are actually two solutions, see expected output for details on what it should be. If you are stuck, but only when you are stuck, then use `decoded_multiplication_scaffold.py`.

Lesson 49: Week 8

Lesson ID: 53339 | Created: 2024-05-27T01:25:49.180409+10:00 | State: active | Status: unattempted | Slide Count: 5

Slides

Slide ID: 362128 | Index: 2 | Title: Exercise 1: Obtaining a sum from a subsequence of digits | Status: unseen

Write a program `sum_of_digits.py` that prompts the user for two natural numbers, say `available_digits` and `desired_sum`, and outputs the number of ways of selecting digits from `available_digits` that sum up to `desired_sum`. For instance, if `available_digits` is 12234 and sum is 5 then there are four (4) solutions: one solution is obtained by selecting 1 and both occurrences of 2 ($1+2+2=5$); one solution is obtained by selecting 1 and 4 ($1+4=5$); one solution is obtained by selecting the first occurrence of 2 and 3 ($2+3=5$); one solution is obtained by selecting the second occurrence of 2 and 3 ($2+3=5$). Here are possible interactions: Input a number that we will use as available digits: 12234

Input a number that represents the desired sum: 5

There are 4 solutions.

Input a number that we will use as available digits: 11111

Input a number that represents the desired sum: 5

There is a unique solution.

Input a number that we will use as available digits: 11111

Input a number that represents the desired sum: 6

There is no solution.

Input a number that we will use as available digits: 1234321

Input a number that represents the desired sum: 5

There are 10 solutions.

Insert your code into `sum_of_digits.py` if you are stuck, but only when you are stuck, then use `sum_of_digits_scaffold.py`.

Slide ID: 362129 | Index: 3 | Title: Exercise 2: Merging two strings into a third one | Status: unseen

Say that two strings `s1` and `s2` can be merged into a third string `s3` if `s3` is obtained from `s1` by inserting arbitrarily in `s1` the characters in `s2`, respecting their order. For instance, the two strings `ab` and `cd` can be merged into `abcd`, or `cabd`, or `cdab`, or `acbd`, or `acdb`, ..., but not into `adbc` nor into `cbda`. Write a program `merging_strings.py` that prompts the user for 3 strings and displays the output as follows: If no string can be obtained from the other two by merging, then the program outputs that there is no solution. Otherwise, the program outputs which of the strings can be obtained from the other two by merging. Here are possible interactions: Please input the first string: `ab`

Please input the second string: `cd`

Please input the third string: `abcd`

The third string can be obtained by merging the other two.

Please input the first string: `ab`

Please input the second string: `cdab`

Please input the third string: `cd`

The second string can be obtained by merging the other two.

Please input the first string: `abcd`

Please input the second string: `cd`

Please input the third string: `ab`

The first string can be obtained by merging the other two.

Please input the first string: `ab`

Please input the second string: `cd`

Please input the third string: `adcb`

No string can be merged from the other two.

Please input the first string: aaaaaa
Please input the second string: a
Please input the third string: aaaa
The first string can be obtained by merging the other two.
Please input the first string: aaab
Please input the second string: abcab
Please input the third string: aaabcaabb
The third string can be obtained by merging the other two.
Please input the first string: ??got
Please input the second string: ?it?go#t##
Please input the third string: it###
The second string can be obtained by merging the other two.
Insert your code into merging_strings.py.If you are stuck, but only when you are stuck, then use
merging_strings_scaffold.py.

Slide ID: 362130 | **Index:** 4 | **Title:** Exercise 3: Eight puzzle | **Status:** unseen

Dispatch the integers from 0 to 8, with 0 possibly changed to None, as a list of 3 lists of size 3, to represent a 9 puzzle. For instance, let `[[4, 0, 8], [1, 3, 7], [5, 2, 6]]` or `[[4, None, 8], [1, 3, 7], [5, 2, 6]]` represent the 9 puzzle with the 8 integers being printed on 8 tiles that are placed in a frame with one location being tile free. The aim is to slide tiles horizontally or vertically so as to eventually reach the configuration. It can be shown that the puzzle is solvable iff the permutation of the integers 1, ..., 8, determined by reading those integers off the puzzle from top to bottom and from left to right, is even. This is clearly a necessary condition since: sliding a tile horizontally does not change the number of inversions; sliding a tile vertically changes the number of inversions by -2, 0 or 2; the parity of the identity is even. Complete the program `eight_puzzle.py` so as to have the functionality of the two functions: `validate_8_puzzle(grid)` that prints out whether or not `grid` is a valid representation of a solvable 8 puzzle; `solve_8_puzzle(grid)` that, assuming that `grid` is a valid representation of a solvable 8 puzzle, outputs a solution to the puzzle characterised as follows: the number of moves is minimal; at every stage, the preferences of the tile to slide are, from most preferred to least preferred: the tile above the empty cell (provided the latter is not in the top row), then the tile to the left of the empty cell (provided the latter is not in the left column), then the tile to the right of the empty cell (provided the latter is not in the right column), then the tile below the empty cell (provided the latter is not in the bottom row).

Slide ID: 362132 | **Index:** 6 | **Title:** Exercise 4: Magic squares | **Status:** unseen

Given a positive integer n , a magic square of order n is a matrix of size $n \times n$ that stores all numbers from 1 up to n^2 and such that the sum of the n rows, the sum of the n columns, and the sum of the two diagonals is constant, hence equal to $n(n^2+1)/2$. Implement in the file `magic_squares.py` the function `print_square(square)`, that prints a list of lists that represents a square, and the function `is_magic_square(square)`, that checks whether a list of lists is a magic square. Examples of execution:

Examples of execution

```
print(is_magic_square([[2,7,6], [1,5,9], [4,3,8]])) # False
print(is_magic_square([[2,7,6], [9,5,1], [4,3,8]])) # True
print(is_magic_square([[8,1,6],[3,5,7],[4,9,2]])) # True
print_square([[8,1,6],[3,5,7],[4,9,2]])
```

False

True

True

8 1 6

3 5 7

4 9 2

On the quiz show "Letters and Numbers" there is a round in which contestants are given six ingredient numbers and one target number, and their challenge is to apply arithmetic operations (addition, subtraction, multiplication, and division) to one or more of the ingredient numbers to get the target number.

For example: Ingredient numbers: 1, 2, 6, 10, 75, 100

Target number: 582

Possible answer: $(100 - (2 + 1)) \times 6 = 582$

Possible answer: $((100 - 2) - 1) \times 6 = 582$

Possible answer: $(6 \times 100) - (2 \times (10 - 1)) = 582$

The ingredient numbers are chosen randomly as follows: 0-4 large numbers, chosen from {25, 50, 75, 100} with no repeats. The remaining are all small numbers, chosen from {1, 2, 3, ..., 10} with no repeats. Each ingredient number can only be used once, but they need not all be used. No fractions are allowed at any stage of the calculation. The target number is also chosen randomly from 100 to 1000. Your challenge is to write a program that is given the ingredient numbers and the target number and calculates all possible answers (if any).

Lesson 50: Sample Exam Questions 2

Lesson ID: 53352 | Created: 2024-05-27T01:25:54.631875+10:00 | State: active | Status: unattempted | Slide Count: 7

Slides

- Slide ID: 362163 | Index: 1 | Title: Sample 2 Question 1 | Status: unseen
- Slide ID: 362164 | Index: 2 | Title: Sample 2 Question 2 | Status: unseen
- Slide ID: 362165 | Index: 3 | Title: Sample 2 Question 3 | Status: unseen
- Slide ID: 362166 | Index: 4 | Title: Sample 2 Question 4 | Status: unseen
- Slide ID: 362167 | Index: 5 | Title: Sample 2 Question 5 | Status: unseen
- Slide ID: 362168 | Index: 6 | Title: Sample 2 Question 6 | Status: unseen
- Slide ID: 362169 | Index: 7 | Title: Sample 2 Question 7 | Status: unseen

Lesson 51: Loans and savings

Lesson ID: 53353 | Created: 2024-05-27T01:25:55.331163+10:00 | State: active | Status: unattempted | Slide Count: 3

Slides

Slide ID: 362170 | Index: 1 | Title: Loans and savings | Status: unseen

Slide ID: 362171 | Index: 2 | Title: Loans and savings | Status: unseen

Slide ID: 362172 | Index: 3 | Title: loans_and_savings.py | Status: unseen

Lesson 52: Pure Prolog

Lesson ID: 53364 | Created: 2024-05-27T01:25:57.719702+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 384758 | Index: 1 | Title: Pure Prolog | Status: unseen

Slide ID: 362191 | Index: 2 | Title: Pure Prolog | Status: unseen

Lesson 53: Week 10

Lesson ID: 53341 | Created: 2024-05-27T01:25:50.518991+10:00 | State: active | Status: unattempted | Slide Count: 5

Slides

Slide ID: 378433 | Index: 6 | Title: Exercise 1: Word ladders | Status: unseen

Write a program `word_ladder.py` that computes all transformations of a word `word_1` into a word `word_2`, consisting of sequences of words of minimal length, starting with `word_1`, ending in `word_2`, and such that two consecutive words in the sequence differ by at most one letter. All words have to occur in a dictionary with name `dictionary.txt`, stored in the working directory. It is convenient and effective to first create a dictionary whose keys are all words in the dictionary `dictionary.txt` with one letter replaced by a “slot”, the value for a given key being the list of words that match the key with the “slot” being replaced by an appropriate letter. From this dictionary, one can then build a dictionary with words as keys, and as value for a given key the list of words that differ in only one letter from the key. The program implements a function `word_ladder(word_1, word_2)` that returns the list of all solutions, a solution being as previously described. Below is a possible interaction: `$ python3`

```
...
>>> from word_ladder import *
>>> for ladder in word_ladder('cold', 'warm'): print(ladder)
...
['COLD', 'CORD', 'CARD', 'WARD', 'WARM']
['COLD', 'CORD', 'WORD', 'WORM', 'WARM']
['COLD', 'CORD', 'WORD', 'WARD', 'WARM']

>>> for ladder in word_ladder('three', 'seven'): print(ladder)
...
['THREE', 'THREW', 'SHREW', 'SHRED', 'SIRED', 'SITED', 'SATED', 'SAVED', 'SAVER', 'SEVER',
'SEVEN']

>>> for ladder in word_ladder('train', 'bikes'): print(ladder)
...
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'BORES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'CARES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'CARES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'PORES', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'COKE', 'POKES', 'PIKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'COKE', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'BORES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'CARES', 'BARES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'CARES', 'CAKES', 'BAKES', 'BIKES']
['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES',
'PORES', 'POKES', 'PIKES', 'BIKES']
```

['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'POKES', 'PIKES', 'BIKES']
 ['TRAIN', 'GRAIN', 'GROIN', 'GROWN', 'GROWS', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'CAKES', 'BAKES', 'BIKES']
 ['TRAIN', 'DRAIN', 'DRAWN', 'DRAWS', 'DRAGS', 'BRAGS', 'BRATS', 'BEATS', 'BELTS', 'BELLS', 'BALLS', 'BALES', 'BAKES', 'BIKES']
 ['TRAIN', 'DRAIN', 'DRAWN', 'DRAWS', 'DRAGS', 'BRAGS', 'BRATS', 'BEATS', 'BESTS', 'BUSTS', 'BUSES', 'BASES', 'BAKES', 'BIKES']
 ['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'BORES', 'BARES', 'BAKES', 'BIKES']
 ['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'BARES', 'BAKES', 'BIKES']
 ['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'CARES', 'CAKES', 'BAKES', 'BIKES']
 ['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'PORES', 'POKES', 'PIKES', 'BIKES']
 ['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'POKES', 'PIKES', 'BIKES']
 ['TRAIN', 'DRAIN', 'DRAWN', 'DROWN', 'CROWN', 'CROWS', 'CROPS', 'COOPS', 'CORPS', 'CORES', 'COKES', 'CAKE...']

Slide ID: 378434 | **Index:** 7 | **Title:** Exercise 2: Word search puzzle | **Status:** unseen

Word search puzzle consists of a grid of letters and a number of words, that have to be read horizontally, vertically or diagonally, in either direction. Write a program `word_search.py` that defines a class `WordSearch` with the following properties: To create a `WordSearch` object, the name of a file has to be provided. This file is meant to store a number of lines all with the same number of uppercase letters, those lines possibly containing spaces anywhere, and the file possibly containing extra blank lines. `__str__()` is implemented. It has a method `number_of_solutions()` to display the number of solutions for each word length for which a solution exists. It has a method `locate_word_in_grid()` that takes a word as argument; it returns `None` if the word cannot be read in the grid, and otherwise returns the x and y coordinates of an occurrence of the first letter of the word in the grid and the direction to follow (N, NE, E, SE, S, SW, W, or NW) to read the whole word from that point onwards. Coordinates start from 0, with the x-axis pointing East, and the y-axis pointing South. It has a method `locate_words_in_grid()` that takes any number of words as arguments, and returns a dictionary whose keys are those words and whose values are `None` or the triple returned by `locate_word_in_grid()` when called with that word as argument. It has a method `display_word_in_grid()` that takes a word as argument and in case the word can be read from the grid, prints out the grid with all characters being displayed in lowercase, except for those that make up word, displayed in uppercase. Here is a possible interaction: \$ python3

```
...
>>> from word_search import *
>>> import pprint
>>> ws = WordSearch('word_search_1.txt')
>>> print(ws)
N D A O E L D L O G B M N E
I T D C M E A I N R U T S L
C L U U E I C G G G O L I I
K M U I M U I D I R I A L T
E U R T U N G S T E N B V H
L I L S L T T U L R U O E I
C M A T E T I U R D R C R U
I D S C A M A G N E S I U M
```

```

MAMPDMUINATITI
PCNPLATINUMDLL
HZEMANGANESEIG
MGITINRUNORITC
RIANNAMERCURYN
UOTCCREPPCEEER
>>> metal = 'PLATINUM'
>>> print(f'{metal}: {ws.locate_word_in_grid(metal)}')
PLATINUM: (3, 9, 'E')
>>> metal = 'SODIUM'
>>> print(f'{metal}: {ws.locate_word_in_grid(metal)}')
SODIUM: None
>>> metals = ('PLATINUM', 'COPPER', 'MERCURY', 'TUNGSTEN', 'MAGNESIUM', 'ZINC',
'MANGANESE',
... 'TITANIUM', 'TIN', 'IRON', 'LITHIUM', 'CADMIUM', 'GOLD', 'COBALT', 'SILVER',
... 'NICKEL', 'LEAD', 'IRIDIUM', 'URANIUM', 'SODIUM')
>>> located_metals = ws.locate_words_in_grid(*metals)
>>> pprint.pprint(located_metals)
{'CADMIUM': (1, 9, 'N'),
'COBALT': (11, 6, 'N'),
'COPPER': (10, 13, 'W'),
'GOLD': (9, 0, 'W'),
'IRIDIUM': (10, 3, 'W'),
'IRON': (11, 11, 'W'),
'LEAD': (4, 5, 'S'),
'LITHIUM': (13, 1, 'S'),
'MAGNESIUM': (5, 7, 'E'),
'MANGANESE': (3, 10, 'E'),
'MERCURY': (6, 12, 'E'),
'NICKEL': (0, 0, 'S'),
'PLATINUM': (3, 9, 'E'),
'SILVER': (12, 1, 'S'),
'SODIUM': None,
'TIN': (6, 9, 'NE'),
'TITANIUM': (12, 8, 'W'),
'TUNGSTEN': (3, 4, 'E'),
'URANIUM': None,
'ZINC': (1, 10, 'SE')}
>>> for metal in metals:
... print(metal, end = '\n')
... ws.display_word_in_grid(metal)
... print()
...
PLATINUM:
ndaoeldlogbmne
itdcmeainrutsI
cluueicgggolii
kmuimuidirialt
eurtungstenbvh
lilslttulruoei
cmatetiurdreru
idscamagnesium

```

mampdmuinatiti
pcnPLATINUMdll
hzemanganeseig
mgitinrunoritc
riannamercuryn
uotccreppoceer

COPPER:

ndaoeldlogbmne
itdcmeainrutsI
cluueicgggolii
kmuimuidirialt
eurtungstenbvh
lilslttulruoei
cmatetiurdreru
idscamagnesium
mampdmuinatiti
pcnplatinumdll
hzemanganeseig
mgitinrunoritc
riannamercuryn
uotc...

Slide ID: 378435 | **Index:** 8 | **Title:** Exercise 3: Possible subtractions yielding a given sum | **Status:** unseen

Write a program `subtractions.py` that takes as input an iterable `L` of nonnegative integers and an integer `N`, and displays all ways of inserting minus signs and parentheses in `L`, resulting in an expression that evaluates to `N`. You will make use of `eval()` in this exercise. Below is a possible interaction:

```
$ python3
...
>>> from subtractions import *
>>> subtractions((1, 2, 3, 4, 5), 1)
1 - ((2 - 3) - (4 - 5))
(1 - ((2 - 3) - 4)) - 5
>>> subtractions((1, 2, 3, 4, 5), 2)
>>> subtractions((1, 2, 3, 4, 5), 3)
1 - (2 - (3 - (4 - 5)))
1 - ((2 - (3 - 4)) - 5)
(1 - (2 - 3)) - (4 - 5)
>>> subtractions((1, 2, 3, 4, 5), 4)
>>> subtractions((1, 2, 3, 4, 5), 5)
(1 - 2) - ((3 - 4) - 5)
>>> subtractions((1, 3, 2, 5, 11, 9, 10, 8, 4, 7, 6), 40)
1 - (((((3 - 2) - 5) - 11) - 9) - (((10 - 8) - 4) - 7) - 6)))
1 - ((((((3 - 2) - 5) - 11) - 9) - 10) - (8 - (4 - (7 - 6))))))
1 - (((((((3 - 2) - 5) - 11) - 9) - 10) - ((8 - (4 - 7)) - 6)))
1 - (((((((((3 - 2) - 5) - 11) - 9) - 10) - (8 - 4)) - (7 - 6)))
1 - ((((((3 - 2) - 5) - 11) - (9 - (((10 - 8) - 4) - 7))) - 6)
1 - (((((((3 - 2) - 5) - 11) - (9 - ((10 - 8) - 4))) - 7) - 6)
1 - (((((((((3 - 2) - 5) - 11) - (9 - (10 - 8))) - 4) - 7) - 6)
1 - ((((((((((3 - 2) - 5) - 11) - (9 - 10)) - 8) - 4) - 7) - 6)
(1 - 3) - (((((2 - 5) - 11) - 9) - (10 - (((8 - 4) - 7) - 6))))
(1 - 3) - ((((((2 - 5) - 11) - 9) - (10 - ((8 - 4) - 7))) - 6)
```

$(1 - 3) - ((((((2 - 5) - 11) - 9) - (10 - (8 - 4))) - 7) - 6)$
 $(1 - 3) - ((((((2 - 5) - 11) - 9) - (10 - 8)) - 4) - 7) - 6)$
 $(1 - (((3 - 2) - 5) - 11) - 9)) - (((10 - 8) - 4) - 7) - 6)$
 $((1 - 3) - (((2 - 5) - 11) - 9) - 10)) - (((8 - 4) - 7) - 6)$
 $(1 - ((((((3 - 2) - 5) - 11) - 9) - 10) - 8)) - (4 - (7 - 6)))$
 $(1 - ((((((3 - 2) - 5) - 11) - 9) - 10) - (8 - (4 - 7)))) - 6$
 $(1 - (((((((3 - 2) - 5) - 11) - 9) - 10) - (8 - 4)) - 7)) - 6$
 $((1 - (((((((3 - 2) - 5) - 11) - 9) - 10) - 8)) - (4 - 7)) - 6$

Slide ID: 378436 | **Index:** 9 | **Title:** Exercise 4: Voting systems | **Status:** unseen

Find out (e.g., in Wikipedia) about these voting systems: (a) one round method, (b) two round method, (c) elimination method, (d) De Borda count, and (e) De Condorcet count. The elimination method works as follows. One adds up the tallies of all candidates who rank 1st and eliminate the candidate(s) who get the minimal number of votes (as ranked 1st candidates). For a given ordering, the candidates who remain and were ranked after the eliminated candidate(s) see their ranking go up so that the ordering is preserved, and rankings range from 1 up to the number of candidates that remain. For instance, if to start with, there are 5 candidates, A, B, C, D, and E who are ranked 1, 2, 3, 4, and 5, respectively, and if B and D are eliminated because they get the least number of votes as 1st candidates across all rankings, then for that particular ranking, A remains ranked 1st, C becomes ranked 2nd, and E becomes ranked third. The process is repeated until there is only one candidate left, or all candidates that remain get exactly the same number of votes as preferred candidates. Then design a program `election.py` that defines a class `Election`, with objects of this class created from Excel files of the kind provided as examples, to which the methods: `one_round_winners()`, `two_round_winners()`, `elimination_winner()`, `de_borda_winners()`, and `de_condorcet_winners()` can be applied. Also, the `__str__()` method is implemented so as to display in textual form the election results recorded in the Excel file. Below is a possible interaction: \$ python3

```

...
>>> from election import *
>>> election = Election('election_1.xlsx')
>>> print(election)
Number of votes Albert Emily Oscar Maria Max
3273 1 5 4 2 3
2182 5 1 4 3 2
1818 5 2 1 4 3
1636 5 4 2 1 3
727 5 2 4 3 1
364 5 4 2 3 1
>>> election.one_round_winners()
The winner is Albert.
>>> election.two_round_winners()
The winner is Emily.
>>> election.elimination_winners()
The winner is Oscar.
>>> election.de_borda_winners()
The winner is Maria.
>>> election.de_condorcet_winners()
The winner is Max.
>>> election = Election('election_2.xlsx')
>>> print(election)
Number of votes Albert Emily Oscar Maria Max
1000 1 2 3 4 5
>>> election.one_round_winners()

```

```

The winner is Albert.
>>> election.two_round_winners()
The winner is Albert.
>>> election.elimination_winners()
The winner is Max.
>>> election.de_borda_winners()
The winner is Albert.
>>> election.de_condorcet_winners()
The winner is Albert.
>>> election = Election('election_3.xlsx')
>>> print(election)
Number of votes Albert
1000 1
1000 1
1000 1
1000 1
1000 1
1000 1
1000 1
>>> election.one_round_winners()
All candidates are winners.
>>> election.two_round_winners()
All candidates are winners.
>>> election.elimination_winners()
All candidates are winners.
>>> election.de_borda_winners()
All candidates are winners.
>>> election.de_condorcet_winners()
All candidates are winners.
>>> election = Election('election_4.xlsx')
>>> print(election)
Number of votes Albert Emily Oscar
1000 1 2 3
1000 2 1 3
>>> election.one_round_winners()
The winners is Albert and Emily.
>>> election.two_round_winners()
The winners is Albert and Emily.
>>> election.elimination_winners()
The winner is Oscar.
>>> election.de_borda_winners()
The winners is Albert and Emily.
>>> election.de_condorcet_winners()
The winners is Albert and Emily.
>>> election = Election('election_5.xlsx')
>>> print(election)
Number of votes Albert Emily Oscar Maria
1000 1 2 3 4
1000 2 3 1 4
1000 3 1 2 4
>>> election.one_round_winners()
The winners are Albert, Emily and Oscar.
>>> election.two_round_winners()

```

The winners are Albert, Emily and Oscar.

```
>>> election.elimination_winners()
```

The winner is Maria.

```
>>> election.de_borda_winners()
```

The winners are Albert, Emily and Oscar.

```
>>> election.de_condorcet_winners()
```

There is no winner.

```
>>> election = Election('election_6.xlsx')
```

```
>>> print(election)
```

Number of votes Albert Emily Oscar

1000 1 2 3

1000 2 1 3

250 2 3 1

250 3 2 1

```
>>> election.one_r...
```

Slide ID: 378437 | **Index:** 10 | **Title:** Exercise 5: Context free grammars | **Status:** unseen

A context free grammar is a set of production rules of the form: $\text{\$symbol_0\$} \rightarrow \text{\$symbol_1\$} \dots$

$\text{\$symbol_n\$}$ where $\text{\$symbol_0\$}, \dots, \text{\$symbol_n\$}$ are either terminal or nonterminal symbols, with

$\text{\$symbol_0\$}$ being necessarily nonterminal. A symbol is a nonterminal symbol iff it is denoted by a word

built from underscores or uppercase letters. A special nonterminal symbol is called the start symbol. The

language generated by the grammar is the set of sequences of terminal symbols obtained by replacing a

nonterminal symbol by the sequence on the right hand side of a rule having that nonterminal symbol on

the left hand side, starting with the start symbol. For instance, the following, where EXPRESSION is the

start symbol, is a context free grammar for a set of arithmetic expressions: $\text{EXPRESSION} \rightarrow$

$\text{EXPRESSION TERM_OPERATOR TERM}$ $\text{EXPRESSION} \rightarrow \text{TERM TERM} \rightarrow \text{TERM}$

$\text{FACTOR_OPERATOR FACTOR TERM} \rightarrow \text{FACTOR FACTOR} \rightarrow \text{NUMBER FACTOR} \rightarrow$

$(\text{EXPRESSION}) \text{NUMBER} \rightarrow \text{DIGIT NUMBER} \mid \text{DIGIT DIGIT} \rightarrow 0 \dots \text{DIGIT} \rightarrow 9$ $\text{TERM_OPERATOR} \rightarrow$

$+\text{TERM_OPERATOR} \rightarrow -\text{FACTOR_OPERATOR} \rightarrow * \text{FACTOR_OPERATOR} \rightarrow /$ Moreover, blank

characters (spaces or tabs) can be inserted anywhere except inside a number. For instance, $(2 + 3) * (10 -$

$2) - 12 * (1000 + 15)$ is an arithmetic expression generated by the grammar. Note that operators associate

to the left. The grammar is unambiguous, in the sense that every expression generated by the grammar

has a unique evaluation. Write down a program `context_free_grammar.py` that implements a function

`evaluate()` which takes a string representing an expression as an argument, checks whether the

expression can be generated by the grammar, and in case the answer is yes, returns the value of the

expression, provided that no division by 0 is attempted; otherwise, the function returns `None`. Below is a

possible interaction: `$ python3`

`...`

```
>>> from context_free_grammar import *
```

```
>>> evaluate('100')
```

100

```
>>> evaluate('(100)')
```

100

```
>>> evaluate('1 - 20 + 300')
```

281

```
>>> evaluate('((((1))-((20))+((300))))')
```

281

```
>>> evaluate('20 * 4 / 5')
```

16.0

```
>>> evaluate('((((20))*((4)))/((5))))')
```

16.0

```
>>> evaluate('1 + 20 * 30 - 400 / 500')
600.2
>>> evaluate('1 + (20*30-400) / 500')
1.4
>>> evaluate('1+(20 / 30 * 400)- 500')
-232.33333333333337
>>> evaluate('1 + 2 * (3+4*5) / (6*7-8/9)')
2.1189189189189186
>>> evaluate('100')
100
>>> evaluate('100 + ')
100
>>> evaluate('100 + -3')
97
>>> evaluate('100 ÷ 50')
2
>>> evaluate('100 / 0')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "evaluator.py", line 10, in evaluate
    raise ValueError("Division by zero")
ValueError: Division by zero
```

Before you tackle the exercise, find out about recursive descent parsers. To easily tokenise the string, check out the `findall()` function from the `re` module. See also related content [Week 10 - Notes 15 Context Free Grammars](#) discussed in [Week 10 Tuesday Lecture](#).

Lesson 54: Course Outline

Lesson ID: 53286 | **Created:** 2024-05-27T01:25:35.74173+10:00 | **First Viewed:** 2025-10-01T22:13:56.511687+10:00 | **Last Viewed Slide ID:** 361891 | **State:** active | **Status:** completed | **Slide Count:** 1

Slides

Slide ID: 361891 | **Index:** 2 | **Title:** Course Outline | **Status:** completed

Lesson 55: Lectures and Tutorials Timetable

Lesson ID: 53287 | **Created:** 2024-05-27T01:25:35.779904+10:00 | **First Viewed:** 2025-10-01T22:21:21.259214+10:00 | **Last Viewed Slide ID:** 361892 | **State:** active | **Status:** completed | **Slide Count:** 1

Slides

Slide ID: 361892 | **Index:** 1 | **Title:** Lectures and Consultations Timetable | **Status:** completed

Lesson 56: Course Schedule

Lesson ID: 53289 | **Created:** 2024-05-27T01:25:35.872332+10:00 | **State:** active | **Status:** unattempted | **Slide Count:** 1

Slides

Slide ID: 361894 | **Index:** 1 | **Title:** Course Schedule | **Status:** unseen

<https://moodle.telt.unsw.edu.au/mod/page/view.php?id=6563908>

Lesson 57: Summary of Assessment Tasks

Lesson ID: 53290 | Created: 2024-05-27T01:25:35.910369+10:00 | State: active | Status: unattempted | Slide Count: 1

Slides

Slide ID: 361895 | Index: 1 | Title: Summary of Assessment Tasks | Status: unseen

Lesson 58: CSE VLAB Gateway

Lesson ID: 53293 | Created: 2024-05-27T01:25:36.217637+10:00 | State: active | Status: unattempted | Slide Count: 1

Slides

Slide ID: 361907 | Index: 1 | Title: CSE VLAB Gateway | Status: unseen

<https://vlabgateway.cse.unsw.edu.au/>

Lesson 59: Python 3 Cheat Sheet

Lesson ID: 53296 | Created: 2024-05-27T01:25:36.337179+10:00 | State: active | Status: unattempted | Slide Count: 1

Slides

Slide ID: 361910 | Index: 1 | Title: Python 3 Cheat Sheet | Status: unseen

Lesson 60: Ed Discussion - Quick Start Guide

Lesson ID: 53294 | **Created:** 2024-05-27T01:25:36.254708+10:00 | **First Viewed:** 2025-10-01T22:21:33.003774+10:00 | **Last Viewed Slide ID:** 361908 | **State:** active | **Status:** completed | **Slide Count:** 1

Slides

Slide ID: 361908 | **Index:** 1 | **Title:** Ed Discussion - Quick Start Guide | **Status:** completed

Lesson 61: Quiz 4

Lesson ID: 55345 | Created: 2024-06-24T18:05:54.074332+10:00 | State: scheduled | Status: unattempted
| Slide Count: 1

Slides

Slide ID: 374397 | Index: 7 | Title: Quiz 4 | Status: unseen

See PDF file below and stub. This quiz is worth 4 marks. Marking is `_good_prime()` 1.5 marks
`smallest_good_prime()` 2.5 marks ----- Total 4.0 marks Quiz 4 is due
Week 7 Thursday 11 July 2024 @ 9.00pm (Sydney time). Please note that late submission with 5% penalty
per day is allowed up to 3 days from the due date, that is, any late submission after Week 7 Sunday 14
July 2024 @ 9.00pm will be discarded. Please make sure not to change the filename `quiz_4.py` while
submitting by clicking on [Mark] button in Ed. It is your responsibility to check that your submission did go
through properly using Submissions link in Ed otherwise your mark will be zero for Quiz 4.

Lesson 62: Week 3 - Functions

Lesson ID: 53307 | **Created:** 2024-05-27T01:25:37.526309+10:00 | **First Viewed:** 2025-09-18T14:41:47.469366+10:00 | **Last Viewed Slide ID:** 361961 | **State:** active | **Status:** attempted | **Slide Count:** 8

Slides

Slide ID: 361960 | **Index:** 157 | **Title:** Functions | **Status:** completed

Suppose you'd like to print lists of things in the following 'smart' way, which takes into account the number of items and uses commas and 'and' accordingly:

A and B

A, B, and C

A, B, C, and D

You can't just use the `join()` method of a string, because it doesn't do this. You have to write a piece of code. And the code required is reasonably complicated:

Print the list 'lst' in a smart way

```
if len(lst) == 1:
```

```
    print(lst[0])
```

```
elif len(lst) == 2:
```

```
    print(lst[0] + ' and ' + lst[1])
```

```
else:
```

```
    print(', '.join(lst[:-1]) + ', and ' + lst[-1])
```

Having to write this code every time you want to smart-print a list would be a pain. Also, if you find an error in the code, or if you think of a way to improve it, you'll have to find all instances of the code and update them one-by-one.

Fortunately, there is a much better way, and that's to define a function which does this smart printing.

Then whenever you want to smart-print a list you can just call the function and provide it with the list.

Here's how your program might look (don't worry if you don't understand the "def" part - you'll be learning this):

Define the function

```
def smart_print(lst):
```

```
    if len(lst) == 1:
```

```
        print(lst[0])
```

```
    elif len(lst) == 2:
```

```
        print(lst[0] + ' and ' + lst[1])
```

```
    else:
```

```
        print(', '.join(lst[:-1]) + ', and ' + lst[-1])
```

Use it

```
smart_print(['A'])
```

```
smart_print(['A', 'B'])
```

```
smart_print(['A', 'B', 'C'])
```

```
smart_print(['A', 'B', 'C', 'D'])
```

This is much better. If you find an error in the code you only need to fix it in one place. If you think of a way to improve it, you only need to improve it in one place. And, as a bonus, the function name itself makes your code more self-documenting - each time you invoke the function its name makes it clear what you are doing, with no need for any comments. Brilliant!

Defining and using functions in this way is an example of code modularisation. It is an important and powerful technique, and is one of the cornerstones of good programming. You'll learn how to do it this week.

Slide ID: 361961 | **Index:** 158 | **Title:** Defining functions | **Status:** completed

So far you've been using functions that are built-in to Python, such as `input()`, `print()`, and `len()`. Like many other languages, Python allows you to define your own functions.

Defining a function You can define a function using a `def` statement, which has the following form:

```
def ():
```

```
    For example: def say_hello():
```

```
    print('Hello')
```

Once you've defined the function you can call it like any other function:

```
def say_hello():
```

```
print('Hello')
```

say_hello() Notice that the code inside the body of the function is not executed when the function is defined - it is only executed when the function is called. Also notice that you must define the function before calling it: say_hello() # Error - the function has not yet been defined

```
def say_hello():
```

```
print('Hello') Naming functions
```

The rules for naming functions are the same as for naming variables. Although it is not required, it has become conventional to use snake case - lower case words, separated by underscores, for example say_hello. You should choose names that help to document your code - naming the above function say_hello, for example, is more explanatory than naming it hello, or, even worse, my_func.

Adding parameters You can add parameters to a function, to specify that the function should receive one or more arguments when it is called. You do this by adding parameter names in the brackets after the function name: def say_hello(name): # Add a parameter called "name"

```
def say_hello(name): # Add a parameter called "name"
```

```
print('Hello,', name)
```

say_hello('James') # Provide 'James' as an argument When you call a function you must supply it with the right number of arguments - one for each parameter. If the function has no parameters then you must supply no arguments; if the function has one parameter then you must supply exactly one argument; and so on. If you supply the wrong number of arguments then Python will raise an error. The say_hello()

function defined above has one parameter, so you must supply it with exactly one argument: def say_hello(name):

```
def say_hello(name):
```

```
say_hello(name):
```

```
print('Hello,', name)
```

say_hello() # Error - not enough arguments def say_hello(name):

```
def say_hello(name):
```

```
print('Hello,', name)
```

say_hello('James', 'Sarah') # Error - too many arguments Adding default values You can give parameters

default values. If no argument is provided for that parameter then the function will use the default value. def say_hello(name = 'James'): # Give the parameter a default value

```
def say_hello(name = 'James'): # Give the parameter a default value
```

```
print('Hello,', name)
```

say_hello() # The default value will be used

say_hello('Sarah') # 'Sarah' will be used instead You can use this feature to make arguments optional -

Python won't raise an error if the argument is not supplied, it will just use the default value. If you set the default value to None then you can use this to detect whether an argument was supplied for the

parameter: def say_hello(name = None):

```
def say_hello(name = None):
```

```
if name is None:
```

```
print('No name was provided')
```

```
else:
```

```
print('Hello,', name)
```

say_hello()

say_hello('Sarah') Returning values A function always returns a value. By default it will return the object

None, but you can use a return statement to get it to return whatever value you want. def sum(x, y):

```
def sum(x, y):
```

```
return x + y # Specify a return value
```

print(sum(1, 2)) The function will exit immediately after a return statement, so any further statements in the

function body will not be executed. def sum(x, y):

```
def sum(x, y):
```

```
return x + y # The function exits here
```

```
print('This will not be printed') # Not executed
```

print(sum(1, 2)) You can have multiple return statements (but only one will get executed): def grade(mark):

```

if mark >= 50:
    return 'Pass'
else:
    return 'Fail' # Only one of these return statements will be executed

```

```

print(grade(73))
print(grade(35))

```

A function can only return one value. This value can, however, be a collection - a list, or a tuple, or a set, or a dictionary. It is fairly common to return a tuple. Here's an example in which a tuple with two elements is returned:

```

def ends(string):
    first_char = string[0]
    last_char = string[-1]
    # Return a tuple with two elements
    # Note that only the comma is needed - round brackets are ass...

```

Slide ID: 361962 | **Index:** 159 | **Title:** Variable scope | **Status:** unseen

If you create a variable inside a function then that variable is only defined inside the function. We say that the variable's scope is limited to the function, or that the variable is locally defined. If you try to use a variable outside its scope then Python will raise an error.

```

def my_func():

```

```

x = 5

```

```

print(x) # Error - x is only defined inside the function

```

Even if you have used the same variable name outside the function, changes to variables defined inside the function are limited to occurring inside the function. This can be a problem if you use locally defined variables with the same name as globally defined variables (i.e. variables not declared within the scope of a function) - this is known as variable shadowing.

```

x = 3 # Globally defined x

```

```

def my_func():
    x = 5 # Locally defined x, no change to the globally defined x

```

```

my_func() # No change to the globally defined x
print(x) # Prints 3, not 5

```

If you want to use globally defined variables inside functions, the safest approach is to provide them to the function as arguments.

```

x = 3

```

```

def my_func(y):
    return y + 2 # Add 2 to the number provided and return the result

```

```

x = my_func(x) # Assign to x the result of of my_func(x) - changes x
print(x) # Prints 5

```

Slide ID: 361963 | **Index:** 161 | **Title:** Nested functions | **Status:** unseen

You can define a function inside another function. When you do, the inside function is called a nested function. Here's an example:

```

def acronym(string):

```

```

    result = ""
    words = string.split(' ')
    def upper_first(string): # A nested function
        return string[0].upper()
    for word in words:
        result += upper_first(word)
    return result

```

```

print(acronym('World Health Organisation'))

```

Because a nested function is defined inside an enclosing

function, it is only available to be called inside that enclosing function. The following program generates an error, because the nested function is called outside its enclosing function:

```
def acronym(string):
    result = ""
    words = string.split(' ')
    def upper_first(string): # Only available inside acronym
        return string[0].upper()
    for word in words:
        result += upper_first(word)
    return result

print(upper_first('hello')) # Error - upper_first is not available here
```

Slide ID: 361964 | **Index:** 162 | **Title:** Lambda functions | **Status:** unseen

You can refer to a function without giving it a name. Suppose, for example, you have a list of names and you want to sort those names by their last letter. You can use the list's `sort()` method to do this. By default, `sort()` sorts them alphabetically, but you can override this default by providing a function to use as the sorting key. If you like, you can first define the function, giving it a name, and then provide it by name to `sort()`:

```
names = ['Geoff', 'Kim', 'Louise', 'Tam', 'Helen']
def last_letter(name):
    return name[-1]
names.sort(key = last_letter) # Use the function defined above
print(names)
```

But you don't need to. You can refer to the function directly when you call `sort()`, without giving it a name. You do this by using a lambda function:

```
names = ['Geoff', 'Kim', 'Louise', 'Tam', 'Helen']
names.sort(key = lambda name: name[-1]) # Use a lambda function
print(names)
```

A lambda function is an expression (not a statement) whose value is a function. You can think of a lambda function as being a function literal. The syntax of a lambda function is as follows:

lambda : Note that there is no return in a lambda function. Lambda functions can have more than one parameter. Here's a lambda function with two parameters:

```
lambda a, b: a + b
```

You can use a lambda function just like you use function names. You can call the function it refers to by using the usual round brackets notation (note that you typically need to put parentheses around the lambda function when you call it, to avoid confusion with neighbouring code):

```
print((lambda a, b: a + b)(2, 4))
```

And you can use it to assign a value to a variable:

```
f = lambda a, b: a + b
print(f(2, 4))
```

Note what is going on in this last example. We are using the lambda function `lambda a, b: a + b` to assign a value to a variable `f`. The value of the lambda function is a function - you can think of it as a literal for that function. So `f` is being assigned a function. We can then use `f` like any other function name. In line 2, we call the function, using `f(2, 4)`. Compare the above with the following example:

```
def f(a, b):
    return a + b
print(f(2, 4))
```

The two examples are similar, but there are some subtle differences. In both examples we end up with `f` being the name of a function. But we get there in two different ways. In the first example, we assign `f` the function using an assignment statement and a lambda function that refers to the function. In the second example, we define `f` using a `def` statement that defines the function.

Slide ID: 361965 | **Index:** 163 | **Title:** Functions are objects | **Status:** unseen

Functions are objects, and you can use them in the same way you use other objects, such as numbers, strings, lists, and so on. Just as you might set a variable's value to a number, such as 1, you might also set it to a function, such as `len()`. Also:

- You can assign a function to a variable
- A function can be an attribute of an object
- A function can be an element of a collection
- Functions can be keys in a dictionary
- You can pass a function as an argument of a function call
- You can return a function as the result of a function call
- And so on.

Because of this we say that Python functions are first class. Because you can supply functions as arguments to functions, you can create functions that operate on functions:

```
def add(x, y):
    return x + y
def subtract(x, y):
```

```

return x - y
def apply(f, x, y): # This function applies function f to values x and y
return f(x, y)

```

```

print(apply(add, 10, 1))
print(apply(subtract, 10, 1))

```

Here's another example. In this case we define a function `compose`, which takes two functions `f` and `g` as arguments and returns a function - the composition of `f` and `g`, which is the function that takes an argument `x` and returns `f(g(x))`:

```

def add1(x):
return x + 1
def subtract1(x):
return x - 1
def compose(f, g): # This function returns a function
return lambda x: f(g(x))

```

```

add2 = compose(add1, add1) # add2 is a new function
print(add2(10))
do_nothing = compose(add1, subtract1) # do_nothing is a new function
print(do_nothing(10))

```

Functions that take functions as arguments, or return functions as values, are known as higher order functions.

Slide ID: 361966 | **Index:** 164 | **Title:** Generators | **Status:** unseen

Before we leave the topic of functions, there is one special type of function that you should know about. Suppose you have a function that returns a collection of objects, perhaps a list. Suppose it is the following one:

```

def squares():
result = []
for x in range(10):
result.append(x**2)
return(result)

```

```

print(squares())

```

The function returns the full list, which you can then iterate over:

```

def squares():
result = []
for x in range(10):
result.append(x**2)
return(result)

```

```

for x in squares():
print(x)

```

Rather than getting the function to return the full list, you can get it to return the elements one at a time, by using a `yield` statement instead of a `return` statement:

```

def squares():
for x in range(10):
yield x**2 # Use a yield statement

```

```

for x in squares():
print(x)

```

Notice what happens when you print:

```

def squares():
for x in range(10):
yield x**2

```

The function now returns a special kind of object, called a generator - it does not return the full list. This generator object generates the elements as they are needed. Why would you do this, rather than have the function return the full list at the outset? If the list is large, and if you don't need its elements all at once, then it is a good way to save memory. Generator expressions

There is an even more concise way to make a generator. Rather than defining a function that returns a generator, you can use a

generator expression. It is exactly like a comprehension, but you use round brackets. This is why there is no tuple comprehension - the round brackets are used for generator expressions instead.

```
squares = (x**2  
for x in range(10)) # Get a generator from a generator expression
```

```
print(squares)  
for x in squares:  
    print(x)
```

Slide ID: 361967 | **Index:** 165 | **Title:** Further reading | **Status:** unseen

You might find the following helpful: [The Python Tutorial at w3schools.com](https://www.w3schools.com/python/python_generators.asp)

Lesson 63: Week 3 - Overview

Lesson ID: 53306 | **Created:** 2024-05-27T01:25:37.436833+10:00 | **First Viewed:** 2025-10-01T22:21:39.889853+10:00 | **Last Viewed Slide ID:** 361959 | **State:** active | **Status:** completed | **Slide Count:** 2

Slides

Slide ID: 361958 | **Index:** 9 | **Title:** Week 3 Tuesday To Do List | **Status:** completed

Week 3 Tuesday To Do ListAdmin/TipsAssignment 1 worth 13 marks will be released today @ 7.15pmQuiz 1 worth 4 marks is due Week 3 Thursday @ 9pmContentContinue "Week 2 - Lists, Tuples, Sets, and Dictionaries" lessonAdding elementsAdding to a list: append(), insert(), extend(), and using + operatorNo adding to tuples since they are immutableAdding to a set: add() and update()Adding to a dictionary by specifying a value for a new key or updating itRemoving elementsRemoving from a list: del, set slice to empty list, pop(), remove(), and clear()No removing from tuples since they are immutableRemoving from a set: remove(), discard(), and clear()Removing from a dictionary: del, pop(), and clear()Modifying elementsModifying list elements: assignment and sliceNo tuples modification since they are immutableModifying set elements: can not be changed, but remove then addModifying dictionary elements: using assignment similarly to list elementsSorting elementssort() method (lists only and in-place)sorted() method (applies to all and returns a list)Joining elements using join() string methodSpecial string operationsStrings as tuplesSplitting stringsSpecial set operations: union, intersection, difference, symmetric difference, comparing setsComprehensionsFiles as lists: readlines(), writelines(), reading CSV filesDates and timesStart "Functions" lessonFunctionsDefining functionsVariable scopeNested functionsLambda functionsFunctions are objectsGeneratorsUseful LinksFor examples of Flow of Execution, see:
<https://runestone.academy/ns/books/published/thinkcspy/Functions/FlowofExecutionSummary.html>Python List sort() Method <https://www.programiz.com/python-programming/methods/list/sort>Python sorted() Method<https://www.programiz.com/python-programming/methods/built-in/sorted>

Slide ID: 361959 | **Index:** 11 | **Title:** Week 3 Thursday To Do List | **Status:** completed

Week 3 Thursday To Do ListAdmin/TipsAssignment 1 discussionQuiz 2 worth 4 marks will be released today @ 7.15pmQuiz 1 worth 4 marks is due today @ 9pmContentContinue "Functions" lessonDefining functionsVariable scopeNested functionsLambda functionsFunctions are objectsGeneratorsWeek 3 - Notes 2 The Monty Hall Problemfunctions from the random module (random(), choice(), randrange(), seed(), ...)input() functionint() functionexceptionsbreak statementistitle() and lower() str class methodssets, sets vs dictionarieschoice() functionrandrange() functionremove() and pop() list class methodsmultiple assignmentsfunctions with default parameter values formatted strings12 possible outcomes:
winning door A A A A B B B B C C C C
first chosen A A B C B B A C C C A B
opened door B C C B A C C A B B A
24 possible outcomes:
winning door A A A A A A A B B B B B B B C C C C C C C
first chosen A A A A B B C C B B B B A A C C C C C C A A B B
opened door B B C C C C B B A A C C C C A A A A B B B B A A
2nd chosen door A C A B A B A C B C B A B A B C C B C A C A C B
Useful Linksdoctest — Test interactive Python examples (will be used for final exam questions)<https://docs.python.org/3/library/doctest.html>

Lesson 64: How to Use Jupyter Notebook: A Beginner's Tutorial

Lesson ID: 53292 | Created: 2024-05-27T01:25:36.029154+10:00 | State: active | Status: unattempted | Slide Count: 1

Slides

Slide ID: 361906 | Index: 5 | Title: How to Use Jupyter Notebook: A Beginner's Tutorial | Status: unseen

<https://www.dataquest.io/blog/jupyter-notebook-tutorial/>

Lesson 65: Week 9 - Overview

Lesson ID: 53327 | Created: 2024-05-27T01:25:42.984426+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 362055 | Index: 9 | Title: Week 9 Tuesday To Do List | Status: unseen

Week 9 Tuesday To Do ListAdmin/TipsQuiz 6 worth 4 marks is due Week 9 Thursday 25/7/24 @ 9pmAssignment 2 worth 13 marks is due Week 11 Monday 5/8/24 @ 10amFinal Exam worth 50 marks to be held on Wednesday 21 August 2024ContentWeek 9 Notes 13 Quadratic EquationsProvide deep understanding of object oriented design and syntaxKeyword argumentsPositional argumentsFunctional designPackage designObject-oriented designSolve Sample Exam Question 2Discuss Assignment 2Discuss Quiz 6

Slide ID: 362056 | Index: 11 | Title: Week 9 Thursday To Do List | Status: unseen

Week 9 Thursday To Do ListAdmin/TipsQuiz 6 worth 4 marks is due today @ 9pmAssignment 2 worth 13 marks is due Week 11 Monday 5/8/24 @ 10amFinal Exam worth 50 marks to be held on Wednesday 21 August 2024There will be two (2) back to back sessions (morning and afternoon) with corralling for the afternoon session. Morning students can't leave until the afternoon students are properly corralledSession preference form released to students today at 4.23pm (check your email):<https://cgi.cse.unsw.edu.au/~exam/24T2/seating/register.cgi>Closes Midday Week 10 Friday 2nd August 2024Seating allocations released Mid Week 11 (Study Break) together with exact time and locationExam environment for practice open until Week 11 Tuesday 6 August 23.59pm. Content Week 9 Notes 14 Levenshtein DistanceSee also <https://edstem.org/au/courses/16645/workspaces/ppDT8Elc4fQLFpiCFpFUBncfwC0ulpsm>Solve Sample Exam Questions 5 and 7Discuss Assignment 2Discuss Quiz 6Useful links about Levenshtein distanceLevenshtein distancehttps://en.wikipedia.org/wiki/Levenshtein_distanceLevenshtein Distance, in Three Flavors <https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm>Another Python version of Levenshtein distance https://folk.idi.ntnu.no/mlh/hetland_org/coding/python/levenshtein.pyLevenshtein Distancehttps://python-course.eu/levenshtein_distance.php

Lesson 66: Week 7

Lesson ID: 53338 | Created: 2024-05-27T01:25:48.687891+10:00 | State: active | Status: unattempted | Slide Count: 5

Slides

Slide ID: 362124 | Index: 92 | Title: Exercise 1: A circle class | Status: unseen

Write a program in which you define a class `Circle`, to facilitate working with circles, and then illustrate its use. Your `Circle` class should: Have an `__init__` special method that allows you to create a `Circle` object by supplying a number which is the radius of the circle. This should be stored in an attribute `radius`. The method should check that the supplied value is a valid radius (i.e. a non-negative float), and deal with invalid values appropriately. Have a `__str__` special method that allows you to print a `Circle` object in an informative way (for example, if the radius is 5 it returns `Circle of radius 5`). Have an instance method called `circumference` that returns the circumference of a `Circle` object. Have an instance method called `area` that returns the area of a `Circle` object. Have `__eq__`, `__lt__`, `__le__`, `__gt__`, and `__ge__` special methods that allow you to compare two `Circle` objects. You should compare them by radius, so that two `Circle` objects are equal when they have the same radius; one `Circle` object is less than another `Circle` object when it has a smaller radius; and so on. Make appropriate use of docstrings. To illustrate the use of your class, here is the kind of code you might add to your program, after your class definition:

```
# Inspect the documentation
print(Circle.__doc__) # Output: A class to facilitate working with circles
print(Circle.circumference.__doc__) # Output: Returns the circumference of the circle
print(Circle.area.__doc__) # Output: Returns the area of the circle
```

```
# Create some Circle objects
circle_1 = Circle(4)
circle_2 = Circle(5.3)
circle_3 = Circle(-2) # Output: an appropriate error message
circle_4 = Circle('a') # Output: an appropriate error message
```

```
# Print them
print(circle_1) # Output: Circle of radius 4
print(circle_2) # Output: Circle of radius 5.3
```

```
# Get their circumference and area
print(circle_1.circumference()) # Output: 25.13
print(circle_1.area()) # Output: 50.27
print(circle_2.circumference()) # Output: 33.3
print(circle_2.area()) # Output: 88.25
```

```
# Compare them
print(circle_1 == circle_2) # Output: False
print(circle_1 > circle_2) # Output: False
```

As a guide, use the `Length` class that was defined in Week 5 content as an illustrative example.

Slide ID: 362125 | Index: 93 | Title: Exercise 2: A text class | Status: unseen

Write a program in which you define a class `Text`, to facilitate working with pieces of text. Define your class such that: Each instance of `Text` has an attribute `words` which holds the words of the text in an array, a method `num_words` which returns the number of words in the text, a method `num_chars` which returns the total number of characters in the words, and a method `word_length` which returns the average number of characters per word (as a float rounded to one decimal place). Get your program to ask the user for a piece of text, and then, using the class you have defined, tell the user the number of words in the text, the

number of characters in the text, and the average number of characters per word in the text. Example: Please enter your text: The quick brown fox jumped over the lazy dog.

Number of words: 9

Number of characters: 36

Average word length: 4.0

Note: you might find it helpful to have your class clean the text before splitting it into words, in the way that you did in the Week 3 Exercise 1: Word frequency practice exercise.

Slide ID: 378413 | **Index:** 94 | **Title:** Exercise 3: A location class | **Status:** unseen

Sometimes we work with data about things that have a location, given as a latitude and a longitude. Define a class to help us work with locations. Give a location two data attributes: lat, and lon. Give also location the following three method attributes: hemisphere(), which returns which hemisphere the location is in: Northern Hemisphere if lat > 0 Southern Hemisphere if lat < 0 Equator if lat = 0 zone(), which returns which zone the location is in, defined by the tropics and the arctic/antarctic circles: North Frigid Zone if lat >= 66.57 North Temperate Zone if lat >= 23.43 Tropical Zone if lat >= -23.43 South Temperate Zone if lat >= -66.57 South Frigid Zone if lat < -66.57 direction_to(), which returns the direction to another location: North if lat < another lat East if lon < another lon West if lon > another lon Add some error checking. For instance, latitudes must be between -90 and 90, and longitudes must be between -180 and 180. To illustrate the use of your class, here is the kind of code you might add to your program, after your class definition:

```
sydney = Location(-33.87, 151.21)
wellington = Location(-41.28, 174.77)
hobart = Location(-42.88, 147.33)
stockholm = Location(59.33, 18.07)
```

```
print(f"Sydney is in the {sydney.zone()}") # Sydney is in the South Temperate Zone
print(f"Stockholm is in the {stockholm.zone()}") # Stockholm is in the North Temperate Zone
```

```
print(f"To get from Sydney to Stockholm you need to travel {sydney.direction_to(stockholm)}") # To get
from Sydney to Stockholm you need to travel North West
print(f"To get from Hobart to Wellington you need to travel {hobart.direction_to(wellington)}") # To get from
Hobart to Wellington you need to travel North East
```

```
nowhere = Location(-200, 151.21) # The output is shown below
# Traceback (most recent call last):
# File "/home/main.py", line 41, in
# nowhere = Location(-200, 151.21)
# ~~~~~
# File "/home/main.py", line 7, in __init__
# raise Exception("Invalid coordinates")
# Exception: Invalid coordinates
```

Slide ID: 378419 | **Index:** 96 | **Title:** Exercise 4: A temperature module | **Status:** unseen

There are many different units in which temperature can be measured. Three of the most common are Celsius, Fahrenheit, and Kelvin (used a lot in science). Your task is to define a Temperature class, which we can use to more easily convert temperatures from one scale to another, and to compare temperatures that are on different scales. The conversions between the scales go as follows: Converting to Celsius:

Celsius = (Fahrenheit - 32) × 5/9

Celsius = Kelvin - 273.15

Converting from Celsius:

Fahrenheit = (9/5 × Celsius) + 32

Kelvin = Celsius + 273.15

Your Temperature class should: Have an `__init__` special method that allows you to create a Temperature object by supplying a number and a unit: either Celsius (C), Fahrenheit (F), or Kelvin (K). Have a `__str__` special method that allows you to print a Temperature object in an informative way. Have an instance method called `to` that has a parameter for a unit, an optional parameter for a number of decimal places, and returns the temperature of the instance in the given unit, rounded to the given number of decimal places, if any were given, otherwise not rounded. Have `__eq__`, `__lt__`, `__le__`, `__gt__`, and `__ge__` special methods that allow you to compare two Temperature objects. Make appropriate use of docstrings. Save your class in a module called `temperature.py`. Import this module into `code.py`, and add some code to `code.py` that illustrates the use of your Temperature class. Example: Here is the kind of code you might use in `code.py` to illustrate your class:

```
# Create Temperature objects
temp_1 = Temperature(32, 'C')
temp_2 = Temperature(100, 'F')
temp_3 = Temperature(324, 'K')
```

Print them

```
print(temp_1) # Outputs Temperature: 32C
print(temp_2) # Outputs Temperature: 100F
print(temp_3) # Outputs Temperature: 324K
```

Convert them

```
print(temp_1.to('F')) # Outputs 89.6
print(temp_2.to('K', 3)) # Outputs 310.928
print(temp_3.to('C', 1)) # Outputs 50.9
```

Compare them

```
print(temp_1 == temp_2) # Outputs False
print(temp_1 < temp_2) # Outputs False
print(temp_1 >= temp_2) # Outputs False
```

As a guide, use the Length class that was defined in Week 5 content as an illustrative example.

Slide ID: 362127 | **Index:** 103 | **Title:** Exercise 5: Mediants | **Status:** unseen

Let two distinct reduced positive fractions $F_1 = \frac{p_1}{q_1}$ and $F_2 = \frac{p_2}{q_2}$ be given, with the denominator set to 1 in case the fraction is 0. The median of F_1 and F_2 is defined as $\frac{p_1 + p_2}{q_1 + q_2}$; it is also in reduced form, and sits between F_1 and F_2 . Let a reduced fraction $F = \frac{p}{q}$ in $(0, 1)$ be given. It can be shown that starting with $\frac{0}{1}$ and $\frac{1}{1}$, one can compute a finite number of mediants and eventually generate F . More precisely, there exists $n \in \mathbb{N}$ and a sequence of pairs of fractions $(F_1^i, F_2^i)_{i \leq n}$ such that: $F_1^0 = \frac{0}{1}$ and $F_2^0 = \frac{1}{1}$; for all i F is the mediant of F_1^{i-1} and F_2^{i-1} . The program `mediants.py` defines a function `mediants_to()` that given as arguments two strictly positive integers p and q with $p < q$ with `mediants_to()` with your code, possibly defining other functions.

Lesson 67: Assignment 2

Lesson ID: 56004 | **Created:** 2024-07-02T22:43:20.128534+10:00 | **First Viewed:** 2025-07-29T10:32:34.225544+10:00 | **Last Viewed Slide ID:** 378866 | **State:** scheduled | **Status:** unattempted | **Slide Count:** 1

Slides

Slide ID: 378866 | **Index:** 2 | **Title:** Assignment 2 | **Status:** seen

See PDF file below and stub. This Assignment 2 is worth 13 marks distributed as follows: Marking Subtotal

__init__() method 3.0 marks

Incorrect input 1.5

Input does not represent a labyrinth 1.5

display_features() method 10.0 marks

gates 1.5

walls that are all connected 1.5

inaccessible inner points 1.5

accessible areas 1.5

accessible cul-de-sacs 2.0

entry-exit paths 2.0

Total 13.0 marks

Assignment 2 is due Week 11 Monday 5 August 2024 @ 10:00am (Sydney time) Please note that late submission with 5% penalty per day is allowed up to 5 days from the due date, that is, any late submission after Week 11 Saturday 10 August 2024 @ 10:00am will be discarded. Please make sure not to change the filename labyrinth.py while submitting by clicking on [Mark] button in Ed. It is your responsibility to check that your submission did go through properly using Submissions link in Ed otherwise your mark will be zero for Assignment 2.

Lesson 68: Week 7 - Plotting with Matplotlib

Lesson ID: 56296 | **Created:** 2024-07-09T16:26:31.17487+10:00 | **First Viewed:** 2025-09-25T11:16:07.37238+10:00 | **Last Viewed Slide ID:** 381285 | **State:** active | **Status:** attempted | **Slide Count:** 3

Slides

Slide ID: 381285 | **Index:** 19 | **Title:** The Matplotlib family | **Status:** completed

We will be looking at how to visualise data using Matplotlib, a powerful Python plotting library with which you can quickly generate plots of your data. It is inspired by Matlab, and many of its objects are similarly named. Hence, if you are familiar with Matlab then you should feel at home using Matplotlib. Matplotlib provides a module called pyplot as a convenient way to access the Matplotlib functionality, and it is actually with pyplot that we do the plotting. To use pyplot you must import it. Note that pyplot is part of the Matplotlib library, but since it is the only part of Matplotlib that we will be using we can use a more targeted import. It is standard to use the alias plt for the pyplot object: `import matplotlib.pyplot as plt`. Plotting with pyplot via pandas. Working directly with pyplot can be a bit laborious. Pandas provides its own `plot()` function as a method of series and data frames, which automatically performs many of the common tasks involved in using pyplot. Because the `plot()` method is part of pandas, you do not need to import anything other than pandas to use it. It is standard to use the alias `pd`: `import pandas as pd`. Plotting with pyplot via seaborn. Although pandas' `plot()` method simplifies the process of plotting with Matplotlib, there are some ways in which it is still limited. Seaborn is a library that is designed to further simplify the task of using pyplot. It is not part of pandas, but it is designed to work well with pandas. It is particularly good for working with categorical (i.e., non-numerical) data. To use seaborn you must import it. It is standard to use the alias `sns`: `import seaborn as sns`. Seaborn has some very nice plotting styles. You can set a style by calling seaborn's `set()` function, and if you do not specify which style you would like then seaborn will just set its default style: `import seaborn as sns`
`sns.set()`. Keep in mind that calling `set()` will affect the style of all matplotlib plots, not just those you create using seaborn. The seaborn website has an excellent example gallery of plots, with the code that is used to produce them.

Slide ID: 381287 | **Index:** 22 | **Title:** Working directly with pyplot | **Status:** unseen

Although it is easier to do plotting with pandas and seaborn, we will go through some basic plotting directly with pyplot. This will give you a better understanding of what is going on behind the scenes when you plot with pandas and seaborn. An example. Suppose you have the following (fictitious) data about quarterly unemployment rates for NSW and VIC, loaded into a Python dictionary: `data = {'NSW': {'Q1': 3.2, 'Q2': 3.4, 'Q3': 3.4, 'Q4': 3.6}, 'VIC': {'Q1': 3.5, 'Q2': 3.4, 'Q3': 3.0, 'Q4': 3.1}}`. The following program uses pyplot to create line plots of these quarterly figures, one line for each state. `import matplotlib.pyplot as plt`

```
data = {
'NSW': {'Q1': 3.2, 'Q2': 3.4, 'Q3': 3.4, 'Q4': 3.6},
'VIC': {'Q1': 3.5, 'Q2': 3.4, 'Q3': 3.0, 'Q4': 3.1},
}
```

```
# Create a new figure and call it 'fig'
fig = plt.figure()
```

```
# Add an axes to the figure and call it 'ax'
ax = fig.add_subplot()
```

```
# Add a line plot to ax
# Use the quarters as the x-values and the NSW percentages as the y-values
ax.plot(data['NSW'].keys(), data['NSW'].values())
```

```
# Add a line plot to ax
# This time use the VIC percentages as the y-values
ax.plot(data['VIC'].keys(), data['VIC'].values())
```

```
# Save the figure
# This step is necessary for getting the plot to show here in Ed
fig.savefig('plot.png')To get a plot to show here in Ed, you must save it using fig.savefig(). You can name
the plot whatever you want.Adding some featuresIt would be better if we added a figure title, some line
labels and a legend, and some axis labels:import matplotlib.pyplot as plt
```

```
data = {
'NSW': {'Q1': 3.2, 'Q2': 3.4, 'Q3': 3.4, 'Q4': 3.6},
'VIC': {'Q1': 3.5, 'Q2': 3.4, 'Q3': 3.0, 'Q4': 3.1},
}
```

```
fig = plt.figure()
```

```
# Add a figure title
fig.suptitle('Unemployment Rates')
```

```
ax = fig.add_subplot()
```

```
# Specify labels this time
ax.plot(data['NSW'].keys(), data['NSW'].values(), label='NSW')
ax.plot(data['VIC'].keys(), data['VIC'].values(), label='VIC')
```

```
# Show a legend
ax.legend()
```

```
# Specify axis labels
ax.set_xlabel('Quarter')
ax.set_ylabel('Unemployment (%)')
```

fig.savefig('plot.png')Using multiple axesIn the figure above, both line plots were drawn on the same axes. You can draw them on separate axes instead, by adding two axes to the figure and specifying how they should be laid out.import matplotlib.pyplot as plt

```
data = {
'NSW': {'Q1': 3.2, 'Q2': 3.4, 'Q3': 3.4, 'Q4': 3.6},
'VIC': {'Q1': 3.5, 'Q2': 3.4, 'Q3': 3.0, 'Q4': 3.1},
}
```

```
fig = plt.figure()
fig.suptitle('Unemployment Rates')
```

```
# Add an axes. It's the first axes of a 1 x 2 grid of axes.
ax1 = fig.add_subplot(1, 2, 1)
```

```
# No need for a label this time
```

```

ax1.plot(data['NSW'].keys(), data['NSW'].values())

# Specify a title for the axes, and labels for the x- and y-axis
ax1.set_title('NSW')
ax1.set_xlabel('Quarter')
ax1.set_ylabel('Unemployment (%)')

# Add an axes. It's the second axes of a 1 x 2 grid of axes.
ax2 = fig.add_subplot(1, 2, 2)

# No need for a label this time
ax2.plot(data['VIC'].keys(), data['VIC'].values())

# Specify a title for the axes, and labels for the x- and y-axis
ax2.set_title('VIC')
ax2.set_xlabel('Quarter')
ax2.set_ylabel('Unemployment (%)')

fig.savefig('plot.png')

```

Some finishing touches

```

import matplotlib.pyplot as plt
data = {
'NSW': {'Q1': 3.2, 'Q2': 3.4, 'Q3': 3.4, 'Q4': 3.6},
'VIC': {'Q1': 3.5, 'Q2': 3.4, 'Q3': 3.0, 'Q4': 3.1},
}

# Set the size to be 10 inches wide by 8 inches tall
fig = plt.figure(figsize=[10, 8])

fig.suptitle('Unemployment Rates')
ax1 = fig.add_subplot(1, 2, 1)
ax1.plot(data['NSW'].keys(), data['NSW'].values())
ax1.set_title('NSW')
ax1.set_xlabel('Quarter')
ax1.set_ylabel('Unemployment (%)')

# Set the y-axis values to go from 3 to 4
ax1.set_ylim(3, 4)

# Set the y-axis ticks to be 3.0, 3.1, 3.2, ..., 4.0
ax1.set_yticks([x/10 for x in range(30, 41)])

# Show gridlines on the axes
ax1.grid()

# Tell ax2 to share its y-axis with ax1
ax2 = fig.add_subplot(1, 2, 2, sharey=ax1)

# Specify the colour of the line
ax2.plot(data['VIC'].keys(), data['VIC'].values(), color='red')

```


Example 1: Simple plotimport matplotlib.pyplot as plt

```
x_numbers = [1, 6, 3]
y_numbers = [2, 4, 6]
```

```
plt.plot(x_numbers, y_numbers)
```

```
plt.savefig('a.png')Example 2: Adding a markerimport matplotlib.pyplot as plt
```

```
x_numbers = [1, 6, 3]
y_numbers = [2, 4, 6]
```

```
plt.plot(x_numbers, y_numbers, '+')
```

```
plt.savefig('a.png')Example 3: Changing the markerimport matplotlib.pyplot as plt
```

```
x_numbers = [1, 6, 3]
y_numbers = [2, 4, 6]
```

```
plt.plot(x_numbers, y_numbers, marker='*')
```

```
plt.savefig('a.png')Example 4: Annual temperatures in NYCimport matplotlib.pyplot as plt
```

```
nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
```

```
plt.plot(nyc_temp, marker='o')
```

```
plt.savefig('a.png')Example 5: Annual temperatures in NYC with yearsimport matplotlib.pyplot as plt
```

```
nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
```

```
years = range(2000, 2013)
```

```
plt.plot(years, nyc_temp, marker='o')
```

```
plt.savefig('a.png')Example 6: Comparing multiple datasetsimport matplotlib.pyplot as plt
```

```
# New York City temperatures (farenheight)
```

```
temp_2000 = [31.3, 37.3, 47.2, 51.0, 63.5, 71.3, 72.3, 72.7, 66.0, 57.0, 45.3, 31.1]
```

```
temp_2006 = [40.9, 35.7, 43.1, 55.7, 63.1, 71.0, 77.9, 75.8, 66.6, 56.2, 51.9, 43.6]
```

```
temp_2012 = [37.3, 40.9, 50.9, 54.8, 65.1, 71.0, 78.8, 76.7, 68.8, 58.0, 43.9, 41.5]
```

```
months = range(1, 13)
```

```
plt.plot(months, temp_2000, months, temp_2006, months, temp_2012)
```

```
plt.savefig('a.png')Example 7: Multiple datasets with legendsimport matplotlib.pyplot as plt
```

```
# New York City temperatures (farenheight)
```

```
temp_2000 = [31.3, 37.3, 47.2, 51.0, 63.5, 71.3, 72.3, 72.7, 66.0, 57.0, 45.3, 31.1]
```

```
temp_2006 = [40.9, 35.7, 43.1, 55.7, 63.1, 71.0, 77.9, 75.8, 66.6, 56.2, 51.9, 43.6]
```

```
temp_2012 = [37.3, 40.9, 50.9, 54.8, 65.1, 71.0, 78.8, 76.7, 68.8, 58.0, 43.9, 41.5]
```

```
months = range(1, 13)
```

```
plt.plot(months, temp_2000, months, temp_2006, months, temp_2012)
```

```
plt.legend([2000, 2006, 2012])
```

```
plt.savefig('a.png')Example 8: Adding a title and labelsimport matplotlib.pyplot as plt
```

```
# New York City temperatures (farenheight)
temp_2000 = [31.3, 37.3, 47.2, 51.0, 63.5, 71.3, 72.3, 72.7, 66.0, 57.0, 45.3, 31.1]
temp_2006 = [40.9, 35.7, 43.1, 55.7, 63.1, 71.0, 77.9, 75.8, 66.6, 56.2, 51.9, 43.6]
temp_2012 = [37.3, 40.9, 50.9, 54.8, 65.1, 71.0, 78.8, 76.7, 68.8, 58.0, 43.9, 41.5]
```

```
months = range(1, 13)
```

```
plt.plot(months, temp_2000, months, temp_2006, months, temp_2012)
```

```
plt.legend([2000, 2006, 2012])
plt.title('Average monthly temperature in NYC')
plt.xlabel('Month')
plt.ylabel('Temperature (F)')
```

```
plt.savefig('a.png')Example 9: Adjusting axes rangesimport matplotlib.pyplot as plt
```

```
nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
```

```
plt.plot(nyc_temp, marker='o')
```

```
print(f'Original axes ranges {plt.axis()}') # Display existing axis values (auto-generated)
```

```
newAxisRange = [0, 12, 53.4, 57.3]
plt.axis(newAxisRange)
```

```
print(f'updated axes ranges {plt.axis()}')
```

```
plt.savefig('a.png')Example 10: Adjusting axes ranges version 2import matplotlib.pyplot as plt
```

```
nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
```

```
plt.plot(nyc_temp, marker='o')
```

```
print(f'Original axes {plt.axis()}') # Display existing axis values (auto-generated)
```

```
plt.axis(ymin = 50) # Set minimum of y-axis to zero
plt.axis(ymax = 60) # Set maximum of y-axis
```

```
print(f'updated axes ranges {plt.axis()}')
```

```
plt.savefig('a.png')Example 11: Plot 1 revisitedimport matplotlib.pyplot as plt
```

```
def create_graph():
    x_numbers = [1, 6, 3]
    y_numbers = [2, 4, 6]
```

```
plt.plot(x_numbers, y_numbers)
plt.savefig('a.png')
```

```
if __name__ == '__main__':
    create_graph()Example 12: Graphing functions - Gravityimport matplotlib.pyplot as plt

# Draw the graph - takes the x and y sets of data as parameters
def draw_graph(x, y):
    plt.plot(x, y, marker='o')
    plt.xlabel('Distance (metres)')
    plt.ylabel('Gravitational force (newton...')
```

Lesson 69: Quiz 5

Lesson ID: 56367 | Created: 2024-07-10T20:20:17.576598+10:00 | State: scheduled | Status: unattempted
| Slide Count: 1

Slides

Slide ID: 381784 | Index: 8 | Title: Quiz 5 | Status: unseen

See PDF file below and stub. This quiz is worth 4 marks. Marking stairs_in_grid() 4 marks
----- Total 4 marks Quiz 5 is due Week 8 Thursday 18 July 2024 @ 9.00pm
(Sydney time). Please note that late submission with 5% penalty per day is allowed up to 3 days from the
due date, that is, any late submission after Week 8 Sunday 21 July 2024 @ 9.00pm will be
discarded. Please make sure not to change the filename quiz_5.py while submitting by clicking on [Mark]
button in Ed. It is your responsibility to check that your submission did go through properly using
Submissions link in Ed otherwise your mark will be zero for Quiz 5.

Lesson 70: Week 9 - Notes 14 Levenshtein Distance

Lesson ID: 53329 | Created: 2024-05-27T01:25:43.678313+10:00 | State: active | Status: unattempted | Slide Count: 7

Slides

Slide ID: 362064 | Index: 1 | Title: Week 9 - Levenshtein distance | Status: unseen

Slide ID: 392418 | Index: 2 | Title: depart_leopard | Status: unseen

Slide ID: 392417 | Index: 3 | Title: paper_pope | Status: unseen

Slide ID: 392419 | Index: 4 | Title: PAPER_to_POPE | Status: unseen

Slide ID: 362065 | Index: 5 | Title: Useful links about Levenshtein distance | Status: unseen

Levenshtein distancehttps://en.wikipedia.org/wiki/Levenshtein_distanceLevenshtein Distance, in Three Flavors <https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm>Another Python version of Levenshtein distance https://folk.idi.ntnu.no/mlh/hetland_org/coding/python/levenshtein.pyLevenshtein Distancehttps://python-course.eu/levenshtein_distance.php

Slide ID: 362066 | Index: 6 | Title: Week 9 - Levenshtein distance | Status: unseen

Slide ID: 362067 | Index: 7 | Title: levenshtein_distance.py | Status: unseen

Lesson 71: Quiz 6

Lesson ID: 56994 | Created: 2024-07-12T20:21:44.502539+10:00 | State: scheduled | Status: unattempted
| Slide Count: 1

Slides

Slide ID: 384766 | Index: 8 | Title: Quiz 6 | Status: unseen

See PDF file below and stub. This quiz is worth 4 marks. Marking size_of_largest_parallelogram() 4 marks
----- Total 4 marks Quiz 6 is due Week 9 Thursday 25 July
2024 @ 9.00pm (Sydney time). Please note that late submission with 5% penalty per day is allowed up to 3
days from the due date, that is, any late submission after Week 9 Sunday 28 July 2024 @ 9.00pm will be
discarded. Please make sure not to change the filename quiz_6.py while submitting by clicking on [Mark]
button in Ed. It is your responsibility to check that your submission did go through properly using
Submissions link in Ed otherwise your mark will be zero for Quiz 6.

Lesson 72: Week 10 - Notes 16 Three Special Perfect Squares

Lesson ID: 53332 | Created: 2024-05-27T01:25:44.263266+10:00 | State: active | Status: unattempted | Slide Count: 6

Slides

Slide ID: 362075 | Index: 1 | Title: Three special perfect squares | Status: unseen

Slide ID: 362076 | Index: 2 | Title: Week 10 - Three special perfect squares | Status: unseen

Slide ID: 362077 | Index: 3 | Title: three_special_perfect_squares_v1.py | Status: unseen

Slide ID: 362078 | Index: 4 | Title: three_special_perfect_squares_v2.py | Status: unseen

Slide ID: 362079 | Index: 5 | Title: three_special_perfect_squares_v3.py | Status: unseen

Slide ID: 362080 | Index: 6 | Title: Three special perfect squares | Status: unseen

Lesson 73: Assignment 1

Lesson ID: 54712 | **Created:** 2024-06-11T14:14:43.361316+10:00 | **First Viewed:** 2025-08-05T10:15:46.555125+10:00 | **Last Viewed Slide ID:** 371449 | **State:** scheduled | **Status:** unattempted | **Slide Count:** 1

Slides

Slide ID: 371449 | **Index:** 1 | **Title:** Assignment 1 | **Status:** seen

See PDF file below and stubs. This Assignment 1 is worth 13 marks. Marking

Rectangles Boundary 5 marks

Moving Die 4 marks

Fishing Towns 4 marks

Total 13 marks

Assignment 1 is due Week 7 Monday 8 July 2024 @ 10:00am (Sydney time) Please note that late submission with 5% penalty per day is allowed up to 5 days from the due date, that is, any late submission after Week 7 Saturday 13 July 2024 @ 10:00am will be discarded. Please make sure not to change the filenames boundary.py, moving_die.py, and fishing_towns.py while submitting by clicking on [Mark] button in Ed. It is your responsibility to check that your submission did go through properly using Submissions link in Ed otherwise your mark will be zero for Assignment 1.

Lesson 74: Week 8 - Overview

Lesson ID: 53324 | Created: 2024-05-27T01:25:42.227261+10:00 | State: active | Status: unattempted | Slide Count: 2

Slides

Slide ID: 374417 | Index: 11 | Title: Week 8 Tuesday To Do List | Status: unseen

Week 8 Tuesday To Do List
Admin/Tips
Final Exam scheduled Wednesday 21 August 2024. The exact time slot (morning or afternoon) and location will be known closer to the exam date.
Quiz 5 worth 4 marks is due
Week 8 Thursday @ 9pm
Assignment 2 worth 13 marks is due
Week 11 Monday @ 10am
Content
Week 8 Notes 10 K-means clustering
namedtuple from collections
math.hypot() subplot() from matplotlib.pyplot module
Week 8 Notes 11 The Tower and Marbles Puzzle
The tower and marbles puzzle - n levels and m marbles
Particular Case $m = 2$
random.randint() function
dict.fromkeys() method
General case
List comprehension
Nicely display a rectangle - string formatting
Discuss Assignment 2
Briefly describe second example
Discuss Quiz 5
Note that what is recorded in the grid are not only 0s and 1s. Values up to dim are recorded.
Useful Links
math.hypot()
Method
https://www.w3schools.com/python/ref_math_hypot.asp
subplot() from matplotlib.pyplot module
https://www.w3schools.com/python/matplotlib_subplot.asp

Slide ID: 362041 | Index: 12 | Title: Week 8 Thursday To Do List | Status: unseen

Week 8 Thursday To Do List
Admin/Tips
Quiz 6 worth 4 marks will be released today @ 7.15pm
Quiz 5 worth 4 marks is due today @ 9pm
Assignment 2 worth 13 marks is due
Week 11 Monday 5/8/24 @ 10am
Content
Extra notes that will not be discussed released (see bottom of Ed Lessons)
Week 8 Notes 12
The Game of Life
numpy module
Comprehensive List vs Non-Comprehensive List
Solve Sample Exam Question 1
Discuss Assignment 2
Discuss Quiz 6
Useful Links
Using 2D arrays/lists the right way in Python
<https://www.geeksforgeeks.org/python-using-2d-arrays-lists-the-right-way/NumPy>
Tutorial
<https://www.w3schools.com/python/numpy/default.asp>

Lesson 75: Week 8 - Notes 11 The Tower and Marbles Puzzle

Lesson ID: 53325 | **Created:** 2024-05-27T01:25:42.264296+10:00 | **First Viewed:** 2025-09-18T14:41:11.567836+10:00 | **Last Viewed Slide ID:** 362043 | **State:** active | **Status:** attempted | **Slide Count:** 4

Slides

Slide ID: 362043 | **Index:** 2 | **Title:** The tower and marbles puzzle | **Status:** completed

Slide ID: 362044 | **Index:** 3 | **Title:** The tower and marbles puzzle | **Status:** unseen

Slide ID: 362045 | **Index:** 4 | **Title:** tower_and_2_marbles.py | **Status:** unseen

Slide ID: 362046 | **Index:** 5 | **Title:** tower_and_m_marbles.py | **Status:** unseen

Lesson 76: Quiz 2

Lesson ID: 54805 | Created: 2024-06-13T13:23:17.039281+10:00 | State: scheduled | Status: unattempted
| Slide Count: 1

Slides

Slide ID: 371850 | Index: 5 | Title: Quiz 2 | Status: unseen

See PDF file below and stub. This quiz is worth 4 marks. Marking remove_values_no_greater_than_index()
1.25 marks cap_sum_to() 1.25 marks increasing_sequence_from() 1.50 marks

----- Total 4.00 marks Quiz 2 is due Week 4

Thursday 20 June 2024 @ 9.00pm (Sydney time). Please note that late submission with 5% penalty per day is allowed up to 3 days from the due date, that is, any late submission after Week 4 Sunday 23 June 2024 @ 9pm will be discarded. Please make sure not to change the filename quiz_2.py while submitting by clicking on [Mark] button in Ed. It is your responsibility to check that your submission did go through properly using Submissions link in Ed otherwise your mark will be zero for Quiz 2.

Lesson 77: Quiz 1

Lesson ID: 54434 | Created: 2024-06-05T18:03:53.870343+10:00 | State: scheduled | Status: unattempted
| Slide Count: 1

Slides

Slide ID: 369530 | Index: 1 | Title: Quiz 1 | Status: unseen

See PDF file below and stub. This quiz is worth 4 marks. Marking Absolute Difference 1.0 mark Horizontal Bars 1.5 marks Vertical Bars 1.5 marks ----- Total 4.0 marks Quiz 1 is due Week 3 Thursday 13 June 2024 @ 9.00pm (Sydney time). Please note that late submission with 5% penalty per day is allowed up to 3 days from the due date, that is, any late submission after Week 3 Sunday 16 June 2024 @ 9.00pm will be discarded. Please make sure not to change the filename quiz_1.py while submitting by clicking on [Mark] button in Ed. It is your responsibility to check that your submission did go through properly using Submissions link in Ed otherwise your mark will be zero for Quiz 1.