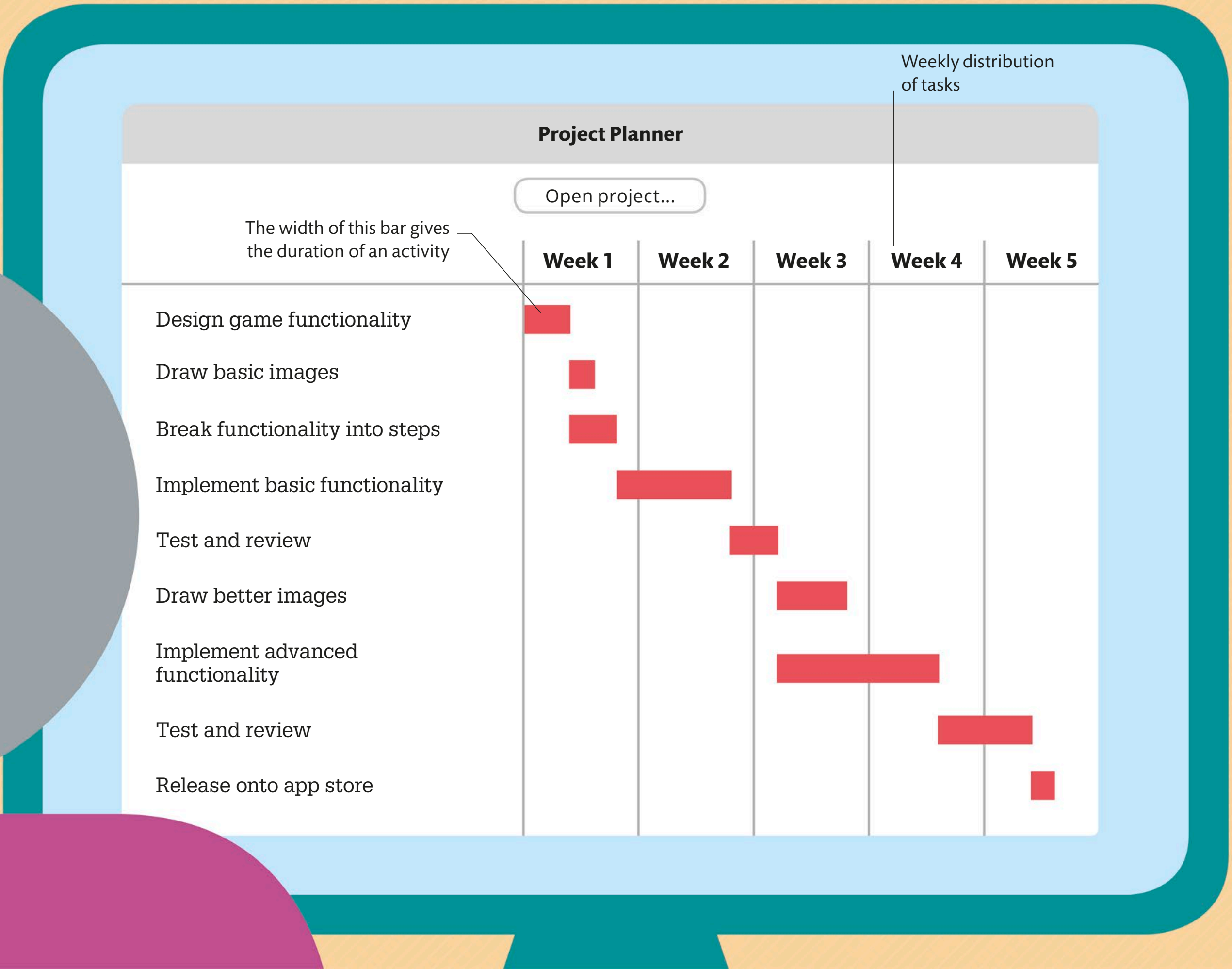
Project planner

Time management tools can be very useful, both at home and at work. There are several applications that help in tracking the progress of daily chores and activities. This project will use Python's tuples, sets, and graphical modules to create a planner for developing a small gaming app.

How it works

This planner will create a schedule to help users plan their work. The program will display a window with a button that a user can press to choose a project file. It will then read a list of tasks from the file and sort them in the order of their starting time based on certain prerequisites. The resulting data will be converted into a chart that will display when each task starts and ends.

Gantt chart
A Gantt chart is a type of bar chart that is used to illustrate the schedule of a project. The tasks to be performed are listed on the y axis and the time periods are listed on the x axis. Horizontal bars on the graph display the duration of each activity.



Weekly distribution of tasks

The width of this bar gives the duration of an activity

YOU WILL LEARN

- How to extract data from a file
- How to use Python sets
- How to use **namedtuples**
- How to create a simple **Tk UI** app
- How to draw using **Tk Canvas**

Time: 1.5 hours

Lines of code: 76

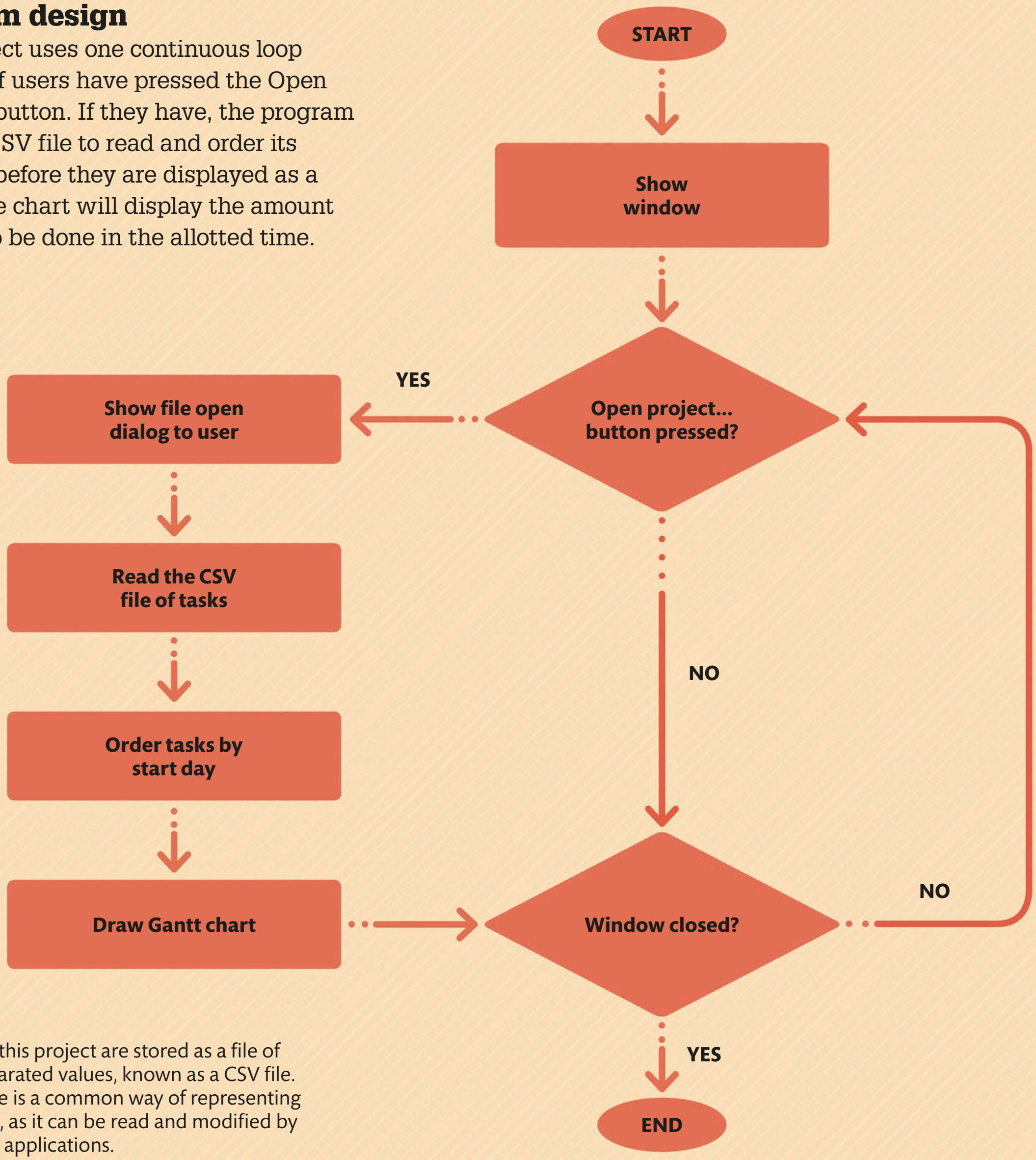
Difficulty level ● ● ● ○ ○

WHERE THIS IS USED

Reading data from files and processing it is common to almost all programs, even the ones that do not use documents in an obvious manner (for example, games). The basic tasks of opening windows, laying out buttons, and drawing custom elements are the building blocks of any desktop application.

Program design

This project uses one continuous loop to check if users have pressed the Open project... button. If they have, the program opens a CSV file to read and order its contents before they are displayed as a chart. The chart will display the amount of work to be done in the allotted time.

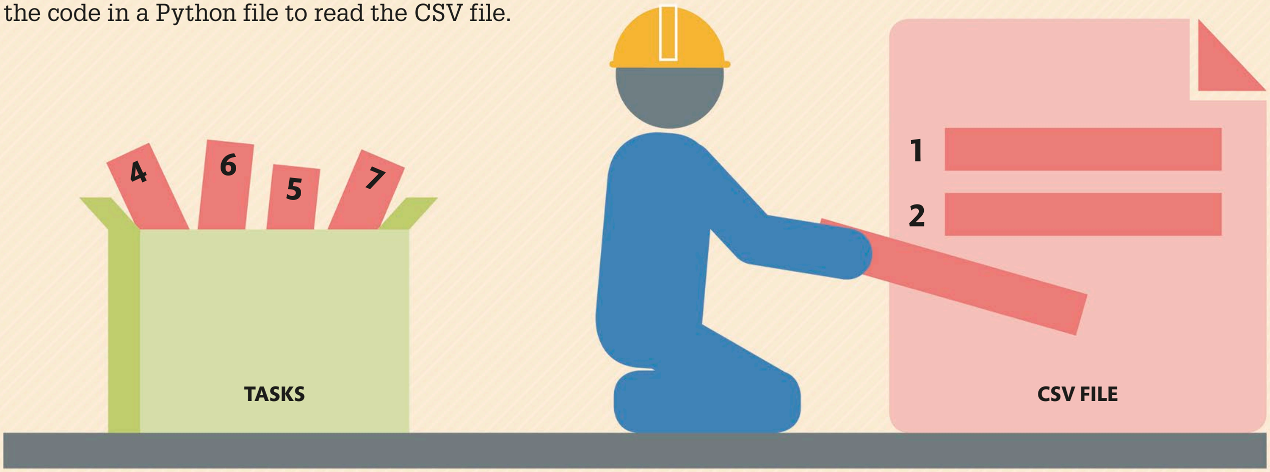

START
Show window
Open project... button pressed?
YES
Show file open dialog to user
Read the CSV file of tasks
Order tasks by start day
Draw Gantt chart
NO
Window closed?
NO
YES
END

CSV file

The tasks in this project are stored as a file of comma-separated values, known as a CSV file. Using this file is a common way of representing tabular data, as it can be read and modified by spreadsheet applications.

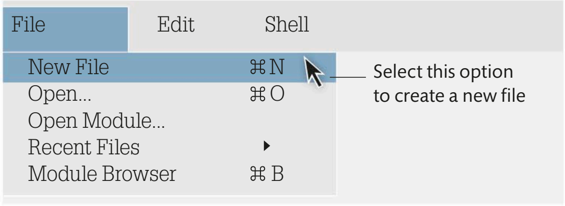
1 Creating and reading the CSV file

To draw the planner in your app, you need to create a CSV file that lists all of the tasks that have to be completed. Next, you will write the code in a Python file to read the CSV file.



1.1 CREATE A NEW FILE

The first step is to create a new file for the Python code. Create a folder called "ProjectPlanner" on your computer. Next, open IDLE and select New File from the File menu. Choose Save As from the same menu and save the file as "planner.py" inside the ProjectPlanner folder.



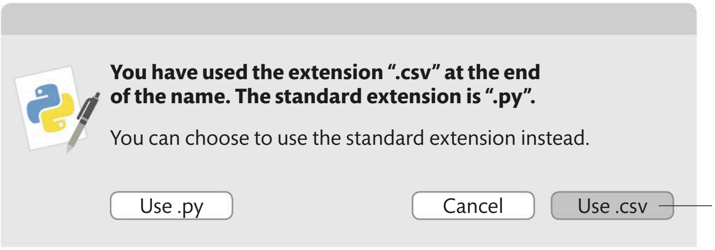
Select this option to create a new file

1.2 CREATE A CSV FILE

Python has a library called **csv** that makes it easy to read and write CSV files. Now add a line of code at the top of your Python file to read the new CSV file. However, before you can read a CSV file, you will need to create one. This can be done with a spreadsheet application, but because a CSV file is a simple text file, you can create it in IDLE. Select New File from the File menu, choose Save As, and save the file in the ProjectPlanner folder. Name this file "project.csv". You may get a warning message when you do this, as ".csv" is not a standard Python extension, but you should use it anyway.

```
import csv
```

Type this line in the "planner.py" file

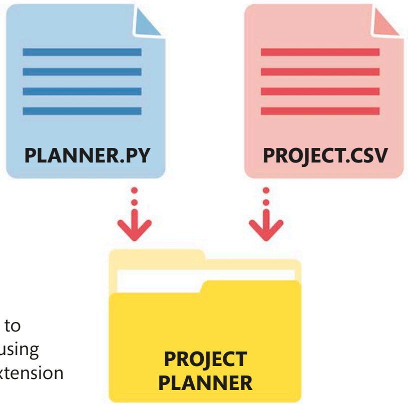


You have used the extension ".csv" at the end of the name. The standard extension is ".py".

You can choose to use the standard extension instead.

Click here to continue using the .csv extension

WARNING MESSAGE



1.3 WRITE A SIMPLE PROJECT

Now you can write a simple plan for a project to develop a small gaming app. Type the following lines into the CSV file with a list of tasks to be completed to create the gaming app. There should be no blank lines at the beginning or end of the file. Each line of text in the file will represent one row of the table and each element in the row will represent one column value. For example, the second row has four column values. Save and close the file once you have typed in the tasks correctly.

```
1,Design game functionality,2,
2,Draw basic images,1,1
3,Break functionality into steps,2,1
4,Implement basic functionality,5,2 3
5,Test and review,2,4
6,Draw better images,3,5
7,Implement advanced functionality,7,5
8,Test and review,4,6 7
9,Release onto app store,1,8
```

The first column value represents the task number

The second column value gives a title to the task

The third column value gives the number of days the task is expected to take

The values in each column are separated by commas

Each line represents one row of the table

The fourth column value gives the prerequisites of the task as task numbers with spaces in between

This row is task **8** with the title **`Test and review`**. It is expected to finish in **4** days and requires tasks **6** and **7** to be completed before it can start

PYTHON TUPLE

A tuple is a data structure like a list, but its length cannot be changed after it has been created and the items inside it cannot be updated. Lists are mostly used to store a sequence of values of the same kind, such as a list of numbers representing the height of a group of people. Tuples, on the other hand, are used when the values are related but of different kinds, such as one person's name, age, and height.

```
>>> numbers = (1, 2, 3, 4, 5)
>>> print(numbers[3])
4
```

`numbers` is a tuple with five values

The value at index position 3 in the tuple

Index numbers are enclosed within square brackets

```
>>> numbers[0] = 4
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    numbers[0] = 4
TypeError: 'tuple' object does not support item assignment
```

Try changing the value at index position 0 in the tuple

Returns an error because the values inside a tuple cannot be updated

1.4 READ DATA FROM THE FILE

The functionality in Python's **csv** library makes it easy to read data from the CSV file. Once the data is read, the values are stored in a Python tuple. The tuple is then stored into a "dictionary" (a data structure where each item has two parts—a key and a value), using the task number (the value from the first column) as the key (see p.160). This will allow you to look up a particular task quickly by its number. Now add this code to your .py file after the import statement from step 1.2. It will open the CSV file, read the rows of data from it, and place the results into a dictionary.

```
def read_tasks(filename):
    tasks = {}
    for row in csv.reader(open(filename)):
        number = row[0]
        title = row[1]
        duration = row[2]
        prerequisites = row[3]
        tasks[number] = (title, duration, \
                         prerequisites)
    return tasks
```

The name of the file is given as the argument to this function

Sets **tasks** to an empty dictionary

Opens the file for reading, uses a **reader** object from the **csv** library to interpret the file as CSV data, and then iterates over each row with a **for** loop

Extracts the four values from the row. The row is indexed by a column number (counting from 0) to obtain a particular value

The function returns the complete dictionary

The values are stored as a tuple in the **tasks** dictionary by task number

SAVE

1.5 TEST THE CODE

Now test the code to make sure you have typed in the instructions correctly. Choose Run Module from the Run menu and switch to the shell window. Type the code below to call the function with the name of the CSV file you created in step 1.2. The function will return a dictionary containing the information from the file. However, all of the values will be read as Python strings, as the **csv.reader** object does not know how to interpret the data that it is reading from a file.

Type this line at the prompt

Reads the data in this CSV file

```
>>> read_tasks("project.csv")
{'1': ('Design game functionality', '2', ''), '2': ('Draw
basic images', '1', '1'), '3': ('Break functionality into
steps', '2', '1'), '4': ('Implement basic functionality',
'5', '2 3'), '5': ('Test and review', '2', '4'), '6': ('Draw
better images', '3', '5'), '7': ('Implement advanced
functionality', '7', '5'), '8': ('Test and review', '4',
'6 7'), '9': ('Release onto app store', '1', '8')}
```

Numbers are also read as strings

1.6 CONVERT TO OTHER DATA TYPES

The "task number" and "task duration" values are numbers in the CSV file. Because these are currently read as strings, it will be better if they can be converted into Python number values instead. Update the **read_tasks()** function as shown below. The task number will always be an integer (whole) number, but the task duration will be a float (decimal) value, as it can take a nonwhole number (like 2.5) of days to finish a task. Save the file and then run the module again to test this.

```
def read_tasks(filename):
    tasks = {}
    for row in csv.reader(open(filename)):
        number = int(row[0])
        title = row[1]
        duration = float(row[2])
        prerequisites = row[3]
```

Converts the task number from a string into an integer number

Converts the task duration from a string into a floating point number

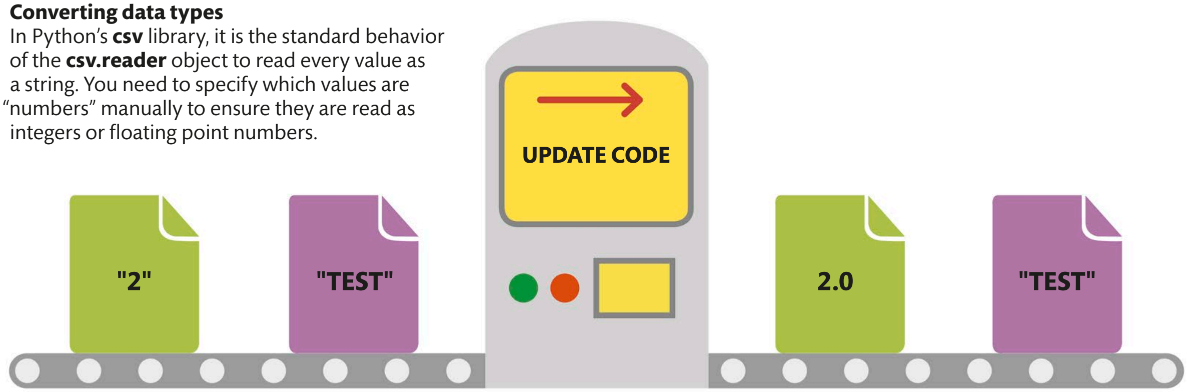
SAVE

```
>>> read_tasks("project.csv")
{1: ('Design game functionality', 2.0, ''), 2: ('Draw basic
images', 1.0, '1'), 3: ('Break functionality into steps', 2.0,
'1'), 4: ('Implement basic functionality', 5.0, '2 3'), 5:
('Test and review', 2.0, '4'), 6: ('Draw better images', 3.0,
'5'), 7: ('Implement advanced functionality', 7.0, '5'), 8:
('Test and review', 4.0, '6 7'), 9: ('Release onto app store',
1.0, '8')}
```

Task numbers are read as integer values

Task duration is read as a floating point value

Converting data types
In Python's **csv** library, it is the standard behavior of the **csv.reader** object to read every value as a string. You need to specify which values are "numbers" manually to ensure they are read as integers or floating point numbers.



PYTHON SETS

A Python set is another data type that is similar to a list, but it can only contain unique values. This makes it similar to the keys of a dictionary. The syntax for writing a set is similar to that of a dictionary. A set can be assigned to a variable in several ways. Try these examples in the shell window.

Just like a dictionary, Python sets are also written inside curly brackets

```
>>> numbers = {1, 2, 3}
```

Defining a set
The variable **numbers** is defined as a set containing the numbers 1, 2, and 3. You should never write an empty set as "numbers = {}", as Python will read it as an empty dictionary. To avoid this, create an empty set by calling the **set()** constructor function.

Adds the number "4" to the set

```
>>> numbers.add(4)
>>> numbers
{1, 2, 3, 4}
>>> numbers.add(3)
>>> numbers
{1, 2, 3, 4}
```

The number "3" is already in the set, so the value inside it does not change

Adding values to a set
You can add values to a set with the **add** method. Because a set only contains unique values, adding a value that is already in the set will do nothing.

Removes the value "3" from the set

```
>>> numbers.remove(3)
>>> numbers
{1, 2, 4}
```

Removing values from a set
Similarly, you can also remove items from a set using the **remove** method.

1.7 PREREQUISITES AS SETS OF NUMBERS

So far, you have converted the task number and task duration into integers, but the prerequisites are still a string ("1" or "2 3"). To read the prerequisites as a collection of task numbers, first split the string into individual values using Python's built-in **split** method. Next, use the **int()** and **map()** functions, as shown here, to turn the string values into a set.

```
>>> value = "2 3"
>>> value.split()
['2', '3']
```

Items separated by spaces will be split into separate values

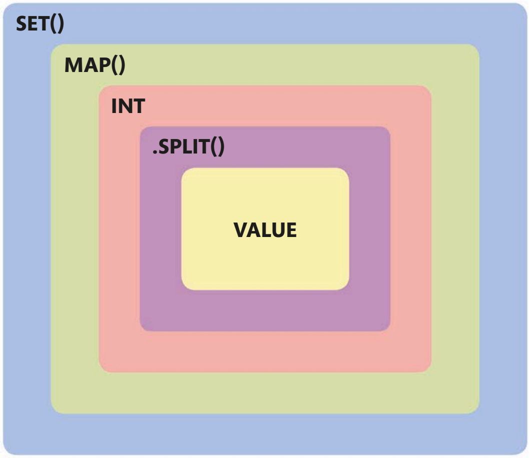
The **split** method takes one string and turns it into a list of strings

map() calls the **int()** function on every string in the list

```
>>> set(map(int, value.split()))
{2, 3}
```

int() converts a string value into an integer

Converts the values returned by **map()** into a (set) data structure



Combining functions
This illustration demonstrates how to combine simple functions to create complex logic. It starts with the original string value and splits it into string parts. The **int()** function is then called on each of these parts using the **map()** function. **set()** turns the result into a Python set.

1.8 MAKE THE PREREQUISITE CHANGES

Now incorporate the code from the previous step into the `read_tasks()` function as shown below. Run the module again and switch to the shell window to test it.

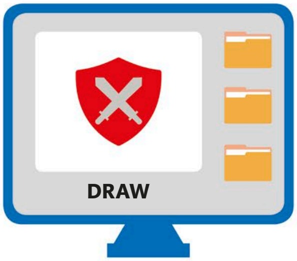
Converts the prerequisite values from strings into sets of integers

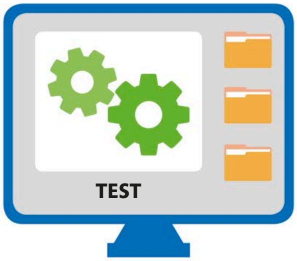
```
import csv
def read_tasks(filename):
    tasks = {}
    for row in csv.reader(open(filename)):
        number = int(row[0])
        title = row[1]
        duration = float(row[2])
        prerequisites = set(map(int, row[3].split()))
        tasks[number] = (title, duration, prerequisites)
    return tasks
```

SAVE

```
>>> read_tasks("project.csv")
{1: ('Design game functionality', 2.0, set()),
2: ('Draw basic images', 1.0, {1}), 3: ('Break
functionality into steps', 2.0, {1}), 4:
('Implement basic functionality', 5.0, {2, 3}),
5: ('Test and review', 2.0, {4}), 6: ('Draw
better images', 3.0, {5}), 7: ('Implement
advanced functionality', 7.0, {5}), 8: ('Test
and review', 4.0, {6, 7}), 9: ('Release onto app
store', 1.0, {8})}
```

All numeric values are now converted into the correct data type





1.9 TEST THE PROGRAM

The data is now ready, and you can try to pull out some specific bits to test it. Run the module again and switch to the shell window. Type the lines of code shown below. This time, you will store the resulting dictionary in a temporary variable so that it can be manipulated.

```
>>> tasks = read_tasks("project.csv")
```

Assigns the data to a temporary variable **tasks**

```
>>> tasks[3]

('Break functionality into steps', 2.0, {1})
```

Pulls out the data for a specific task by indexing **tasks** with the task number

Data is returned as a tuple of three values with specific index positions

Title is at index position 0

Duration is at index position 1

Prerequisites are at index position 2

```
>>> tasks[3][1]

2.0
```

Extracts the duration of this task by indexing again with the value **1**

Returns the value (duration) at index position **[1]** in task number **[3]**

1.10 USE NAMED TUPLES

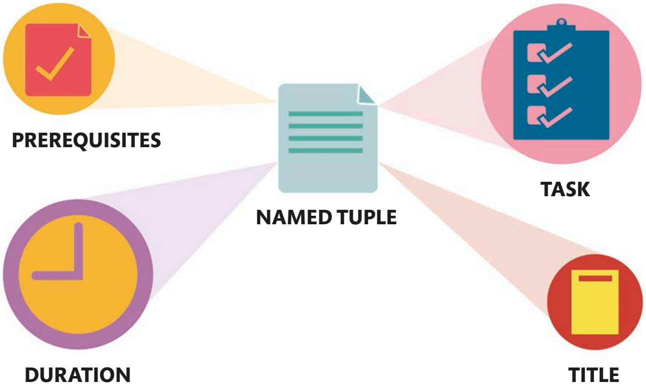
Getting task values by their index positions is not an ideal way to extract them. It will be better if they can be referred to by a proper name, such as "title" or "duration". Python provides a standard way of doing this. You can create named tuples that will allow you to extract the values within them by name instead of position. Add this code at the top of your file to create a named tuple type and store it in a variable.

```
import csv

from collections import namedtuple

Task = namedtuple("Task", ["title", "duration", "prerequisites"])

def read_tasks(filename):
```

Imports the **namedtuple()** function from the **collections** module

Defines a named tuple called **Task**

The value names are given as a list of strings



1.11 CALL THE NAMED TUPLE TYPE

The named tuple type created in the previous step is stored in the variable **Task**. You can create new values of this type by calling **Task** like a function. Update the **read_task()** function in your code to call **Task** instead of creating a tuple in the normal way. Run the module and switch to the shell window to test the code. First, you will display the values in the shell (output 1), then you will try to extract one of these values by using its name (output 2).

The named tuple **Task** is stored in the **tasks** dictionary

```
def read_tasks(filename):
    tasks = {}
    for row in csv.reader(open(filename)):
        number = int(row[0])
        title = row[1]
        duration = float(row[2])
        prerequisites = set(map(int, row[3].split()))
        tasks[number] = Task(title, duration, prerequisites)
    return tasks
```

SAVE

```
>>> tasks = read_tasks("project.csv")
>>> tasks[3]
Task(title="Break functionality into steps", duration=2.0,
prerequisites={1})
```

Names are displayed in the shell for each of the values in the named tuple

OUTPUT 1

```
>>> tasks[1].title
"Design game functionality"
>>> tasks[3].duration
2.0
>>> tasks[4].prerequisites
{2, 3}
```

Extracts the title of **task[1]** by name

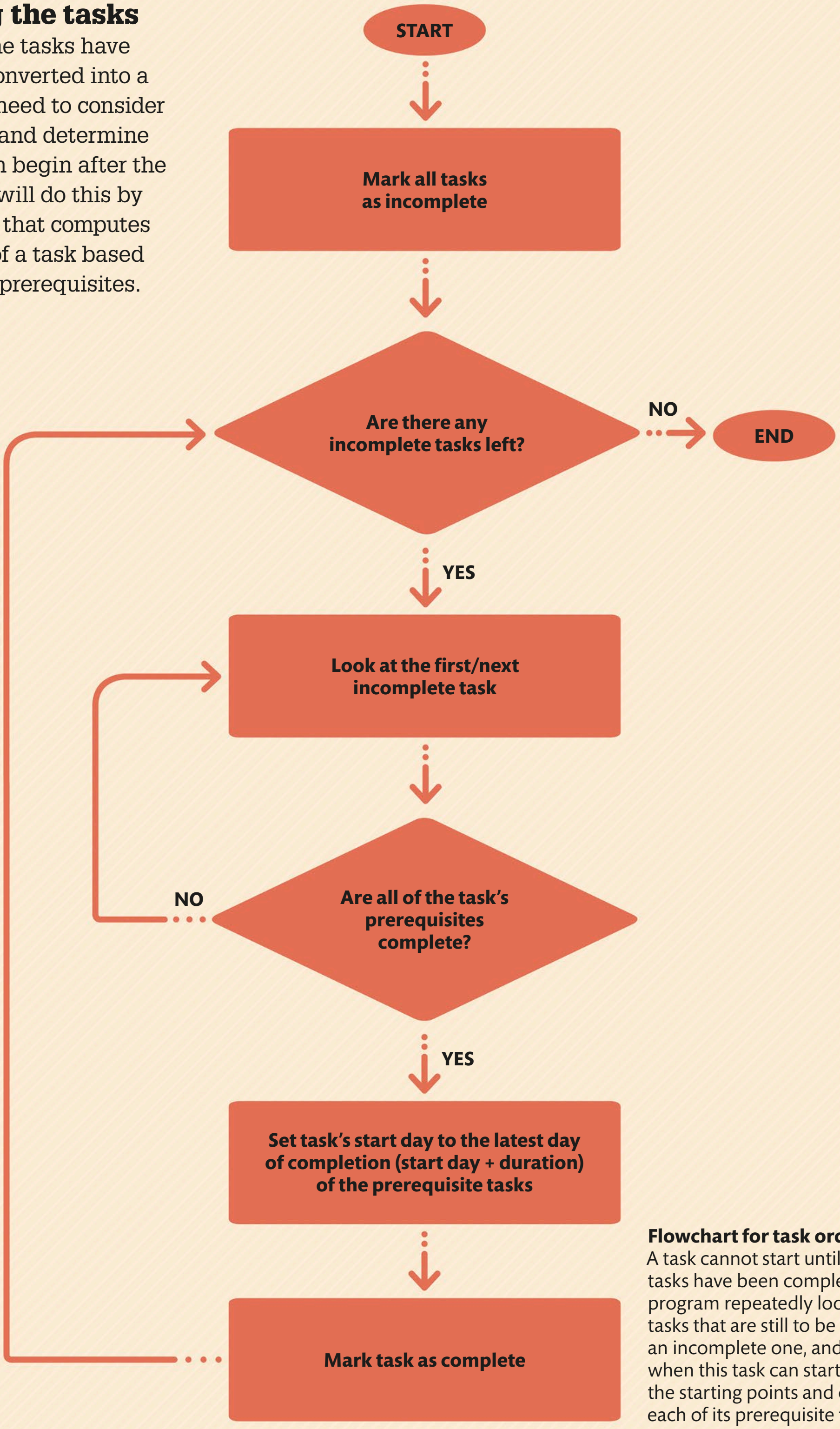
Extracts the duration of **task[3]** by name

OUTPUT 2

Extracts the prerequisites of **task[4]** by name

2 Ordering the tasks

Now that the tasks have been read in and converted into a useful format, you need to consider how to order them and determine when each task can begin after the project starts. You will do this by creating a function that computes the starting point of a task based on the status of its prerequisites.


START
Mark all tasks as incomplete
Are there any incomplete tasks left?
NO
END
YES
Look at the first/next incomplete task
Are all of the task's prerequisites complete?
NO
YES
Set task's start day to the latest day of completion (start day + duration) of the prerequisite tasks
Mark task as complete

Flowchart for task ordering logic
A task cannot start until its prerequisite tasks have been completed. The program repeatedly loops over all of the tasks that are still to be completed, picks an incomplete one, and then calculates when this task can start by computing the starting points and durations of each of its prerequisite tasks.

2.1 IMPLEMENT THE LOGIC

You can now implement the logic for ordering the tasks. Add the following function at the end of the file. This will return a dictionary that will map each task number to a start day, expressed as a number of days from the start of the entire project. So the first task(s) will begin at 0 days.

```
    return tasks
def order_tasks(tasks):
    incomplete = set(tasks)
    completed = set()
    start_days = {}
    while incomplete:
        for task_number in incomplete:
            task = tasks[task_number]
            if task.prerequisites.issubset(completed):
                earliest_start_day = 0
                for prereq_number in task.prerequisites:
                    prereq_end_day = start_days[prereq_number] + \
                                     tasks[prereq_number].duration
                    if prereq_end_day > earliest_start_day:
                        earliest_start_day = prereq_end_day
                start_days[task_number] = earliest_start_day
                incomplete.remove(task_number)
                completed.add(task_number)
                break
    return start_days
```

Starts with all the tasks incomplete and no start days

Gets the task and checks if its prerequisites have been completed

Loops over the incomplete task numbers while there are still any left

Computes the earliest this task can start based on the end days of its prerequisites

Breaks out of the **for** loop. The loop will start again if there are still some incomplete tasks left

Stores the start date and remembers that this task has been completed

Returns the computed dictionary

ISSUBSET SET METHOD

The **issubset** set method checks whether one set is contained within another set. An empty set is a subset of any set, including another empty set. This means that **task.prerequisites.issubset(completed)** will be true for a task with no prerequisites and will begin immediately, even when no tasks have been completed yet. The **earliest_start_day** is set to 0 before looping over a task's prerequisites. If there are no prerequisites, then this task will use 0 as its start day. Once this task is added to the completed set, it will allow the tasks that depend on it to begin.

2.2 TEST THE CODE

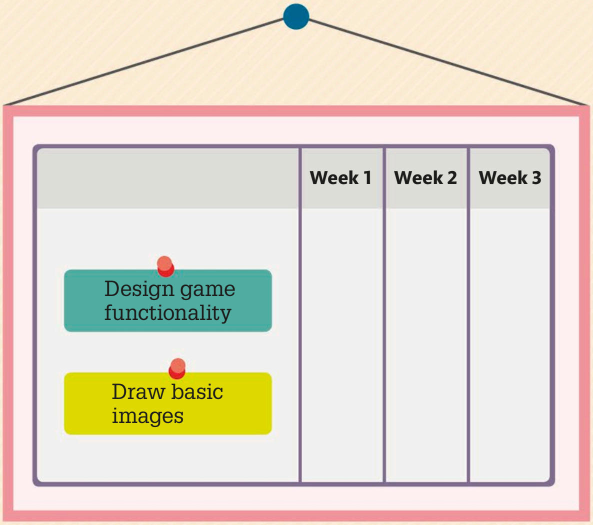
Save the code and run the module to test the `order_tasks()` function at the prompt. You will see that task 1 can begin immediately (after 0 days) and task 9 is the last task to start, 22 days after the project begins. Tasks 2 and 3 will both start at the same time, as will tasks 6 and 7. It is assumed that the user will be able to do both tasks at the same time.

```
>>> tasks = read_tasks("project.csv")
>>> order_tasks(tasks)
{1: 0, 2: 2.0, 3: 2.0, 4: 4.0, 5: 9.0, 6: 11.0, 7: 11.0,
8: 18.0, 9: 22.0}
```

These tasks start at the same time because they have the same prerequisites

3 Drawing the chart

Now that you have read the CSV file and ordered the tasks inside of it, it is time to draw a chart for the project. Python has a built-in cross-platform toolkit for graphical applications called **Tk**. You will use this to open a window and draw inside of it.



3.1 IMPORT THE TOOLKIT

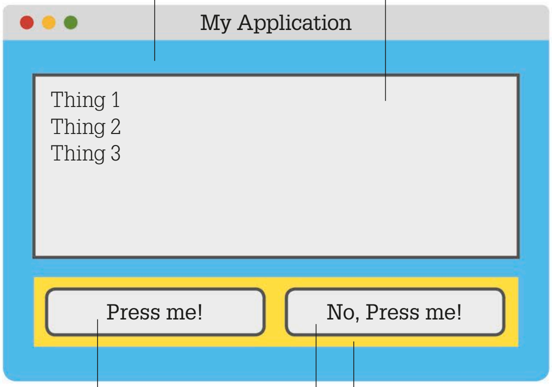
Start by importing the **Tk** functionality into your program. It is found in Python's standard library called **tkinter**—short for Tk Interface. Add this code at the top of the .py file. By convention, the **import** statements are ordered alphabetically at the top of the file, but it does not matter if they are arranged in a different order.

```
import csv
import tkinter
```

Imports the **Tk** functionality

THE Tk GUI

Visual elements in **Tk** are called "widgets." Widgets are placed inside one another to create a hierarchy of graphical elements. The "root" (first) widget created in this hierarchy is the top-level window widget. Widgets are created by calling their **Tk** constructors with the parent widget as the first argument, followed by a set of keyword arguments specifying different attributes of the widget, such as its size and color. Widgets are visually packed within their parent widgets. **Tk** module's `mainloop()` function draws the widgets onscreen and handles events such as mouse clicks and key presses. This function does not return while the window is open. If you want to do anything after the window opens, you will have to define functions that will be called by `mainloop()` when specific events happen, such as a button being pressed.



The root window widget

List widget packed on top of root window

Button widget packed at the left side of frame

Button widget packed at the right side of frame

Frame container widget packed at the bottom of the root window

3.2 CREATE A WINDOW

Next, add this code at the end of the .py file to create a window. It will contain a button and a canvas widget. The button will display some text as a label, and the canvas widget will define an area that you can draw into. You need to specify the size and background color of the canvas widget.

```
    return start_days
root = tkinter.Tk()
root.title("Project Planner")
open_button = tkinter.Button(root, text="Open project...", \
                             command=open_project)
open_button.pack(side="top")
canvas = tkinter.Canvas(root, width=800, \
                        height=400, bg="white")
canvas.pack(side="bottom")
tkinter.mainloop()
```

Creates a **Tk** top-level window widget (see box, above)

Creates a button widget and places it at the top edge of the window

Gives the window a title

Creates a canvas widget and places it at the bottom edge of the window

Runs the **Tk** main event-handling function

SAVE

3.3 RUN THE CODE

If you run the code at this point, you will see a blank white window with no button inside of it. You will also get an error message in the shell window. This is because the **open_project()** function has not been defined yet. You will need to close this window to continue.

```
====== RESTART: /Users/tina/ProjectPlanner/planner.py ======
Traceback (most recent call last):
  File "/Users/tina/ProjectPlanner/planner.py", line 35, in <module>
    open_button = tkinter.Button(root, text="Open project...",
command=open_project)
NameError: name 'open_project' is not defined
>>>
```

The program will crash and display this error in the shell window

3.4 ACTIVATE THE BUTTON

The button you created in step 3.2 should allow you to select a .csv project file that will then be drawn into a chart. To do this, use a **Tk** file dialog found in a submodule of **tkinter**. Add the import statement at the top of your file as shown. Next, add a new **open_project()** function just below the **order_tasks()** function from step 2.1. If you run the program now, you will get another error message, as the **draw_chart()** function has not been defined yet.

```
import tkinter
from tkinter.filedialog import askopenfilename
```

Imports a single function from **tkinter.filedialog** rather than importing the entire module

Calls the function to open a file dialog for choosing a CSV file

Specifies the dialog title

"." is a special directory name for the "current" directory

```
    return start_days
def open_project():
    filename = askopenfilename(title="Open Project", initialdir=".", \
                               filetypes=[("CSV Document", "*.csv")])
    tasks = read_tasks(filename)
    draw_chart(tasks, canvas)
```

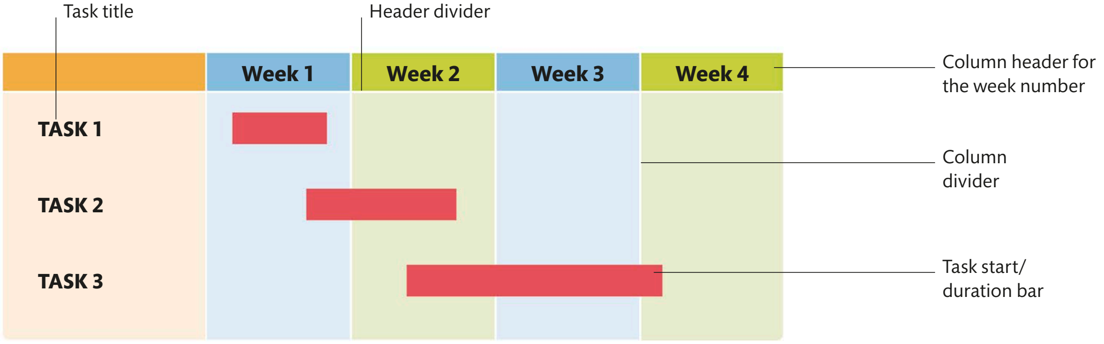
Draws a chart of the tasks in the canvas widget

Reads the tasks from the .csv file returned by the dialog

Specifies the acceptable file format

3.5 DRAW THE CHART

It is time to draw the project as a Gantt chart. Before drawing the chart, you will first need to decide what you want it to look like and what visual elements you need to draw it. Add this code below the code from step 2.1 (above the `open_project()` function) to draw the headers and dividers of the chart. This will define a `draw_chart()` function and gives default values to some of its arguments. Only the first two arguments (`tasks` and `canvas`) are actually required to call the function. The arguments with default values are optional and will take the values that you have specified, creating some local "constants" in the function.



Arguments with default values specify where to draw the elements and how much space they will take on the canvas

Default value of an argument

```
def draw_chart(tasks, canvas, row_height=40, title_width=300, \
               line_height=40, day_width=20, bar_height=20, \
               title_indent=20, font_size=-16):
    height = canvas["height"]
    width = canvas["width"]
    week_width = 5 * day_width
    canvas.create_line(0, row_height, width, line_height, \
                       fill="gray")
    for week_number in range(5):
        x = title_width + week_number * week_width
        canvas.create_line(x, 0, x, height, fill="gray")
        canvas.create_text(x + week_width / 2, row_height / 2, \
                           text=f"Week {week_number+1}", \
                           font=("Helvetica", font_size, "bold"))
def open_project():
```

Defines the height and width of the canvas as local variables

Draws a horizontal line for the header, one row down and across the entire width of the chart

Loops through the number of weeks from 0 to 4

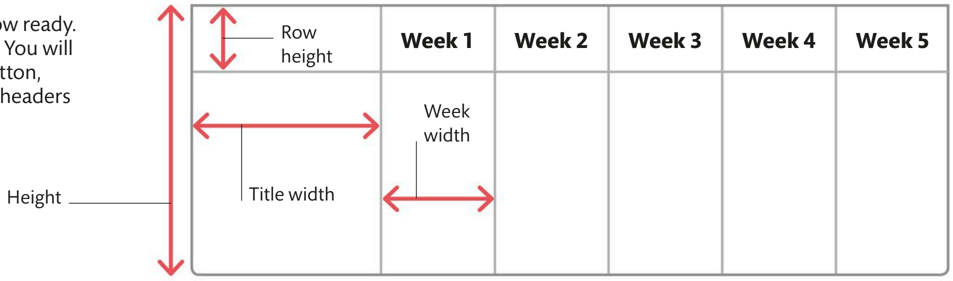
Sets **x** to the width of the title plus the week width times the number of the week

Draws a vertical line at **x** down the entire height of the chart

Draws a text string at a point half a week width past **x** and half a row down

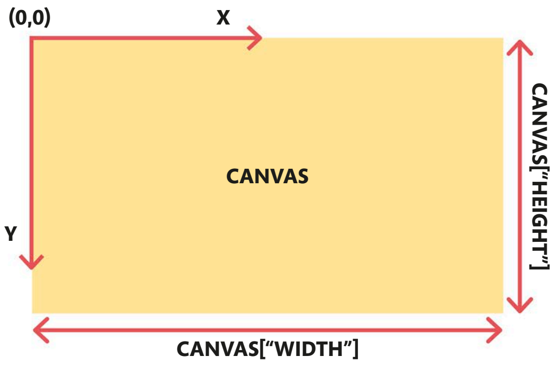
3.6 RUN THE CODE

Your Gantt chart is now ready. Save the file and run the code. You will now be able to click on the button, choose a CSV file, and see the headers and dividers being drawn.



THE Tk CANVAS WIDGET

The Canvas widget provides a space onscreen inside which you can add elements, such as lines, rectangles, and text. You need to call methods on the canvas object to create the elements. These methods take one or more coordinates as arguments, followed by a number of optional keyword arguments that allow the user to specify styling information, such as colors, line thicknesses, or fonts (see tables, below). Canvas coordinates are specified in pixels from the top left corner of the drawing area. Colors can be specified either by their names, such as “red” or “yellow”, or by their hex code, such as “#FF0000”. Text is drawn centered on the given coordinates by default. The anchor keyword argument can be set to a “compass point” constant (**tkinter.N**, **tkinter.NE**, and **tkinter.E**) to draw the text with a corner or edge at the coordinates instead.



BASIC METHODS

Method	Description
create_line(x1, y1, x2, y2, ...)	Adds a line from **(x1, y1)** to **(x2, y2)**
create_rectangle(x1, y1, x2, y2, ...)	Adds a rectangle from **(x1, y1)** to **(x2, y2)**
create_oval(x1, y1, x2, y2, ...)	Adds an oval with a bounding box from **(x1, y1)** to **(x2, y2)**
create_text(x1, y1, text=t, ...)	Adds a text label anchored at **(x1, y1)** showing string **t**

ADDITIONAL STYLING ARGUMENTS

Argument	Description
width	Line width
fill	Fill color of a shape or the color of lines and text
outline	Outline color of shapes
font	Font used for text, either a tuple of (name, size) or (name, size, style)
anchor	Anchor point of the text used when drawing at the specified coordinates

3.7 DRAWING THE TASKS

Finally, add this code to draw the task title and the task duration bar for each task. Type these lines at the end of the **draw_chart()** function. Save the file and run the code to see the complete Gantt chart when you open the "project.csv" file.

```
 ...canvas.create_text(x + week_width / 2, row_height / 2, \
                       text=f"Week {week_number+1}", \
                       font=("Helvetica", font_size, "bold"))
    start_days = order_tasks(tasks)
        y = row_height
        for task_number in start_days:
            task = tasks[task_number]
            canvas.create_text(title_indent, y + row_height / 2, \
                               text=task.title, anchor=tkinter.W, \
                               font=("Helvetica", font_size))
            bar_x = title_width + start_days[task_number] \
                    * day_width
            bar_y = y + (row_height - bar_height) / 2
            bar_width = task.duration * day_width
            canvas.create_rectangle(bar_x, bar_y, bar_x + \
                                    bar_width, bar_y + \
                                    bar_height, fill="red")
            y += row_height
```

Draws the task title anchored at the center left of the text, half a row below **y** and **title_indent** in from the left

Orders the tasks to get the start days

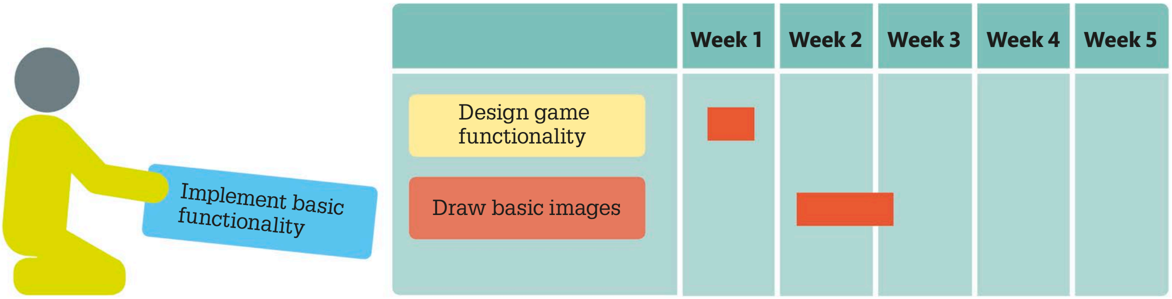
Begins with **y**, one row down from the top of the canvas

Loops over the task numbers in the order that they occur in the **start_days** dictionary

Calculates the coordinates of the top left corner of the bar and its width

Adds a vertical space of **row_height** into the original **y**

Draws a red-colored bar using these values



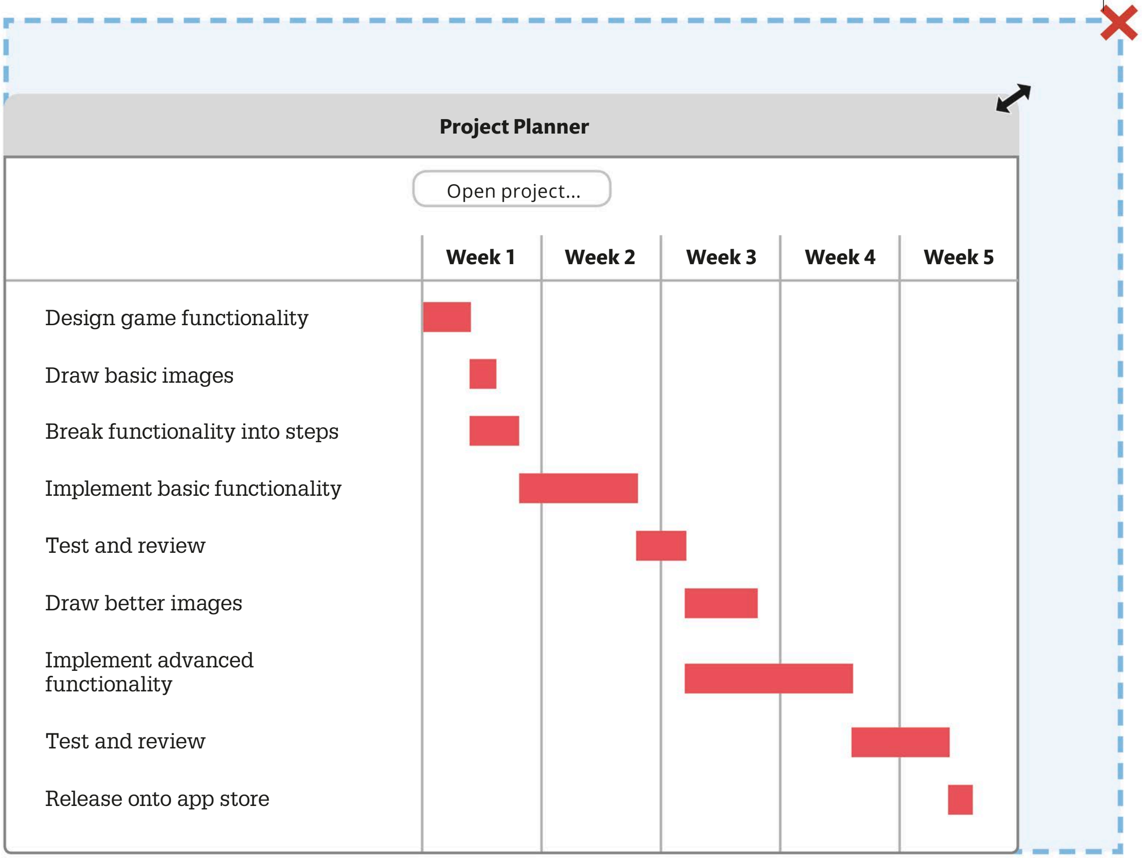
Hacks and tweaks

Stop the window from resizing

At the moment, the user can manually resize the window of the Gantt chart. However, this causes the contents to move around or to be cut off. Drawing the window properly when it is resized is quite complicated, but you can stop it from being resized instead. Add this line of code to the program to make this change.

```
root.title("Project Planner")
root.resizable(width=False, height=False)
open_button = tkinter.Button(root, text="Open project...", \
                             command=open_project)
```

Prevents the root window from resizing in any direction



You will not be able to resize the window anymore

Use a frame to layout the button

You can use a **Tk Frame** widget to change the position of the Open Project... button. Currently it is stuck in the middle of the window at the top. You can place it in the top left corner and add a bit of space around it. Add the following lines of code at the bottom of the .py file to create the **`button_frame`** and then update the **`open_button`** so it sits inside this widget.

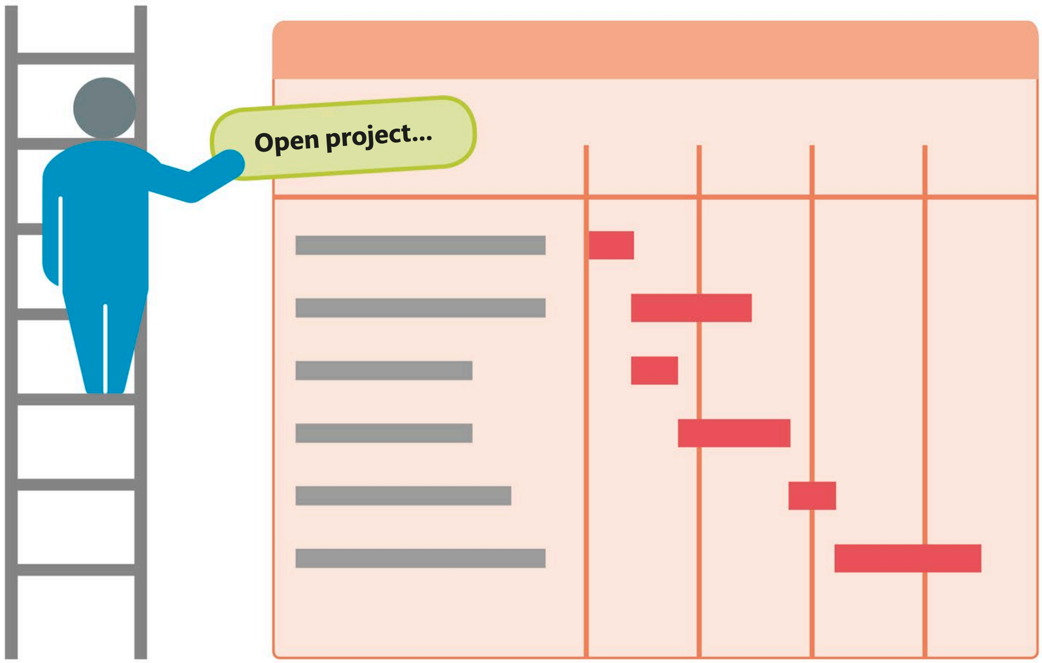
```
root = tkinter.Tk()
root.title("Project Planner")
root.resizable(width=False, height=False)
button_frame = tkinter.Frame(root, padx=5, pady=5)
button_frame.pack(side="top", fill="x")
open_button = tkinter.Button(button_frame, text="Open project...", \
                             command=open_project)
open_button.pack(side="left")
canvas = tkinter.Canvas(root, width=800, height=400, bg="white")
```

Creates a frame at the root of the window, with a small amount of **x** and **y** padding

Places the frame at the top of the window, filling the entire width of **x**

Places the button at the left of the frame

Creates the **`open_button`** inside the **`button_frame`** instead of the root



Add a filename label

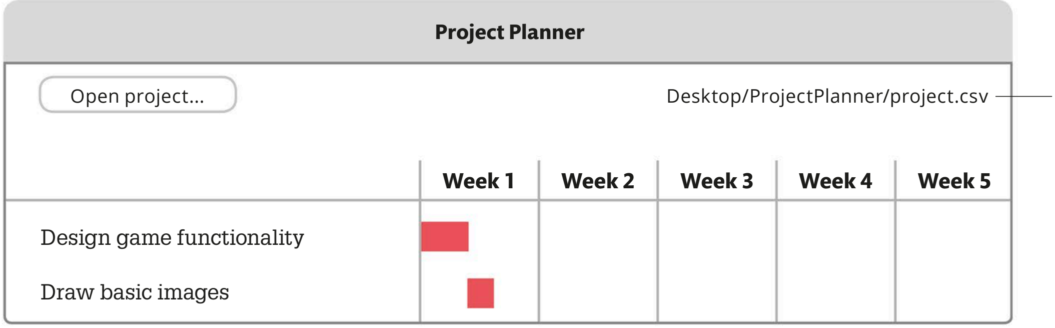
You can also place a label inside the window with the name of the file that you are looking at. Add the following lines to the code as shown. The **config** method used in the code will allow you to reconfigure a widget after it has been created. It takes the same named keyword as the original widget-creation function. This will allow you to specify the text attribute of the **Label** widget after you have opened the file.

```
def open_project():
    filename = askopenfilename(title="Open Project", initialdir=".", \
                           filetypes=[("CSV Document","*.csv")])
    tasks = read_tasks(filename)
    draw_chart(tasks, canvas)
    filename_label.config(text=filename)
root = tkinter.Tk()
root.title("Project Planner")
root.resizable(width=False, height=False)
button_frame = tkinter.Frame(root, padx=5, pady=5)
button_frame.pack(side=”top”, fill=”x”)
open_button = tkinter.Button(button_frame, text="Open project...", \
                             command=open_project)
open_button.pack(side="left")
filename_label = tkinter.Label(button_frame)
filename_label.pack(side="right")
canvas = tkinter.Canvas(root, width=800, height=400, bg="white")
```

Updates the text attribute of the label with the name of the file

Creates a new label inside **button_frame**

Places the label to the right of the frame



Project Planner

Open project...

Desktop/ProjectPlanner/project.csv

Week 1 Week 2 Week 3 Week 4 Week 5

Design game functionality

Draw basic images

The name of the file will appear to the right of the Open project... button

Add a clear button

You can also add another button to your program that will clear all of the items from the window and erase the chart. Add the following lines to the code to create it.

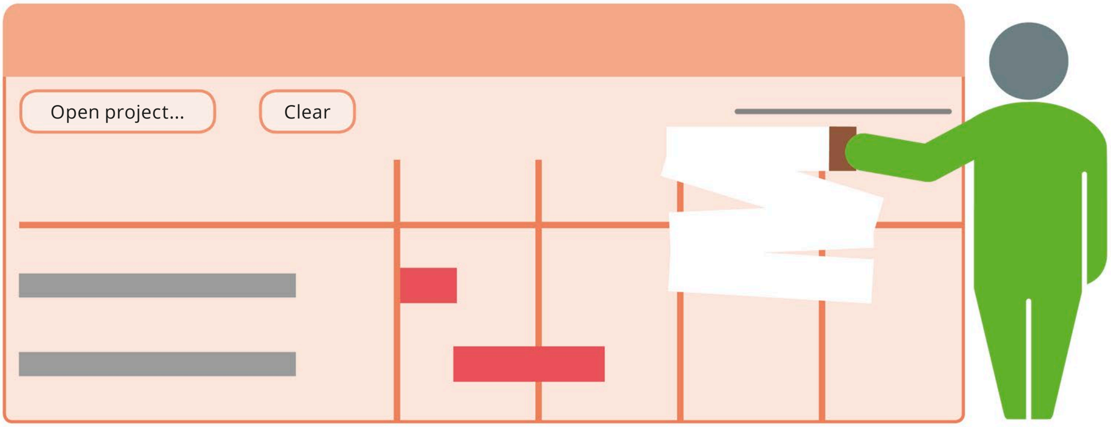
```
    draw_chart(tasks, canvas)
    filename_label.config(text=filename)
def clear_canvas():
    filename_label.config(text="")
    canvas.delete(tkinter.ALL)
root = tkinter.Tk()
root.title("Project Planner")
open_button = tkinter.Button(root, text="Open project...", \
                             command=open_project)
open_button.pack(side="left")
clear_button = tkinter.Button(button_frame, text="Clear", \
                              command=clear_canvas)
clear_button.pack(side="left")
filename_label = tkinter.Label(button_frame)
canvas = tkinter.Canvas(root, width=800, height=400, bg="white")
canvas.pack(side="bottom")
```

Updates the text attribute of the label to an empty string

Deletes all of the existing items on the drawing canvas

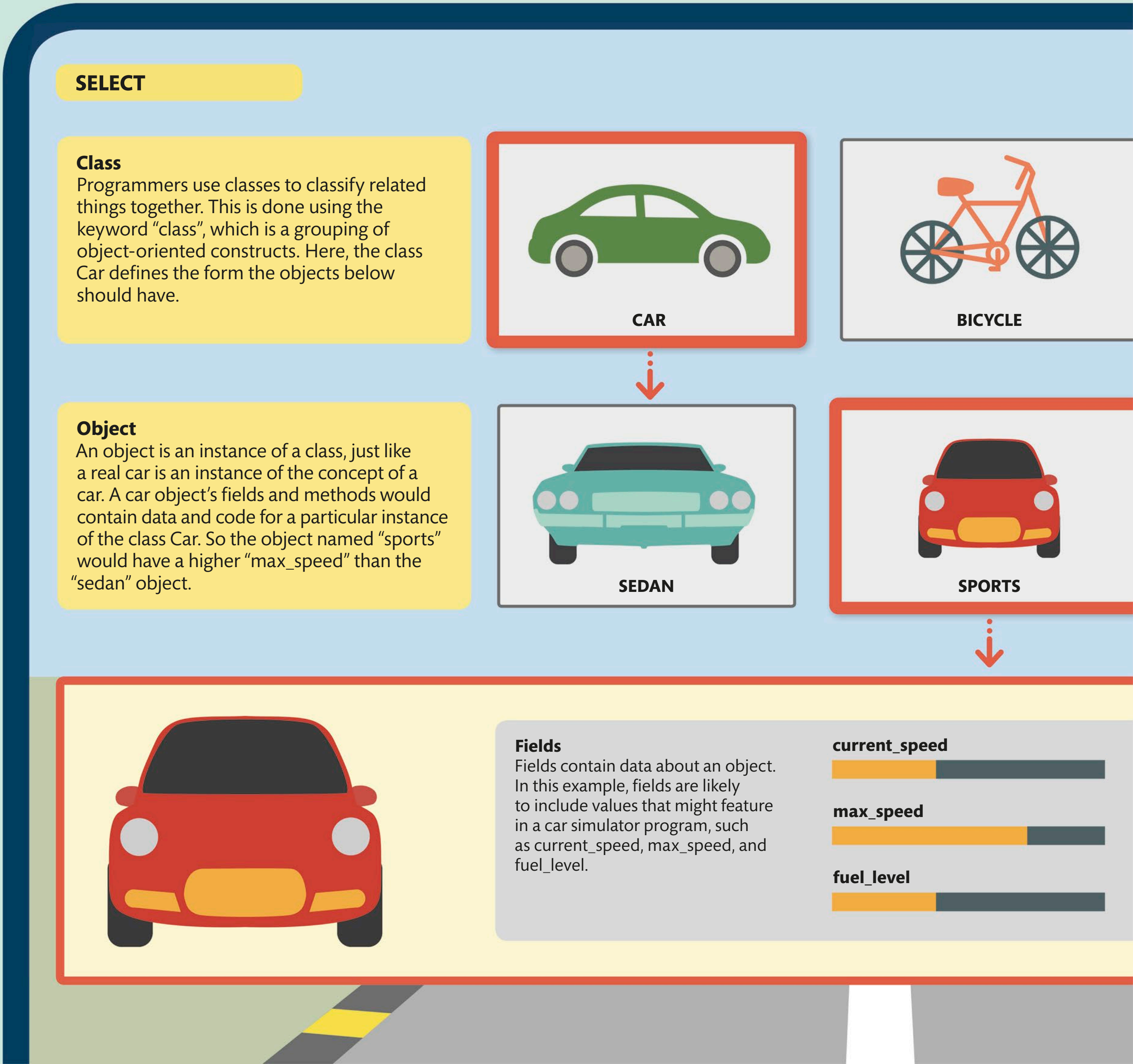
Creates a new button inside the window that will call the `clear_canvas()` function when pressed

Places the new button on the left side of the window



Objects and classes

One of Python's most important features is that it is an object-oriented language (see p.25). This means that data in Python can be arranged in terms of classes and objects, which allows users to have one blueprint from which multiple objects can be created.



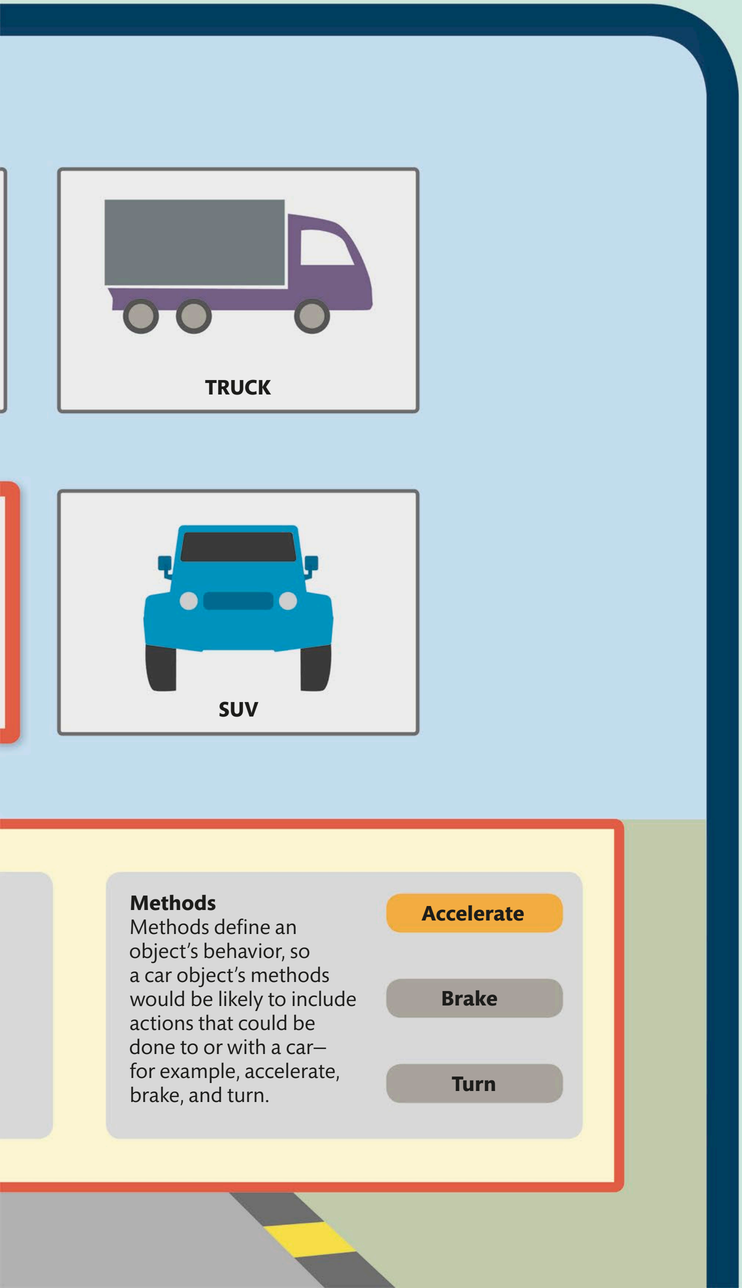
Class
Programmers use classes to classify related things together. This is done using the keyword "class", which is a grouping of object-oriented constructs. Here, the class Car defines the form the objects below should have.

Object
An object is an instance of a class, just like a real car is an instance of the concept of a car. A car object's fields and methods would contain data and code for a particular instance of the class Car. So the object named "sports" would have a higher "max_speed" than the "sedan" object.

Fields
Fields contain data about an object. In this example, fields are likely to include values that might feature in a car simulator program, such as current_speed, max_speed, and fuel_level.

What are objects and classes?

An object is a data type that is modeled after a real-world item, such as a car, allowing programmers to create a computer representation of it. Objects usually consist of two parts: fields, containing data, and methods, containing code. A class, on the other hand, defines the form a particular object should have. It is like the "idea" of an object and lays out the types of field that object would have and what its methods would do.



Methods
Methods define an object's behavior, so a car object's methods would be likely to include actions that could be done to or with a car—for example, accelerate, brake, and turn.

Describes the attributes common to any car

Fields of the class Car

```
class Car:
    current_speed = 0
    max_speed = 0
    fuel_level = 0
    def accelerate(self):
        print("speeding up")
    def break(self):
        print("slowing down")

my_car = Car()
my_car.current_speed = 4.5
my_car.max_speed = 8.5
my_car.fuel_level = 4.5
```

Methods of the class Car

Sets the fields of the **my_car** object to specific values

Instantiating a class
A program that allows users to model the functioning of a car might include the class Car, with attributes common to all cars. A user's car (here, a sports model) would then be an object, with fields containing values related to that particular car and methods defining the actions done with the car.