FireFire.
Note that code I write here is truncated. I may not write arguments that may be needed in the function as well.

Okay, I made main.py first.

```
import pygame
from pygame import DOUBLEBUF, HWSURFACE

def main():
	pygame.init()


main()
```

You need a start screen, the main screen, and also the end screen, so you add in the outline into the function as well. At this point you also need all the window basics, ie. screens etc.

```
import pygame
from pygame import DOUBLEBUF, HWSURFACE

def main():
	pygame.init()

	settings = Settings()
	screen = pygame.display.set_mode(settings.screen_dimensions, DOUBLEBUF | HWSURFACE)
	pygame.display.set_caption("Lel")

	while True:
		start_screen()
		main_game()
		end_screen()


main()
```
I also create a Settings class to store some generic settings. **You do not actually need to do this, but if you are scaling a game up, would you rather sieve through hundreds of lines of code to change one specific value whenever it is used or store that value somewhere and only change it there?**

It is fine to just put functions that you have not created yet to give you an outline of what you need to do. Your IDE will scream at you but ignore it like a boss.

Work on start_screen(). You need to show messages, and also display a background picture.
At this point I realised that these 2 things will change a lot depending on which screen you are on, so I created classes for all of them for easy instantiation. You can view them on your own.

They can be completed by this point, with a few logic that might not be apparent now. For example, the speed for which the background will move and whether they will repeat or not.

```
def start_screen():
    prompts = […]
    chosen_prompt = random.choice(prompts)    # Add imports as you go.

    picture_path = r'Images/screen.png'
    background = Background(screen, picture_path, speed=0.35, repeat=True)

    game_name = Message(screen, "Fire Pew Pew", (250, 0, 0), 100, size='big')
    start_prompt = Message(screen, f"Press 'SPACE' to {chosen_prompt}", (250, 0, 0), -200)

    clock = pygame.time.Clock()

    interval = 0
      while True:
        for event in pygame.event.get():
            full_quit(event)

            if event.type == pygame.KEYDOWN and event.key == pygame.K_SPACE:
                return

        background.update()
        game_name.show_message()
        interval = show_start_prompt(interval, start_prompt)

        pygame.display.flip()
        clock.tick(140)
```

Use Clock() to make sure you are getting a rather fixed FPS. For the number in tick(), make sure to use the same value as well for other game loops so the values for speeds etc. you use will be consistent.

full_quit() is a refactor of the function to make sure the whole function is not so logically heavy and easier to read at the same time. This will prove useful since other game loops will be using it as well later on.

show_start_prompt() is a fancy function to make the prompt blink. You don’t actually need this but it was a nice thing to add :)

Notice I do a `return` in one logic gate. This is the thing that will bring us to the main game loop.

Returning to our main game function main(), we can now add some logic to see how the different screens interact.

```
def main():
	--snip--

	return_to_start = True
	while True:
		if return_to_start:
			start_screen()  # While we are on the start screen, the code is only here.

		score = main_game()
		if score is None:  # This means the player chooses to restart in the middle
			return_to_start = False
			continue

		return_to_start = end_screen()  # Whether return to start screen or restart immediately
```

You can also guess how you might want a function to work even on the outline. Here I guess that main_game() will return a score value while end_screen() will return a value based on where we want to go after the end screen.

Notice that there is actually no way we will break out of the loop in main(). This is okay. `sys.exit()` will close everything for us. Remember to do `import sys`.

Let’s take a look at one of the classes. Specifically, the background class.

```
class Background:
	def __init__(self, screen, picture_path, speed=float(0), repeat=False):
		# Screen settings
		self.screen = screen
		self.screen_rect = self.screen.get_rect()

		# Image
		self.image = pygame.image.load(picture_path).convert_alpha()

		# Position background
		self.rect = self.image.get_rect()
		self.left = float(0)
		self.centery = float(self.screen_rect.centery)
		self.rect.left = self.left
		self.rect.centery = self.centery

		--snip--

```

convert_alpha() changes the formatting of the picture to something that Pygame can understand easily. For large projects, this can give significant improvements to the speed and efficiency of the game.

Note how the rects are initialised and how a rect is positioned. You may have noticed that I do not put values to a rect directly. I do this for 2 reasons:

1. This is how pygame works
2. In order to do operations on a rect value, they need to be done indirectly of the rect value itself. Do all operations on a value to hold the rect value you want, then assign it back to the rect value itself later on. (Don't ask me why, it's just how Pygame works. However, you may have guessed that this is due to how pixels need to be integers and can never be floats.)

This method of placing and initialising values will also be used for the other classes that we will create later on.

Back to the main code.

Once we return from start screen, the code will move on to main_game().

```
def main_game(screen, settings):
	picture_path = r"Images/BACKGROUND.png"
	background = Background(screen, picture_path, speed=settings.bg_speed, repeat=False)
	firetruck = FireTruck(screen, settings)

	clock = pygame.time.Clock()

	while True:
		for event in pygame.event.get():
			if event.type == pygame.QUIT:
				sys.exit()

			if … and event.key == pygame.K_p and pause_screen(screen):
				reset_game(settings, fires, waters)
				return

		pygame.display.flip()
		clock.tick(140)
```

We get a new picture for the background in the main game, and also create the FireTruck class since we know for sure the firetruck will appear here. We might not be doing anything with it now but I chose to add it in at this point of time.

Also note pause_screen(). This is the function that will decide what to do when the player presses ‘p’ to pause.

```
def pause_screen(screen):
	Message(screen, "Gamed Paused", (0, 153, 0), 100, size='big').show_message()
	Message(screen, "Press 'Q' to quit, 'C' to continue, 'R' to retry", (0, 153, 0), …).show_message()
	pygame.display.flip()

	while True:
		for event in pygame.event.get():
			full_quit(event)

			if event.type == pygame.KEYDOWN:
				if event.key == pygame.K_c:
					return

			if event.key == pygame.K_r:
				return True
```

Usually I do not like making such nested statements but I don’t think there is any other way here.

Note how it will return different values based on which button we pressed. Refer back to the previous code-blocks to see how the different return values give different results. This is also how we decide where to move on from here.

We also used full_quit() here again.

Since there is logic to bring us to the end screen, we make the end_screen() function next.

```
def end_screen(screen, score):
	Message(screen, "Game Over!", (0, 153, 0), 100, size='big').show_message()
	Message(screen, f"Your score is {score}", (0, 153, 0), -150, ...).show_message()
	Message(screen, "Press 'R' to retry, 'E' to go back to main menu, ...).show_message()
	pygame.display.flip()

	while True:
		for event in pygame.event.get():
			full_quit(event)

			if event.type == pygame.KEYDOWN:
				if event.key == pygame.K_r:
					return False

				if event.key == pygame.K_e:
					return True
```

Again, refer back to the previous code-blocks to see how the different values will interact with the different screens. We also used full_quit() here again. All hail refactorisation and code reuse.

Also note the number of messages we have already created! A Message class helps to keep things neat and easy to re-initialise. Imagine making a font object and then making it a surface everytime we need a message. We let the class do the work. Sometimes, readability of code takes precedence, especially if you are working as a team.

If you haven't already, you might want to run the game once to make sure everything does what you want it to do. You don't want to check too late (having to sieve through numerous lines of code to find an error) or too early (don't check everytime you add one line of code).

If you haven't realised, the bulk of the game has actually been completed! The only thing to add now would be the main game logic. All your screens should work and flow as intended.

Let's go back to main_game().

```
def main_game(screen, settings):
	--snip--

	fires = FireGroup()
	waters = Group()  # from pygame.sprite import Group

	clock = pygame.time.Clock()

	while True:
		--snip--

		if firetruck.update():
			score = (fires.total_spawns - len(fires)) / fires.total_spawns
			reset_game(settings, fires, waters)
			return score

		pygame.display.flip()
		clock.tick(140)
```

The Group class is an efficient way for us to work with sprites. We will see how later.

FireGroup() is a subclass of Group that I created to keep track of the number of fires created. You don't actually need to do this.

You can also instantiate a variable before the game loop and increment it everytime a fire is created. But can you imagine having to pass that variable through all the functions anytime it is needed? Let the group do this behind the scenes for you. The variable will go wherever the group goes, and this will prove useful, since we have to add the fire into the group anyways.

Let’s take a look at the update() method for the firetruck class, since (as you may have seen), it contains the logic to bring us to the end screen.

```
class FireTruck:
	--snip--

	# Image
	self.image = pygame.transform.rotozoom(pygame.image.load(...).convert_alpha(), 0, 0.5)

	--snip--

	def move(self):
		self.right += self.xmove
		self.rect.right = self.right

		if self.rect.right >= self.screen_rect.right + 20:  # When firetruck at right edge of screen
			return True

	def update(self):
		self.blitme()
		if self.move():
			return True
```

I did some rough guessing with regards to the ‘20’. In game, this means that the end screen will show when the firetruck reaches the end of the screen.

Refer to the previous code-block to see how the return value affects whether the game ends or not.

For self.image, you are free to use pygame.transform.scale, but I prefer using rotozoom because scaling is done proportionately for width and height automatically.

For the path to the file, take note about whether you are using WindowsPath (\) or PosixPath (/). Python will automatically convert forward slashes to backward slashes if you are on Windows, so using ‘/’ should be good. If it does not work and you are cross-platform, take a look at the pathlib module.

Right, now all we need is the fire and water sprites and the logic for them. We will add the Fire Sprite first.
```
class Fire(Sprite):  # from pygame.sprite import Sprite
	def __init__(self, screen, settings, firetruck, background):
		Sprite.__init__(self)
		self.settings = settings

		--snip--
		self.mask = pygame.mask.from_surface(self.image)
		--snip--

	def move(self, background):
		self.centerx -= background.xmove
		self.rect.centerx = self.centerx

	def blitme(self):
		self.screen.blit(self.image, self.rect)

	def update(self, background):
		self.move(background)
		self.blitme()
```
Sprites and Groups are very special (in fact, they are found under the same module, pygame.sprite) because they provide us with an efficient way of dealing with multiple instances of the same type of object and also between ‘Groups’ of such objects. Simply make Fire (and also Water later on) a subclass of Sprite and you are good to go.

`.update()` is a ‘special’ method. This is because the Group class also has a `.update()` method. When using the `.update()` method on a group of sprites, it will do the exact same method on each and every sprite’s `.update()` method.

There is also another kind of method that is roughly similar to this called `.draw()`, which, as you might have guessed, draws each sprite in the group. However, we can just create our own `blitme()` method and put it inside `update()` so update() moves and also draws the sprite on the screen for us.

We also pass `background` to update() so the speed of the fire is the same as that of the background at all times.

Making a mask for the image also ensures that the hitbox of the fire only contains pixels that are actually coloured. Wouldn’t it be magical if you shot a water about 2 times the width away from the fire but it still hit, just because the fire picture would be that big?

Now, add a function to update fires in main_game().
```
def main_game(screen, settings):
	--snip--

	while True:
		for event in pygame.event.get():
			if event.type == pygame.QUIT:
				sys.exit()

			if … and event.key == pygame.K_p and pause_screen(screen):
				reset_game(settings, fires, waters)
				return

		# Update background
		background.update()

		# Update fire
		update_fire(screen, settings, fires, firetruck, background)
```
Make sure to do it after updating the background, or the background will draw over whatever you did for the fires. In fact, try to do it by layers; anything at the back comes first followed by everything else.

```
def update_fire(screen, settings, fires, firetruck, background):
	# If chance and background is still scrolling
	if random.random() <= settings.fire_spa... and background.rect.right >= settings.screen_width:
		create_fire(screen, settings, fires, firetruck, background)

	fires.update(background)
```
```
def create_fire(screen, settings, fires, firetruck, background):
	fire = Fire(screen, settings, firetruck, background)
	fires.add(fire)
	fires.total_spawns += 1
```
Use the random module to create a chance for a fire to spawn.

I do ` background.rect.right >= settings.screen_width` because of this line in the Fire class:
	```self.centerx = float(random.randint(self.ft_rect_left, self.background_rect.right – 300))```

to ensure that the limits are met and the game won’t crash if `self.ft_rect_left` becomes greater than `self.background_rect.right – 300`.

That statement ensures that no fires spawn behind the firetruck. This is to ensure that no fires will spawn outside on the left of the screen. That wouldn’t be fair for the player, although in real life fires don’t care about your feelings.

	```

Here, we can also see how making a special group for the fire pays off. We are 2 levels deep into a function call before we increment the total fire count. Passing a variable this deep before doing anything with it would probably confuse you and drive you nuts.

If you did everything correctly, the fires will now spawn randomly as you drive through the neighbourhood.

How do you put the fires out then? Go ahead and make a Water class for your water sprites. Similarly, make sure to include a mask, and pass `background` to update() so it moves at the same speed as the background.

There will be differences between the Fire and Water class. However, you can still make a simple outline for the class. Judging from the fire class, you will probably need `screen` and `settings` in your class so pass that to __init__() first.

For how to detect when to shoot the water, I chose to use the mouse. However, you are free to do whatever method you want.

```
def main_game(screen, settings):
	--snip--

	while True:
		for event in pygame.event.get():
			if event.type == pygame.QUIT:
				sys.exit()

			if … and event.key == pygame.K_p and pause_screen(screen):
				reset_game(settings, fires, waters)
				return

			# Create water when conditions are met
			check_create_water(screen, settings, waters, event)

		# Update background
		background.update()

		# Show number of water left
		show_water_left(screen, settings)

		# Update fire
		update_fire(screen, settings, fires, firetruck, background)

		# Update water
		waters.update(background)

		--snip--
```

For the water, we are unable to make one function to create and update the water at the same time. This is due to how pygame deals with detecting mouse inputs. You might be tempted to make another event loop so you can group them together but this will ensure that not all your mouse inputs will be detected. I have tried. Also this means that your water objects will only update if you click. Not what you want.

In order to check mouse inputs, you have to do it in an event loop. This is why check_create_water() is inside the main event loop for main_game().

```
def check_create_water(screen, settings, waters, event):
	if event.type == pygame.MOUSEBUTTONDOWN and pygame.mouse.get_pressed()[0] and
settings.max_water > 0:
	create_water(screen, settings, waters)
```
```
def create_water(screen, settings, waters):
	coor = pygame.mouse.get_pos()
	water = Water(screen, settings, coor)
	settings.max_water -= 1
	waters.add(water)
```
Subtract 1 everytime you create a water object so you don’t get to shoot water freely. Your fire engine is not connected to a hydrant. We want the water to spawn where the user clicks, so get the position of the mouse when the user clicks and pass it into Water class.

show_water_left() simply states the number of water you can shoot remaining on the screen.
```
def show_water_left(screen, settings):
	water_left = Message(screen, f"Waters left: {settings.max_water}", (230, 0, 0), 0)
	water_left.rect.topleft = (0, 0)
	water_left.show_message()
```

To make the water extinguish the flames, check for collisions. This is also only efficiently possible with Groups and Sprites. Good luck using nested for loops (which is a bad idea for a script that needs efficiency) to check each sprite with each sprite for both the water and fire otherwise.
```
def main_game(screen, settings):
	--snip--

	while True:
		for event in pygame.event.get():
			if event.type == pygame.QUIT:
				sys.exit()

			if … and event.key == pygame.K_p and pause_screen(screen):
				reset_game(settings, fires, waters)
				return

			# Create water when conditions are met
			check_create_water(screen, settings, waters, event)

		# Update background
		background.update()

		# Show number of water left
		show_water_left(screen, settings)
```

```
                # Update fire
                update_fire(screen, settings, fires, firetruck, background)

                # Check water fire collisions
                pygame.sprite.groupcollide(waters, fires, False, True,
collided=pygame.sprite.collide_mask)

                if firetruck.update():
                        score = (fires.total_spawns - len(fires)) / fires.total_spawns
                        reset_game(settings, fires, waters)
                        return score

                pygame.display.flip()
                clock.tick(140)
```
Your game is completed! Everything actually runs as it should if you did everything correctly.

But wait, how about reset_game()?
```
def reset_game(settings, *args):
        settings.max_water = 20
        for i in args:
                i.empty()
```
Make sure to empty the fire group and water group after every game or if the player restarts so you won't be acting on them in the next game. Also, reset settings.max_water to its original value or you will be forced to play the next game while staring at the carnage being unable to do anything.

Let's check back to our main() function.
```
def main():
        --snip--

        return_to_start = True
        while True:
                if return_to_start:
                        start_screen()

                score = main_game()
                if score is None:
                        return_to_start = False
                        continue

                return_to_start = end_screen()
```
Take note of what values end_screen() can return and how this will effect the screen to be drawn after the end screen. Remember to call main() or your script will just be a bunch of functions doing nothing. That should be it. Take care now.