algorithmics

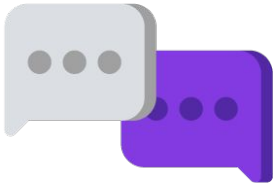
Module 2. Lesson 2.

# Interface design

Link to guidelines

## Discussion:

# Test application



# The client appreciated your work

The creators of the Crazy People YouTube channel and their subscribers really liked the program from ProTeam. Now they want to run another competition with prizes.

The finalists of the competition will have to answer questions about the Crazy People channel, so the YouTubers need **an application with questions that will display messages about the prizes people win**.

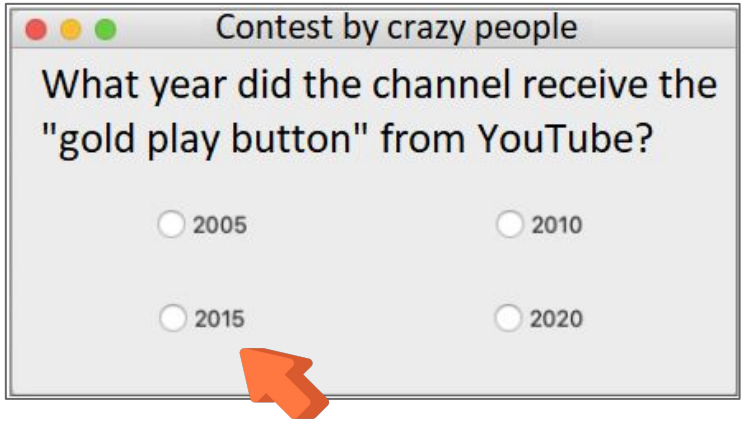
*Ready to get started?*

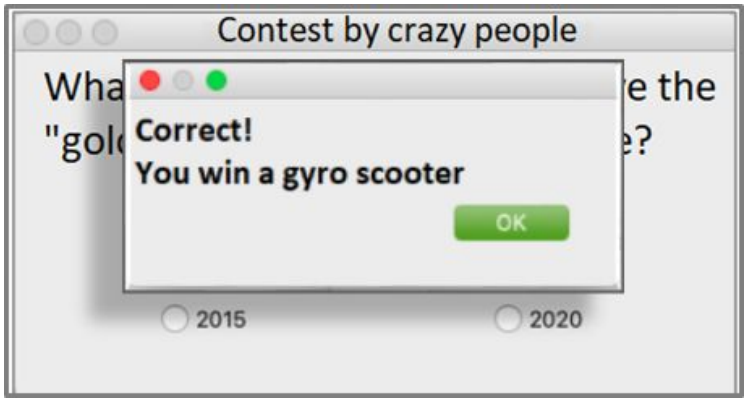
*Cole,*
*senior developer*

Discussing work tasks

# Let’s look at the task

- Product type: windowed application.
- Functionality:
  - display a test question with answer options;
  - display a prize depending on the answer selected.




What tools do we need to complete such a request?

Discussing work tasks

# Let’s look at the task

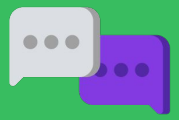
The "Competition" application

Application functionality

- Nothing complicated.

Application interface

- Creating a question window with a choice of possible answers.
- Arranging the widgets in the question window.
- Processing mouse events with interface elements.
- Displaying a result window depending on the answer.



Discussing work tasks

# Let’s look at the task

The "Competition" application

Application functionality

- Nothing complicated.

*We need to learn new widgets and better understand layouts.*

Application interface

- Creating a question window with a choice of possible answers.
- Arranging the widgets in the question window.
- Processing mouse events with interface elements.
- Displaying a result window depending on the answer.

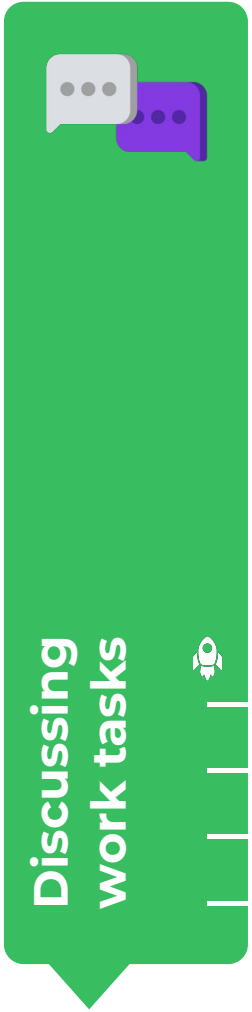
Discussing work tasks

# The goal of the work day is

***to learn new widgets in the PyQt library and create an app for a trivia competition.***

# Today you will

- review what a class, class constructor, windowed application, and widgets are;
- learn how to arrange widgets in layouts and combine multiple layouts into one;
- create a new PC app!



# Qualifications

# Show your knowledge of classes and the PyQt library.

Qualifications

What is an **object**?
Property? Method?

What is a **class**?
Do you know ready-made object classes?

Qualifications

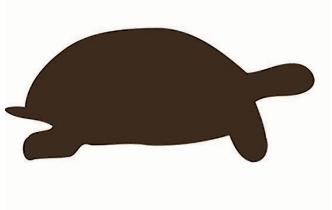
## A property

is a variable inside an object.

## A method

is a function inside an object.

---



It's a turtle

**Object**

**Object class**

**Properties + Methods**

**Our knowledge about all these objects**

Qualifications

# What does a class consist of?
# How do we create a class?

Qualifications

# Creating classes

To create a class, we need to do the following:

- list the **properties** in the constructor that define the characteristics of an instance of the class;
- list the **methods** for managing an instance.

```
class [Class name]():
    def __init__(self, [Value]):
        self.[Property name] = [Value]
    def [Method name](self):
        [Action with object and properties]
        [Action with object and properties]
```

**What is a constructor ?**
**What is self?**

Qualifications

# Creating classes

To create a class, we need to do the following:

- list the **properties** in the constructor that define the characteristics of an instance of the class;
- list the **methods** for managing an instance.

```
class [Class name]():
    def __init__(self, [Value]):
        self.[Property name] = [Value]
    def [Method name](self):
        [Action with object and properties]
        [Action with object and properties]
```

A special **constructor** function that creates an instance of a class with the specified properties.

__init__

**Two underscores.**

Qualifications

# Creating classes

To create a class, we need to do the following:

- list the **properties** in the constructor that define the characteristics of an instance of the class;
- list the **methods** for managing an instance.

```
class [Class name]():
    def __init__(self, [Value]):
        self.[Property name] = [Value]
    def [Method name](self):
        [Action with object and properties]
        [Action with object and properties]
```

**self** is a parameter indicating the object to which the method is applied.

**self.property** is a property of the object to which the method is applied.

Qualifications

# What is a windowed application ?
# What does it consist of?

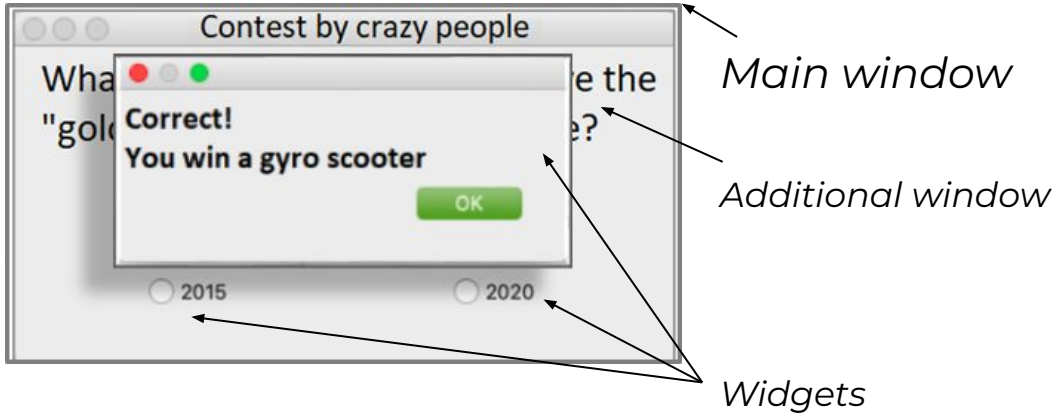
Qualifications

# A windowed application

**is a computer program that interacts with the user through a <u>graphical interface.</u>**

A standard windowed application consists of the following:

- ❑ a main **window**,
- ❑ controls (**widgets**),
- ❑ additional windows (optional).




Qualifications

# How do we <u>create</u> an app?
# What widgets do you know?

Qualifications

# PyQt5 —

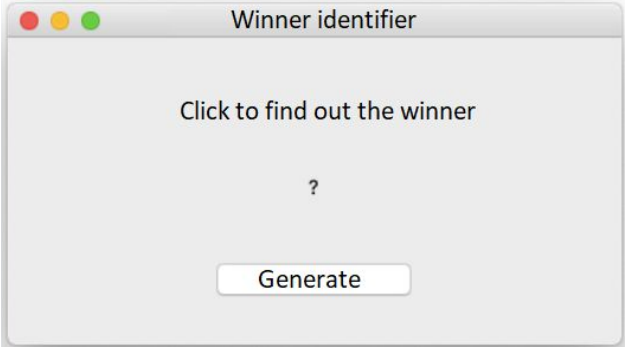
**is a cross-platform library for building windowed applications.**

```
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QLabel, QVBoxLayout
```

| *Object* | *Designation* |
|---|---|
| Application | `QApplication` |
| Application window | `QWidget` |
| Label | `QLabel` |
| Button | `QPushButton` |
| Vertical guide line | `QVBoxLayout` |

Qualifications

```python
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QLabel, QVBoxLayout

app = QApplication([])
main_win = QWidget()
main_win.setWindowTitle('Winner Identifier')
button = QPushButton('Generate')
text = QLabel('Click to find out the winner')
winner = QLabel('?')

line = QVBoxLayout()
line.addWidget(text, alignment = Qt.AlignCenter)
line.addWidget(winner, alignment = Qt.AlignCenter)
line.addWidget(button, alignment = Qt.AlignCenter)
main_win.setLayout(line)

main_win.show()
app.exec_()
```




*Example of an application implemented earlier.*

# What is event handling ?
# How do we handle mouse events ?

Qualifications

# Event handling (button clicking)

1. *Describe actions* when clicking on a button in a separate processor function.
2. *Use a command* to link the function and the widget.

```
def show_winner():
```

```
button.clicked.connect(show_fun_title)
```

**THE FUNCTION IS DESCRIBED SEPARATELY**

**NAME OF THE PROCESSOR FUNCTION WHICH THE BUTTON RESPONDS TO**

Qualifications

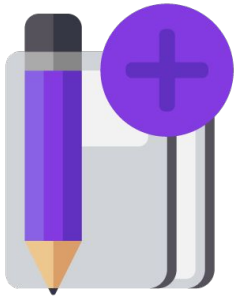
# Qualifications confirmed!

Great, you are ready to brainstorming and work on your tasks!

Qualifications

## Brainstorming:

# Object-oriented programming



# PyQt library structure

The PyQt library has a complex **hierarchical structure**.

Some classes have a shared ancestor class, so their instances have similar properties and methods.

To be able to use the PyQt tools in a meaningful way, let's learn more about **inheritance** in object-oriented programming.

**QtCore**

The **Qt** module:

constants and flags which determine how the window looks

**QtWidgets**

The **QApplication, QWidget, QLabel,** and **QVBoxLayout** modules:

widgets and how they are arranged

. . .

Brainstorming

# Classes and subclasses

If we think of examples of classes from real life, we will find that some classes are "descendants" of other classes.

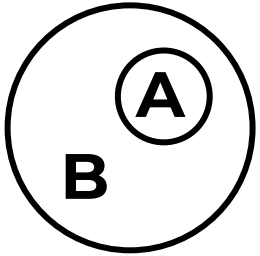
***all buttons are widgets***

**all cats are animals**

**all desks are tables**

***all comets are celestial bodies***

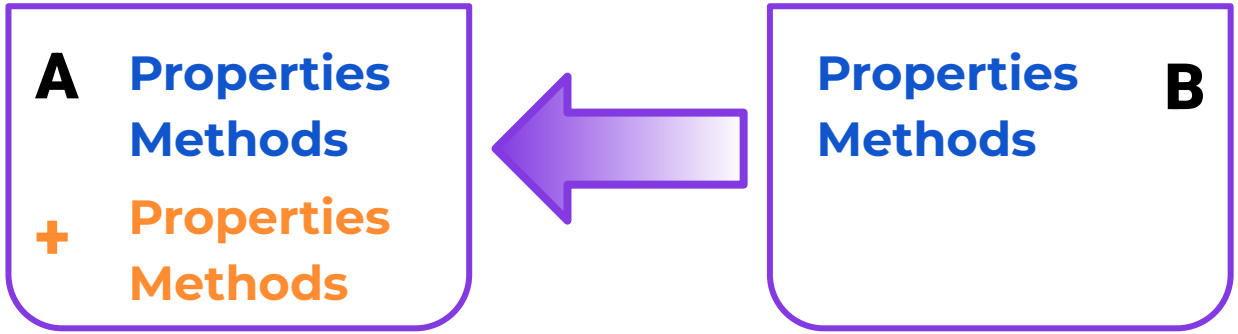
**all cars are vehicles**



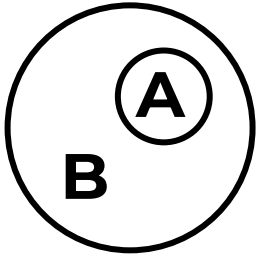

Brainstorming

# Inheritance

Class inheritance helps us **transfer all the skills** previously written for a **more general class into** another, more private class, the **inheritor class**.




**Inheritor class** **Superclass**




**class A is nested within class B**

Brainstorming

# Superclass and inheritor class

If the super class is already written, then,

to create an inheritor class, we will need to do the following:

- when creating the inheritor, specify the name of the superclass;
- add the necessary methods to the inheritor class;

```
class Inheritor name(Superclass name):
    def __init__(self, Value):
        super().__init__(Value)
    def Method name(self):
        Action with object and properties
```

*Option in which **no new properties are introduced**.*

*The inheritor class is only supplemented with a **new method**.*

Brainstorming

# Superclass and inheritor class

If the super class is already written, then,

to create an inheritor class, we will need to do the following:

- when creating the inheritor, specify the name of the superclass;
- add the necessary methods to the inheritor class;

```
class Inheritor name(Superclass name):
    def __init__(self, Value):
        super().__init__(Value)
    def Method name(self):
        Action with object and properties
```

**super** refers to the superclass.

The inheritor constructor works like a superclass constructor.

Brainstorming

# Superclass and inheritor class

If the super class is already written, then,

to create an inheritor class, we will need to do the following:

- when creating the inheritor, specify the name of the superclass;
- add the necessary methods to the inheritor class;
- create an inheritor constructor.

```
class Inheritor name(Superclass name):
    def __init__(self, Value, Value):
        super().__init__(Value)
        self.New property = Value
    def Method name(self):
        Action with object and properties
```

*Option in which **new properties are introduced**.*

*The constructor adopts the properties of the superclass and adds a new one.*

Brainstorming

# Creating classes

**Example (for console).** Application and Mobile Application.

The Application class stores information about the application's name, description, and size.

Let's create an inheritor class called Mobile Application with a new property and method.

```
class Application():
    def __init__(self, title_text, description_text, volume_num):
        self.title = title_text
        self.description = description_text
        self.volume = volume_num
    def print_info(self):
        print('Application', self.title)
        print('Description:', self.description)
        print('Application size:', self.volume)
```

Brainstorming

```python
class Application():
    def __init__(self, title_text, desription_text, volume_num):
        #...
    def print_info(self):
        #...
class MobileApplication(Application):
    #constructor of the inheritor class with a new field
    def __init__(self, title_text, desription_text, volume_num, system_type_text):
        super().__init__(title_text, desription_text, volume_num)
        self.system_type = system_type_text
    #new method of the inheritor class
    def setup_application(self):
        print('Installation of', self.system_type, 'the application is complete')

mobile_app = MobileApplication('My Notes', 'Smart notes and checklists', 50,
'Android')
mobile_app.print_info()
mobile_app.setup_application()
```

Brainstorming

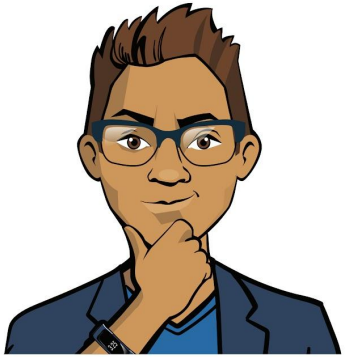
```python
class Application():
    def __init__(self, title_text, description_text, volume_num):
        #...
    def print_info(self):
        #...
class MobileApplication(Application):
    def __init__(self, title_text, desription_text, volume_num, system_type_text):
        #...
    def setup_application(self):
        #...
mobile_app = MobileApplication('My Notes', 'Smart notes and checklists', 50,
'Android')
mobile_app.print_info()
mobile_app.setup_application()
```

```
The My Notes application
Description: Smart notes and checklists
App size: 50
Installation of the Android application is
complete
Bash-3.2$
```

Brainstorming

# Before continuing, let's check the following:

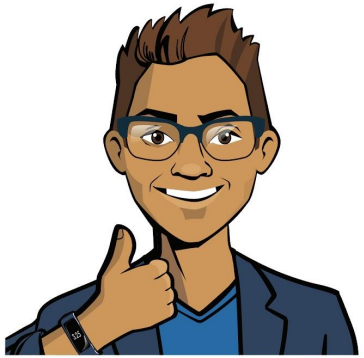
1. What will the previous program display if we create an instance of MobileApplication with the following data:
   - `'NeedForSpeed'`
   - `'A game in the "Racing" genre'`
   - `150`
   - `'ios'`
2. Is it possible to create an instance of the Application class in the previous program?
3. The lead developer suggested adding another version field (app version) to the MobileApplication class. How can we do it?



Brainstorming

# Conclusions :

1. Class inheritance helps us transfer all the skills previously written for a more general class into another, more private class, the inheritor class.
2. The following is described in the inheritor class:
   - a new constructor (borrowing superclass properties using the **super()** method),
   - new properties (in the constructor),
   - new methods.



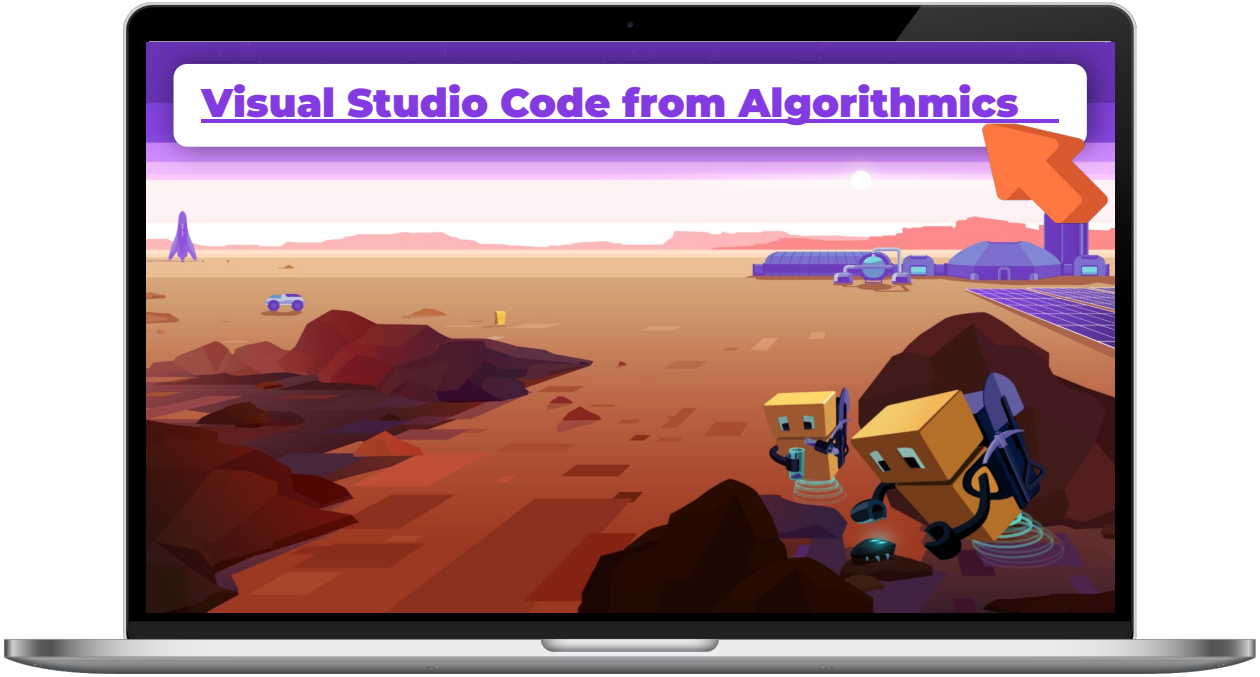
Brainstorming

## Visual Studio Code:

# More preparation for creating applications

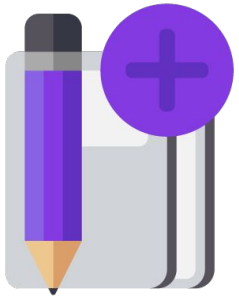
# Complete the task in VS Code

## VSC. Inheritance



## Brainstorming:

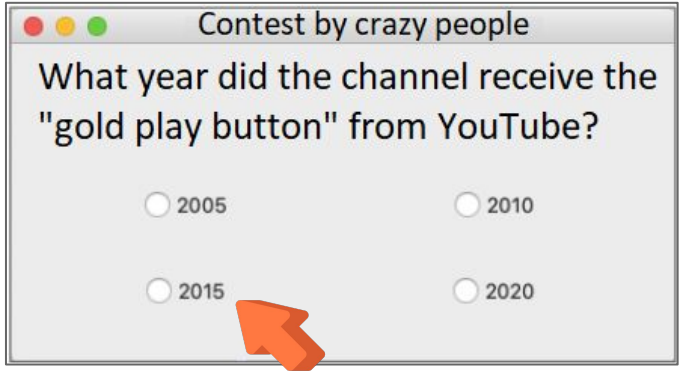
# Interface design

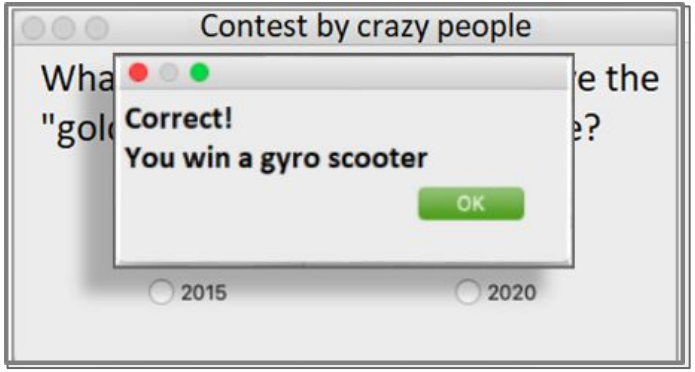


# Let’s look at the task

To complete the task, we will need to do the following:

- learn new widgets for creating message boxes and radio buttons;
- learn how to create > 1 layout and place widgets in them.




Brainstorming

# The "Competition" application

**Step 1.** Creating the necessary widgets.

| *Object* | *Designation* |
|---|---|
| Application | `QApplication` |
| Application window | `QWidget` |
| Label | `QLabel` |
| Guide line (vertical, horizontal) | `QVBoxLayout, QHBoxLayout` |
| Message window | `QMessageBox()` |
| Radio button | `QRadioButton` |

*How do we import the necessary library modules?*

Brainstorming

# Importing PyQt modules

**QtCore**

The **Qt** module:

constants and flags which determine how the window looks

It needs to be imported for the `alignment` parameter.

**QtWidgets**

The **QApplication, QWidget, QLabel,** and **QVBoxLayout** modules:

widgets and how they are arranged

It needs to be imported for the corresponding widgets.

. . .

```
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import (
    QApplication, QWidget, QLabel, QVBoxLayout, QHBoxLayout, QMessageBox
)
```

Brainstorming

# Creating an application and main window

```
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QLabel, QVBoxLayout

app = QApplication([])
my_win = QWidget()
my_win.show()
app.exec_()
```

Don't forget to resize the window and choose where it will appear.

Brainstorming

# Creating a question and answer options

*Answer option*:

| *Method* | *Purpose* |
|---|---|
| `btn_answer = QRadioButton('Signature')` | **A constructor** that creates a "Radio button" type object with a signature. |

*Question (you already know this widget)*:

| *Method* | *Purpose* |
|---|---|
| `question = QLabel('What year?')` | **A constructor** that creates a "Label" type object with the specified text. |

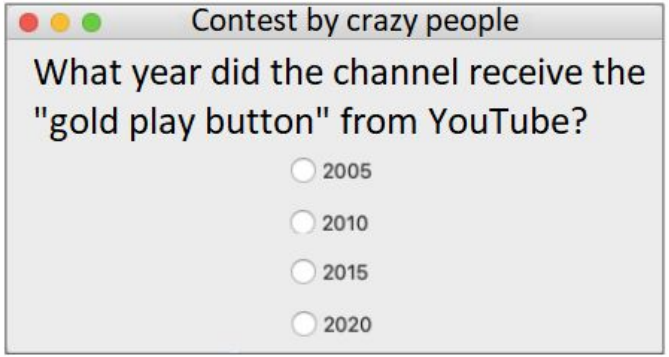
Brainstorming

# Positioning widgets along a line

| *Method* | *Purpose* |
|---|---|
| `v_line = QVBoxLayout()` | **A constructor** that creates a "Vertical line" type object. |
| `v_line.addWidget(`<br>`  title, alignment = Qt.AlignCenter`<br>`)` | A method that adds a widget to the line and positions it in the center. |
| `my_win.setLayout(`**`v_line`**`)` | Adds the resulting line and its objects to the application window. |

Brainstorming

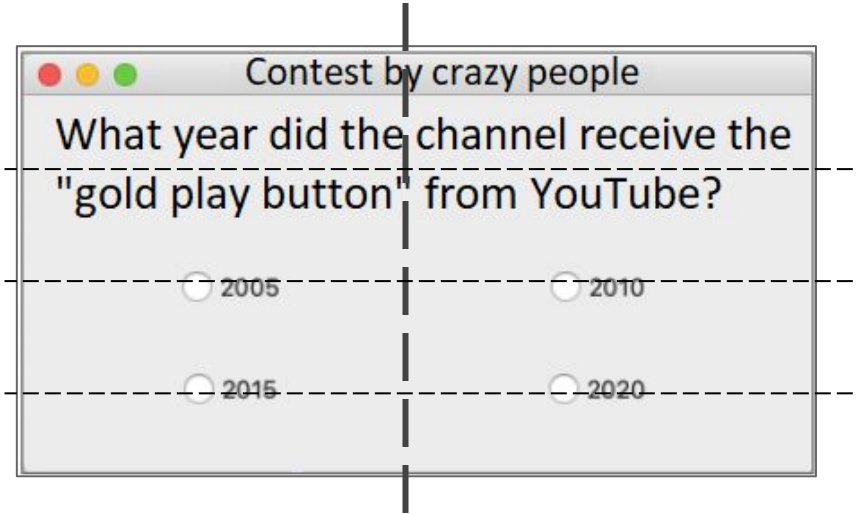
```python
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QApplication, QWidget,  QPushButton, QHBoxLayout, QVBoxLayout, QLabel,
QMessageBox, QRadioButton

app = QApplication([])
main_win = QWidget()
main_win.setWindowTitle('Competition from Crazy People')
question = QLabel('What year did the channel receive the "gold play button" from YouTube?')
btn_answer1 = QRadioButton('2005')
btn_answer2 = QRadioButton('2010')
btn_answer3 = QRadioButton('2015')
btn_answer4 = QRadioButton('2020')
layout_main = QVBoxLayout()
layout_main.addWidget(question, alignment = Qt.AlignCenter)
layout_main.addWidget(btn_answer1, alignment = Qt.AlignCenter)
layout_main.addWidget(btn_answer2, alignment = Qt.AlignCenter)
layout_main.addWidget(btn_answer3, alignment = Qt.AlignCenter)
layout_main.addWidget(btn_answer4, alignment = Qt.AlignCenter)

main_win.setLayout(layout_main)
main_win.show()
app.exec_()
```




*The application with a question and four possible answers.*

Brainstorming

# Positioning widgets using layouts



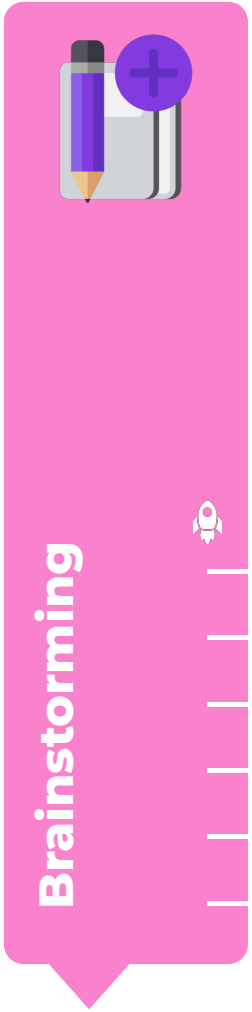

To position the widgets the way the customer wants, you need to do the following:

- ❑ Create three horizontal guide lines.
- ❑ Add the necessary widgets to each line. We will get three layouts with widgets.
- ❑ Add these layouts to the main vertical line.
- ❑ Bind the main layout (vertical line layout) to the application window.

Brainstorming

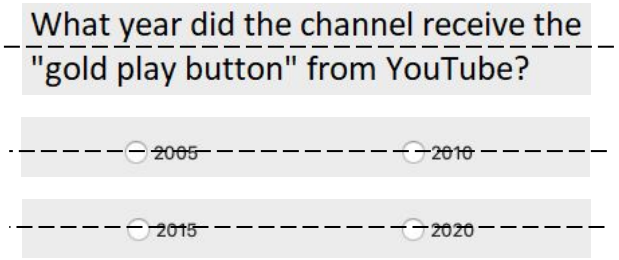
# Positioning widgets using layouts

```
layoutH1 = QHBoxLayout()
layoutH2 = QHBoxLayout()
layoutH3 = QHBoxLayout()
```



# Positioning widgets using layouts

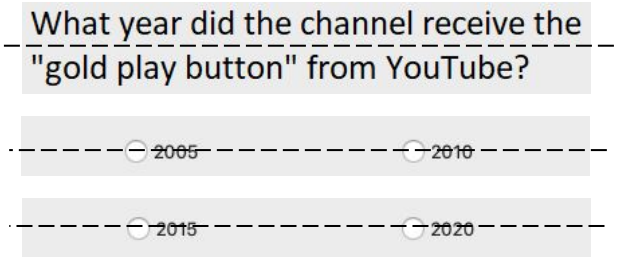
```
layoutH1 = QHBoxLayout()
layoutH2 = QHBoxLayout()
layoutH3 = QHBoxLayout()
layoutH1.addWidget(question, alignment = Qt.AlignCenter)
layoutH2.addWidget(btn_answer1, alignment = Qt.AlignCenter)
layoutH2.addWidget(btn_answer2, alignment = Qt.AlignCenter)
layoutH3.addWidget(btn_answer3, alignment = Qt.AlignCenter)
layoutH3.addWidget(btn_answer4, alignment = Qt.AlignCenter)
```




Brainstorming

# Positioning widgets using layouts

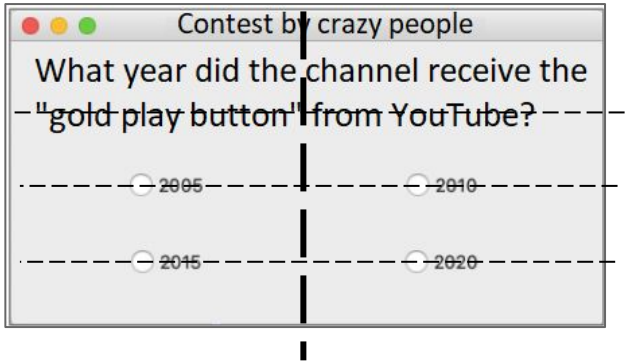
```
layoutH1 = QHBoxLayout()
layoutH2 = QHBoxLayout()
layoutH3 = QHBoxLayout()
layoutH1.addWidget(question, alignment = Qt.AlignCenter)
layoutH2.addWidget(btn_answer1, alignment = Qt.AlignCenter)
layoutH2.addWidget(btn_answer2, alignment = Qt.AlignCenter)
layoutH3.addWidget(btn_answer3, alignment = Qt.AlignCenter)
layoutH3.addWidget(btn_answer4, alignment = Qt.AlignCenter)
layout_main = QVBoxLayout()
layout_main.addLayout(layoutH1)
layout_main.addLayout(layoutH2)
layout_main.addLayout(layoutH3)
```




Brainstorming

# Positioning widgets using layouts

```
layoutH1 = QHBoxLayout()
layoutH2 = QHBoxLayout()
layoutH3 = QHBoxLayout()
layoutH1.addWidget(question, alignment = Qt.AlignCenter)
layoutH2.addWidget(btn_answer1, alignment = Qt.AlignCenter)
layoutH2.addWidget(btn_answer2, alignment = Qt.AlignCenter)
layoutH3.addWidget(btn_answer3, alignment = Qt.AlignCenter)
layoutH3.addWidget(btn_answer4, alignment = Qt.AlignCenter)
layout_main = QVBoxLayout()
layout_main.addLayout(layoutH1)
layout_main.addLayout(layoutH2)
layout_main.addLayout(layoutH3)
main_win.setLayout(layout_main)
```




Brainstorming

# Creating a notification window

*A window with a notification label and a ready-made "OK" button:*

| *Method* | *Purpose* |
|---|---|
| `victory_win = QMessageBox()` | **A constructor** that creates a notification window. |
| `victory_win.setText('Correct!')` | **A method** that displays the specified text in the window. |
| `victory_win.exec_()` | Leaves the window open. |

*The button is created automatically.*

*Clicking on "OK" closes the window.*


Brainstorming

# Radio button click handling

The customers want the following to happen:

- when clicking on the correct radio button ("2005"), a window appears saying "Correct! You win a gyro scooter!”
- when clicking on any other radio button, a window appears saying "No, it was in 2005. You win a company poster."

*How do we program this?*

Brainstorming

# Button click handling

Possible solution. Write two processor functions.

- the correct radio button responds to the processor function that creates a window with a message about winning;
- the other radio buttons respond to the function that creates the window with the correct answer.

```
def show_win():
```

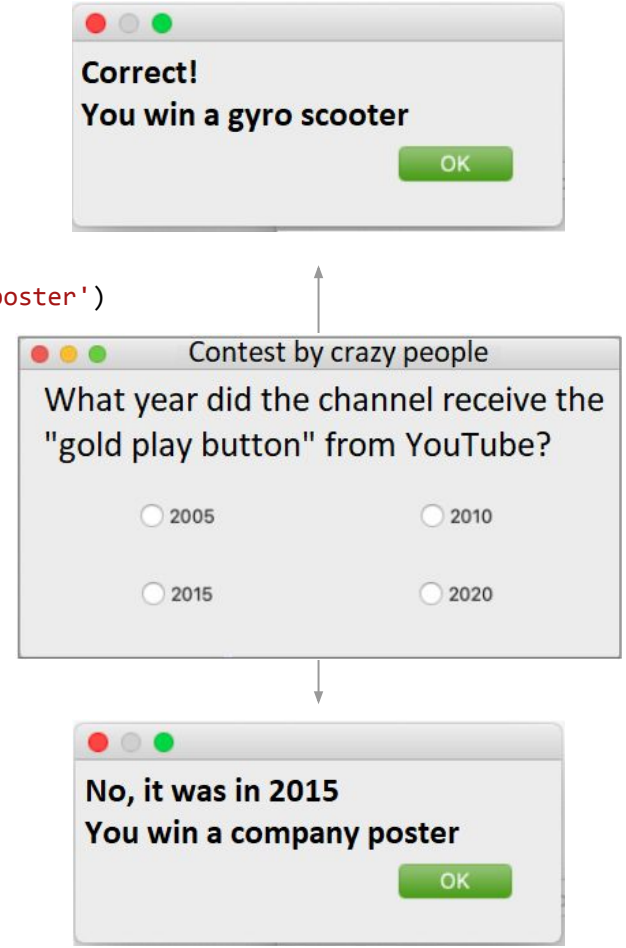
```
btn_answer1.clicked.connect(show_win)
```

**THE FUNCTION IS DESCRIBED SEPARATELY**

**NAME OF THE PROCESSOR FUNCTION WHICH THE BUTTON RESPONDS TO**

Brainstorming

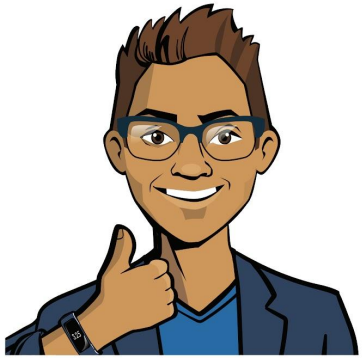
```python
#...importing libraries
def show_win():
    victory_win = QMessageBox()
    victory_win.setText('Correct!\nYou win a gyro scooter')
    victory_win.exec_()
def show_lose():
    victory_win = QMessageBox()
    victory_win.setText('No, it was in 2015.\nYou win a company poster')
    victory_win.exec_()
#...creating an application and main window
#...creating question and answer widgets
#...creating layouts and adding widgets

btn_answer3.clicked.connect(show_win)
btn_answer1.clicked.connect(show_lose)
btn_answer2.clicked.connect(show_lose)
btn_answer4.clicked.connect(show_lose)
main_win.show()
app.exec_()
```




Brainstorming

# Conclusions :

1. The **QRadioButton** widget is responsible for radio buttons, and the **QMessageBox** is responsible for notification windows.
2. Multiple widgets can be added to the same layout, and the layouts themselves can be nested within one another.

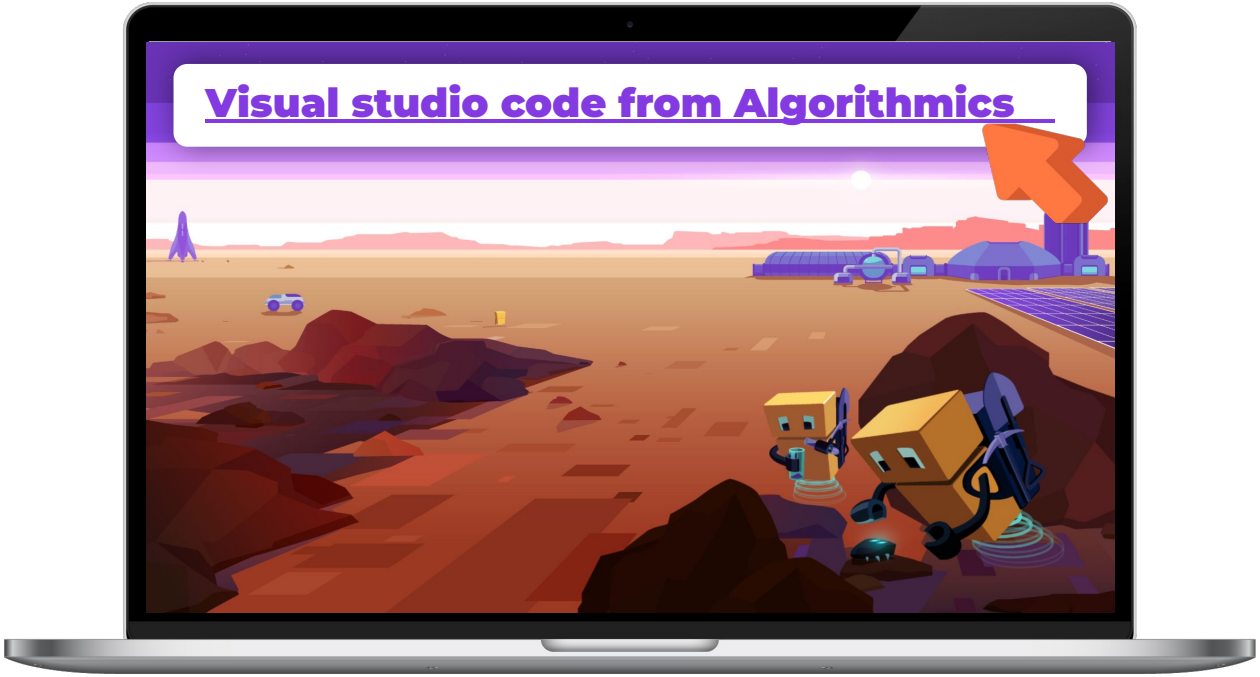


Brainstorming

# Visual Studio Code:
# Competition

# Complete the task in VS Code

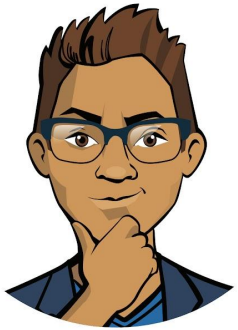
## VSC. Crazy people: competition



# End of the work day

# Let’s end the work day by answering these technical questions:

1. What is a superclass? Inheritor class?
   What does an inheritor class contain?
2. How does the PyQt library work?
   What widgets did you learn today?
3. What is a layout? How do we position widgets using layouts?



*Cole,*
*senior developer*

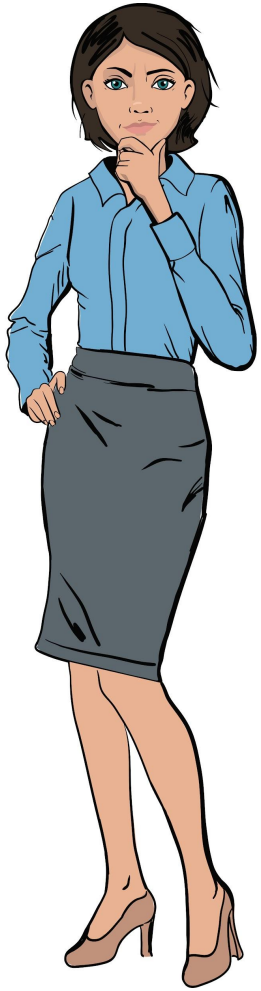
*Emily,*
*project manager*

Summing up the work day

# Evaluating your work performance

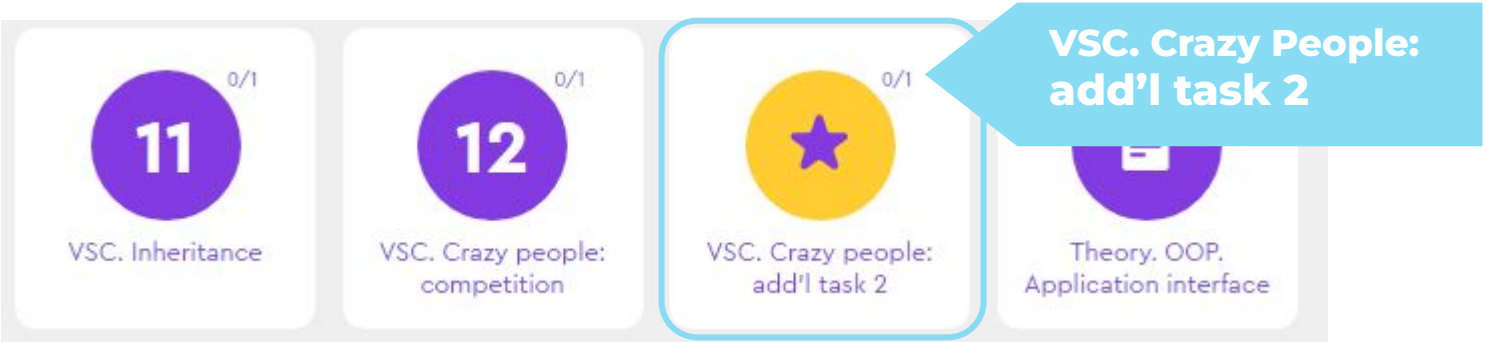
Answer the questions with your colleagues:

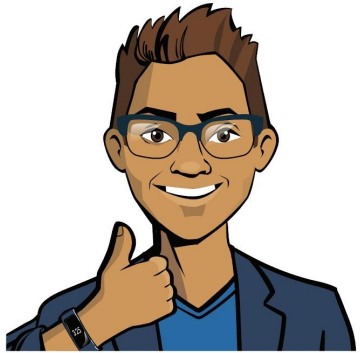
1. What worked best?
2. What didn't work out the way you wanted?
3. What can you do to avoid setbacks next time?



Summing up the work day

# Additional tasks to improve your performance

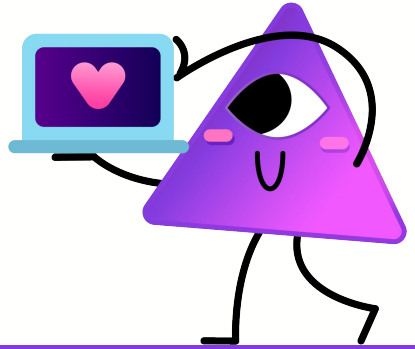


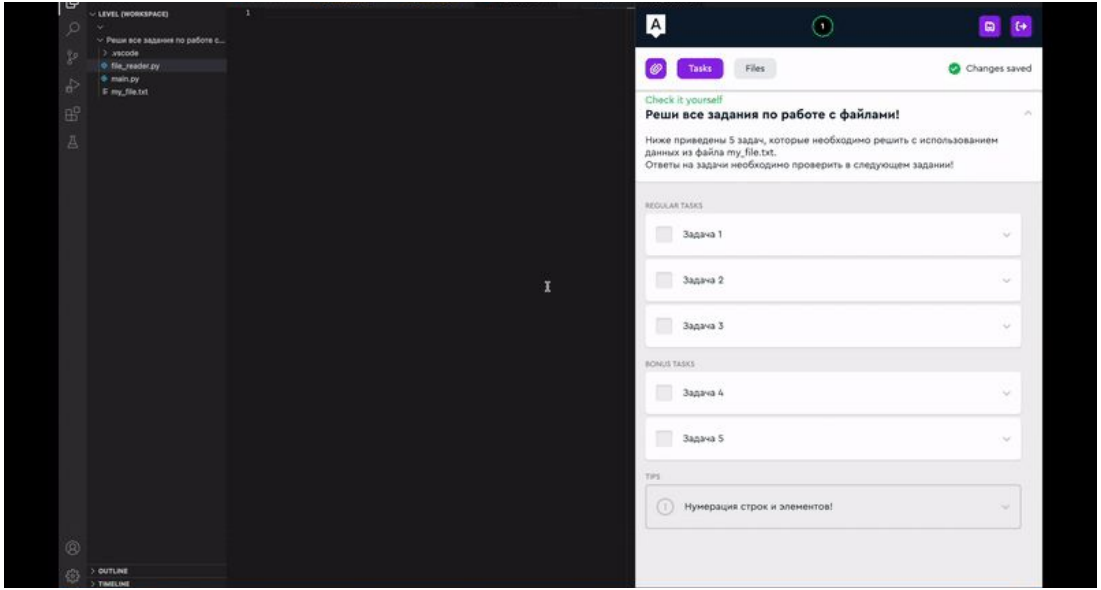
Summing up the work day

Laboratory

# Publishing VS. Code projects



# Save the project

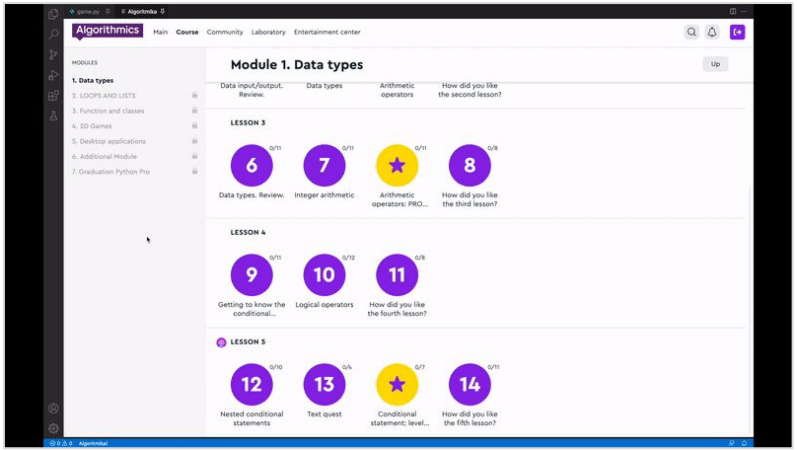



1. Click on the icon 
2. Enter the name.

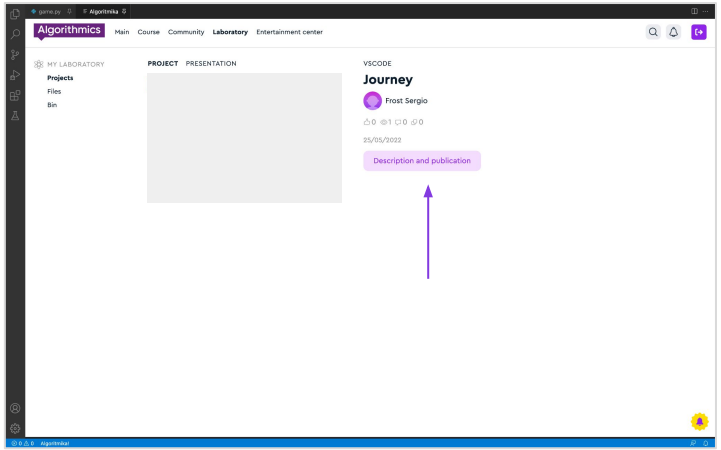
Use only Latin letters, numbers and the sign "_" The name should not be the same as other project names



# Go to the lab



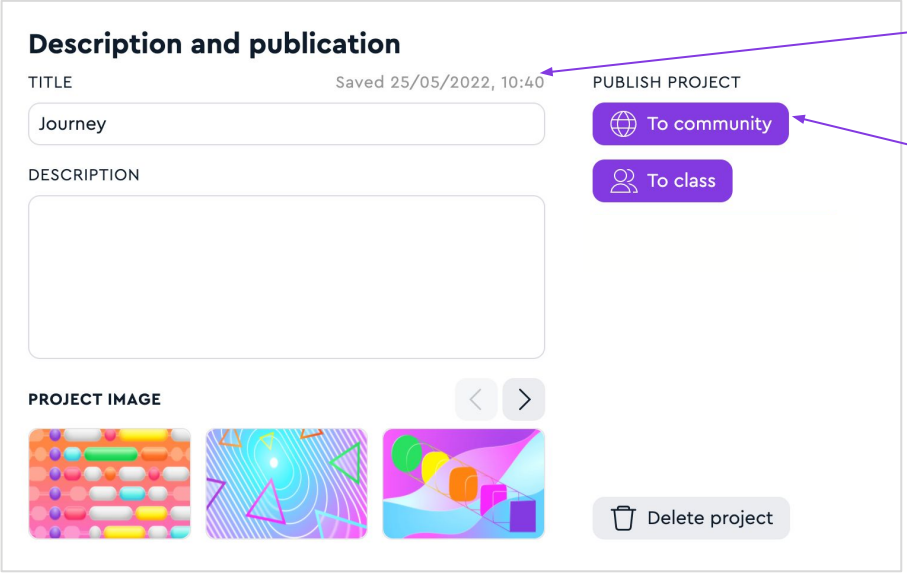

Click on the project to proceed to editing the description and setting up the publication




Click the "Description and Publication" button



# Add a description and publish




2. 1. Enter the name and description

2. 2. Publish your project to the Hall of Fame or Class