# How to Program Computer Games

Do you have an idea for a computer game and want to make it real? Or have you ever wondered how computer games are written? Or are you even thinking of working in game development? This wikiHow will show you how to write different types of...

Part 1 of 4:

## Making a Text-Based Game

1. 


**Choose a programming language.** All programming languages are different, so you will have to decide which one to use to write your game. Every major programming language supports text input, text output and if-constructions (the main things you need for a simple text-based game), so you can decide yourself. Here are some factors to consider:

1. Field of application: some programming languages, like JavaScript, are designed to be used for web sites, others, like Python, C or C++, are designed to run on a computer. Some languages have one specific purpose, like R, which is mainly used for statistical analysis. For your game, you should use a language with a broader application field (Python, C, C++, JavaScript, many others).
2. Ease of use: although writing a program should be easy enough after some practice in any normal programming language (i. e. not one specifically designed to be confusing and unusable like Malbolge), some are friendlier to beginners than others. Java and C, for example, require the programmer to understand more programming concepts than Python. Also, Python error messages are easier to understand for a beginner than, for example, C error messages.
3. Platform compatibility: you probably want people on different systems, such as Linux, Mac or Windows, to all be able to play your game. So you shouldn't use a language that is only supported on a few systems, like for example Visual Basic, which is only supported on Windows. This is also

a good reason not to code your game in Assembler, which is specific to the system and processor, and is also quite hard to program with.

This article will use Python for the examples of a text-based game, but you can look up how the concepts are done in any other programming language.

2. **Install the necessary tools.** You need something to compile or interpret your program with, and you need something to write/edit it with. If you want to follow the example in this article, you should install Python and learn how to run programs. If you want to, you can set up some IDE or use one that Python provides (it's called IDLE), but you can also just use your favourite text editor that supports plain text.
3. **Print some text.** The player will want to know what is going on and what they have to do, so you should print some text for them.
    1. This is done with the `print()` function in Python. To try it out, open a new file with the .py extension, enter the following code into it, save and run it:

        ```
        print("Welcome to the number guessing game!") print(
        "Enter a whole number between between 1 and 1000:")
        ```

4. **Introduce some randomness into your game.** If nothing is random, the game will be exactly the same every time, and the player will get bored quickly.
    1. In this case, a number should be randomly chosen at the start of the program so that the player won't always guess the same number. Since it should remain the same throughout the program, you should store the random number in a variable.
    2. Python doesn't provide a random number function in its core. But it has a standard library (this means the user won't have to install anything extra) that does. So go to the beginning of your code (before the `print()` functions) and type the line `import random`.
    3. Use the random function. It is called `randint()`, is in the `random` library which you just imported, and takes the minimal and maximal value the number can have as argument. So go back to the end of your code and enter following line:

        ```
        rightNum = random.randint(0,1000)
        ```

5. **Get the player's input.** In a game, the player wants to do something or interact with something. In a text-based game, this is possible by entering text.
    1. Since the code you entered prints the instruction to enter a number to the player, it should also read the number they enter. This is done with `input()` in Python 3, and `raw_input()` in Python 2. You should write in Python 3, as Python 2 will become outdated soon. Add the following line to your code to store the player's input in a variable called `number`:

        ```
        userNum = input()
        ```

6. **Turn the user's input into a usable data type.**
    1. Make the player's input a number. Now, this might seem confusing because they just entered a number. But there is a good reason: Python assumes that all input is text, or how it is called in programming, a string. And this text contains the number you want to get. Python provides a function to convert a string that only contains a number to the number inside. Type:

        ```
        userNum = int(userNum)
        ```

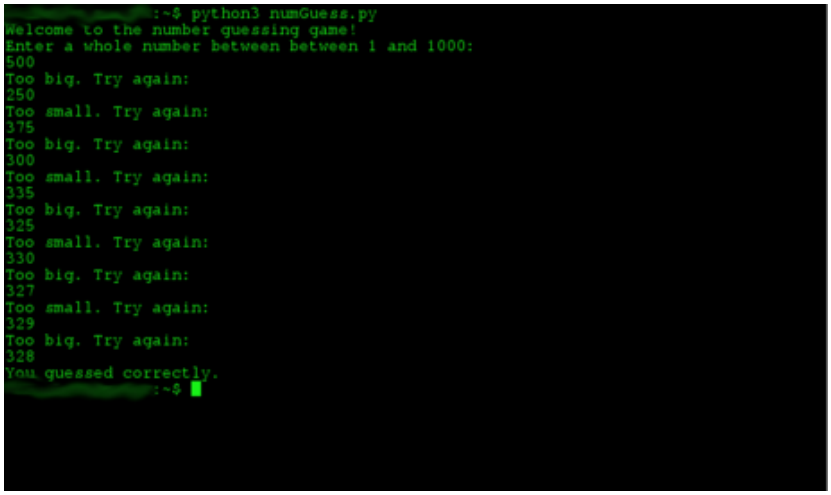
7. **Process the user's input.** It would be pointless to just ask the player to enter random things. You should actually do something with the information that the user entered.

1. Compare the user's number to the correct number. While the numbers are not the same, it should make the user enter another number. When the numbers match, it should stop getting new input, tell the user that they guessed correctly, and quit the program. This is done with the following code:

   ```
   while userNum != rightNum: userNum = int(input())
   ```

8. **Give the player feedback.** While you already have processed the input, the user can't see this. You need to actually print the result to the user so they understand what's happening.
   1. Surely, you could just tell the user whether their number is right or wrong. But with that approach, the player would have to guess 1000 times in the worst case, which would be very boring.
   2. So tell the player whether their number is too small or too big. This will reduce the number of guesses significantly: If, for example, the user guesses 500 first, and is told that it's too big, there are only 500 possible numbers now instead of 1000. This is done with if-constructions, so replace the `print("Wrong. Try again.")` with one.
   3. Be aware that checking whether two numbers are the same is done with == , not with = . = assigns the value right of it to the variable left of it!

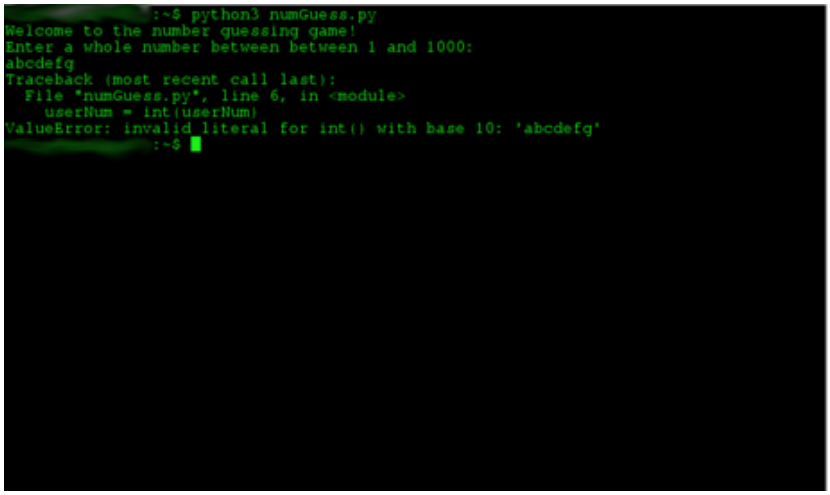
```
if userNum  rightNum: print("Too small. Try again:") if userNum >
rightNum: print("Too big. Try again:")
```

9. 


**Test your code.** As a programmer, you should be sure that your code works in all cases before considering it finished.

1. When programming in python, make sure that you get the indentations correct. Your code should look like this:

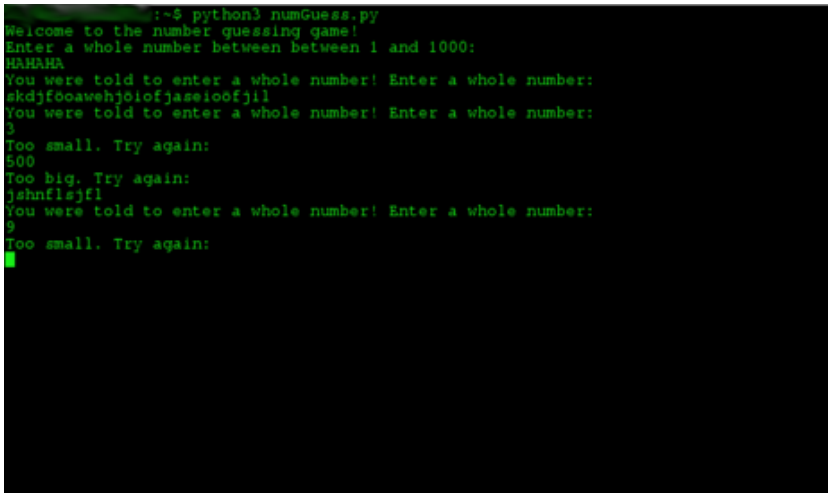
   ```
   import random print("Welcome to the number guessing game!") print(
   "Enter a whole number between 1 and 1000:") rightNum = random.
   randint(0,1000) userNum = input() userNum = int(userNum) while
   userNum != rightNum: if userNum  rightNum: print(
   "Too small. Try again:") if userNum > rightNum: print(
   "Too big. Try again:") userNum = int(input()) print(
   "You guessed correctly.")
   ```

10. 


**Validate the input.** A user shouldn't be able to break your program with simple actions like entering the wrong thing. Validating the input means whether the user entered the correct thing before processing it.

1. Open the game again and try entering anything that's not a number. The game will exit with a `ValueError`. To avoid this, you should implement a way to check whether the input was a number.
2. Define a function. Since validating the input is quite long, and you have to do it multiple times, you should define a function. It will take no arguments and return a number. First, write `def numInput():` at the top of your code, directly under the `import random`.
3. Get the user's input once. Use the `input()` function and assign the result to the variable `inp`.
4. While the user's input is not a number, ask them to enter a number. Checking whether a string is a number is done with the `isdigit()` functions, which only allows a whole number, so you won't have to check for that separately.
5. When the input is a number, convert it from string to number and return the result. Use the `int()` function for converting the string to an integer. This will make the conversion in the main code unnecessary, and you should remove it from there.
6. Replace all calls to `input()` in the main code with calls to `numInput()`.
7. The code of the `numInput()` function will look like this:

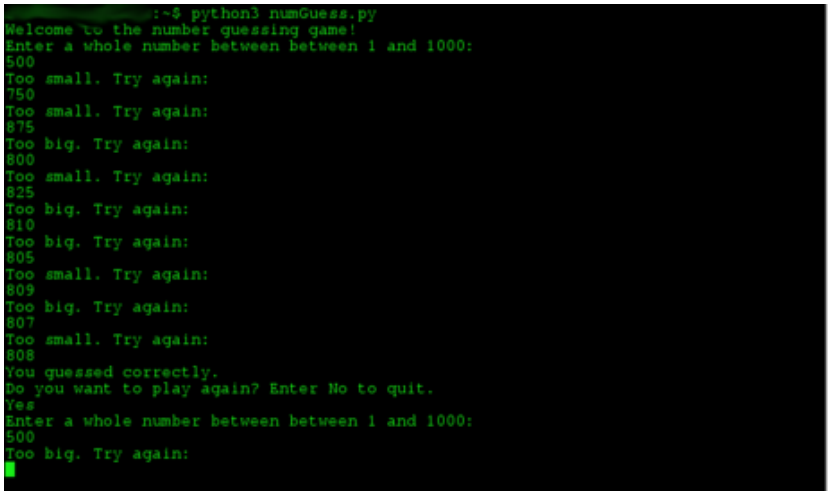
```
def numInput(): inp = input() while not inp.isdigit(): print(
"You were told to enter a whole number! Enter a whole number:") inp =
input() return int(inp)
```

11. 


**Test the game again.** Especially pay attention to whether your input validation works by entering something wrong on purpose.

1. Try entering some text when the program asks you for a number. Now, instead of exiting with an error message, the program will ask you for a number again.

12. **Suggest restarting the game when it finishes.** This way, the player could play your game for a longer time without having to constantly restart it.
    1. Put all code except the import and the function definition into a while-loop. Set `True` as the condition: this will always be true, so the loop will continue forever.
    2. Ask the player whether they want to play again after they guessed the number correctly. Use the `print()` function.
    3. If they answer "No", break out of the look. If they answer anything else, continue. Breaking out of a loop is done with the `break` statement.
    4. Move the "Welcome to the number guessing game" outside the while loop. The player probably doesn't want to be welcomed every time they play the game. Move the instruction `print("Welcome to the number guessing game!"` above the `while True:`, so it will be printed only once, when the user starts the first game.

13. 


**Test the game.** You need to be sure that your game still works after implementing new features.

1. Make sure to answer both "Yes" and "No" at least once to make sure that both options work. Here is what your code should look like:

```
import random def numInput(): inp = input() while not inp.isdigit
(): print(
"You were told to enter a whole number! Enter a whole number:") inp
 = input() return int(inp) print(
"Welcome to the number guessing game!") while True: print(
"Enter a whole number between 1 and 1000:") rightNum = random.
randint(0,1000) userNum = numInput() while userNum != rightNum: if
userNum  rightNum: print("Too small. Try again:") if userNum >
rightNum: print("Too big. Try again:") userNum = numInput() print(
"You guessed correctly.") print(
"Do you want to play again? Enter No to quit.") if input() == "No":
 break
```

14. **Write other text-based games.** How about writing a text adventure next? Or a quiz game? Be creative.

    **Tip**: It's sometimes helpful to look in the documentation if you're not sure how something is done or how a function is used. The Python 3 documentation is found at https://docs.python.org/3/. Sometimes searching for whatever you want to do on the internet also returns good results.

Part 2 of 4:

## Making a Game with 2D Graphics

1. **Choose a library.** Making graphics is very complicated, and most programming languages (including Python, C++, C, JavaScript) provide only minimal or even no support for graphics in the core or the standard libraries. So you'll have to use an external library to be able to make graphics, for example Pygame for Python.
    1. Even with a graphics library, you'll have to worry a lot about low-level things like how to display a menu, how to check whether the user clicked on it, how to display the tiles, and so on. If you prefer to focus on developing the actual game, and especially if the game you want to make is complex, you should use a game engine library, which implements such things.

    This article will use Python with Cocos2D to show how to make a simple 2D platformer. Some of the mentioned concepts may not exist in other game engines. Refer to their documentation for more information.

2. **Install the library you chose.** Cocos2D for Python is simply installed with `sudo pip3 install cocos2d`.
3. **Make a new directory.** You will use things like images and sounds in your game. You should keep these things in the same directory as the program, and the directory shouldn't contain anything else so that you can easily see what assets you have in the game.
4. **Change into the new directory and create a new code file.** It should be named `main`, with the file extension of your programming language. If you write a large and complex program where it makes sense to have multiple program files, this will show which one is the main one.
    1. In this example, this file, which should be called `main.py`, will contain all your code. But the directory you created will still be useful for other media files.
5. **Make a window.** This is the basic prerequisite for a game with graphics. You can add the simplest content now, like for example a background colour.
        1. Import the necessary cocos2d sub-modules: `cocos.director`, `cocos.scene` and `cocos.layer`. This is done with `from subModuleName import *`, where

subModuleName is the submodule you want to import. The difference between `from ... import *` and `import ...` is that you don't have to put the module name in front of everything you use from that module with the former.

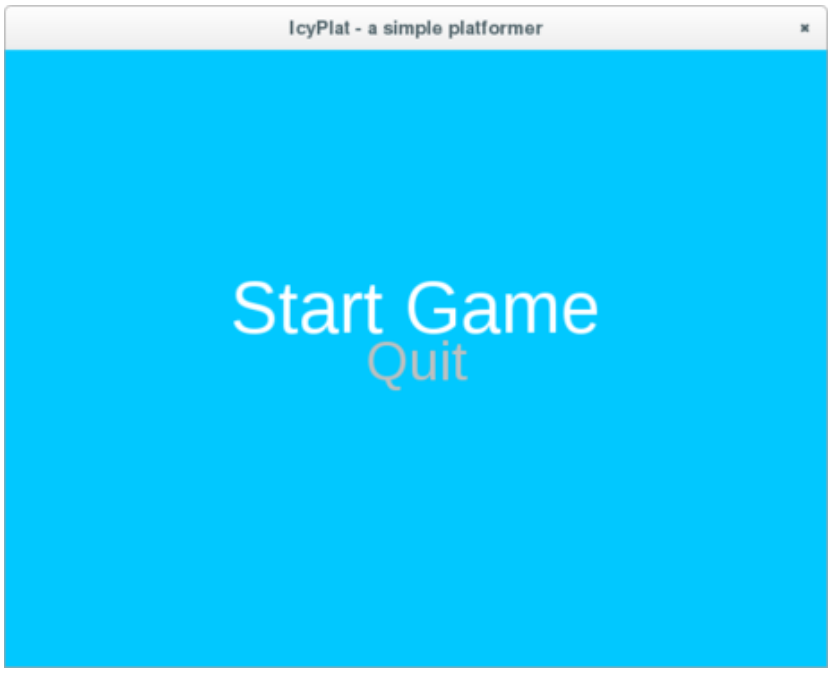
2. Define a subclass `MainMenuBgr` of the `ColorLayer`. This basically means that any main menu background you create will behave like a color layer with some changes you make.
3. Initialize the cocos director. This will give you a new window. If you don't set some caption, the window will have the same caption as the file name (main.py), which doesn't look very professional. Allow the window to be resized with by setting `resizable` to `True`.
4. Define a function `showMainMenu`. You should put the code for showing the main menu into a function because this will allow you to easily return to the main menu by calling the function again.
5. Create a scene. The scene consists of one layer for now, which is an object of the `MainMenuBgr` class you defined.
6. Run this scene in the window.

```
from cocos.director import * from cocos.scene import * from cocos.layer
 import * class MainMenuBgr(ColorLayer): def __init__(self): super(
MainMenu, self).__init__(0,200,255,255) def showMainMenu(): menuSc =
Scene(MainMenuBgr()) director.run(menuSc) director.init(caption=
"IcyPlat - a simple platformer", resizable=True) showMainMenu()
```

6. **Create a main menu.** Besides the actual game, you should have an option to close the game. You'll also add credits later, but ignore them for now. A main menu will avoid making entering the game too unexpected.
    1. Import necessary modules. You need to import `cocos.menu` (again with the `from` instruction) and `pyglet.app` (this time with `import`).
    2. Define MainMenu as a subclass of Menu.
    3. Set the alignment of the main menu. You have to set the vertical and horizontal alignment separately.
    4. Create a list of menu items and add create the actual menu. You should have the menu items "Start Game" and "Quit". Make sure to put every created menu item inside of brackets. A menu item has a label and a callback function for when it's clicked. For the "Start Game" item, use the `startGame` function (you'll write it soon), for the "Quit" item, use "pyglet.app.exit" (already exists). Create the actual menu by calling `self.create_menu(menuItems)`.
    5. Define `startGame()`. Just put `pass` into the definition for now, you'll replace that when you write the actual game.
    6. Go to the place in your code where you created the `menuSc` scene, and add a MainMenu object to it.
    7. Your entire code should now look as follows:

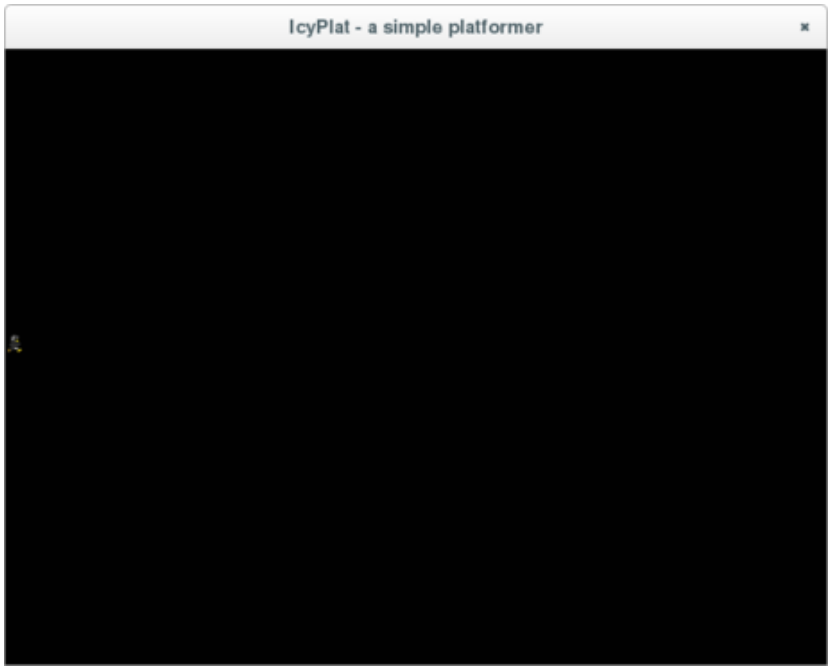
```
from cocos.director import * from cocos.menu import * from
cocos.scene import * from cocos.layer import * import pyglet.app
class MainMenuBgr(ColorLayer): def __init__(self): super(
MainMenuBgr, self).__init__(0,200,255,255) class MainMenu(Menu):
def __init__(self): super(MainMenu, self).__init__("") self.
menu_valign = CENTER self.menu_halign = CENTER menuItems = [(
MenuItem("Start Game", startGame)), (MenuItem("Quit", pyglet.app.
exit))] self.create_menu(menuItems) def startGame(): pass def
showMainMenu(): menuSc = Scene(MainMenuBgr()) menuSc.add(MainMenu
()) director.run(menuSc) director.init(caption=
```

```
"IcyPlat - a simple platformer", resizable=True) showMainMenu()
```

7. 


**Test your code.** It is important to test you code at such an early stage, while it is still short and relatively simple: this way you will know any mistakes in the basic structure, and can fix them before they cause more problems.

1. The code from the instructions should open a window, captioned "IcyPlat - a simple platformer", that you can resize and that has a light blue background. It should have a menu with two items: when you click on "Start Game", nothing happens; when you click on "Quit", the program quits.

8. 


**Display a sprite in the game.** The sprite is like a "game object". In a platformer, for example, one should be the main figure that the player can control (in this step, it'll only be displayed however). Things like background decorations or object the player can interact with can also be sprites, but you should only add

one first to understand the concept, then you can add whatever else you want.

1. Import the `cocos.sprite` submodule with the from-import-expression.
2. Find an image. You can't display a sprite if you don't have a picture for it. You can draw one, or you can get one from the internet (watch out for the license, though, if you're planning to publish your game), for example from here (crop the image so you only have one running penguin). Make sure to put your image into the same directory as the program.
3. Create the sprite's layer and the sprite. Create the layer as a new object of the `ScrollableLayer` class. Create the sprite as a `Sprite` object and set its position to (8, 250). For reference, the point (0, 0) is in the bottom left corner. This is quite high, but it will make sure that the penguin doesn't get stuck in the ice.
4. Add the sprite to the sprite's layer.
5. Create a new scene out of the sprite's layer and run it.

```
def startGame(): figLayer = ScrollableLayer() fig = Sprite('pingu.png')
 fig.position = (75, 100) figLayer.add(fig) # gameSc = Scene(figLayer)
director.run(gameSc)
```

1. You can run your code now if you want. You will see a small penguin figure (or whatever you drew) on a black background after you click on "Start Game".

9. **Decide what your landscape will consist of.** In most games, your sprites shouldn't just float in the void. They should actually stand on some surface, with something around them. In 2D games, this is often done with a tile set and a tile map. The tile set basically says what kind of surface squares and background squares exist, and what they look like.
    1. Create a tile set. The tile set for this game will be very basic: one tile for ice and one tile for sky. The ice tile used in this example is from here, under CC-BY-SA 3.0.
    2. Create a tile set picture. That's a picture of all tiles, which have to all be of the same size (edit them if they aren't) and have the size you want to see in the game, next to each other. Save your picture as `icyTiles.png`.
    3. Create the tile set description. That's an XML file. The XML file contains information on how big the tiles are in the tile set picture, which picture to use, and where to find which tile there. Create an XML file named `icyTiles.xml` with the code below:

```
  size="16x16" file="icyTiles.png">  id="i-ice" offset="0,0" />
id="i-sky" offset="16,0" />  id="ice"> ref="i-ice" />  id="sky">
ref="i-sky" />
```

10. **Make an actual structure out of the elements of your landscape.** If you made a tile set, this should be done in the form of a tile map. A tile map is like a map that defines which tile is at which position in your level. In the example, you should define a function to generate tile maps because designing tile maps by hand is very tedious. A more advanced game would usually have some sort of level editor, but for becoming familiar with 2D game development, an algorithm can provide good enough levels.
    1. Find out how many rows and columns are needed. For this, divide the screen size by the tile size both horizontally (columns) and vertically (rows). Round the number upwards; you need a function of the math module for that, so add `from math import ceil` to the imports at the top of your code.
    2. Open a file for writing. This will erase all previous content of the file, so choose a name that no file in the directory has yet, like `levelMap.xml`.
    3. Write the opening tags into the file.
    4. Generate a tile map according to the algorithm. You use the one in the code below, or you can come up with one on your own. Make sure to import the `randint` function from the module `random`: it's required for the code below to work, and whatever you come up with will

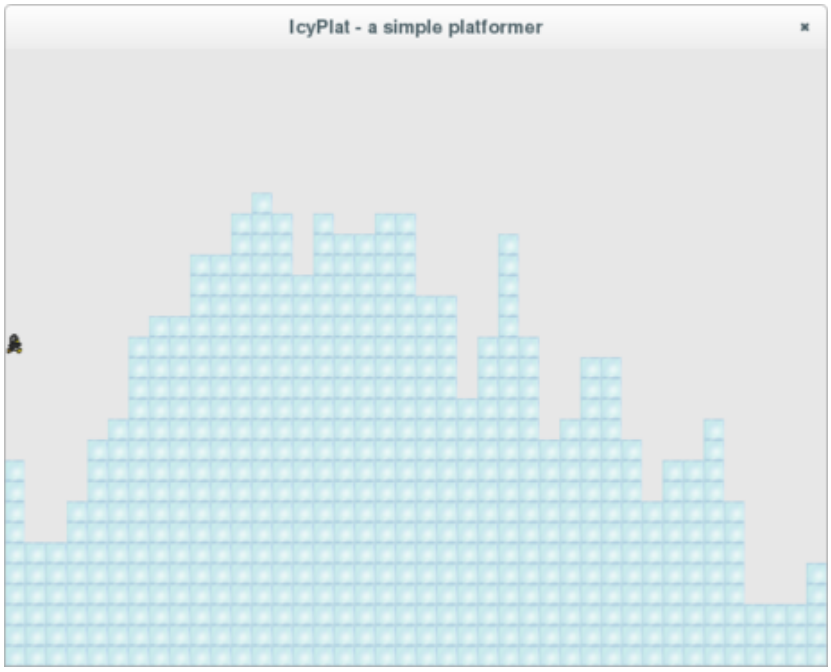
probably also need random integers. Also, make sure to put sky tiles and ice tiles in different layers: ice is solid, sky is not.

5. Write the closing tags into the file and close the file.

```
def generateTilemap(): colAmount = ceil(800 / 16)*3
# (screen width / tile size) * 3 rowAmount = ceil(600 / 16)
# screen height / tile size tileFile = open("levelMap.xml","w")
tileFile.write('nnn') iceHeight = randint(1,10) for i in range(0,
colAmount): tileFile.write('') makeHole = False if randint(0,50) == 10
and i != 0: # don't allow holes at the spawnpoint makeHole = True for j
 in range(0,rowAmount): if makeHole: tileFile.write('n') else: if j =
iceHeight: tileFile.write('n') else: tileFile.write('n') iceHeight =
randint(iceHeight-5, iceHeight+5) if iceHeight  0:
# limit tiles from going too low iceHeight = randint(1,5) if iceHeight
> rowAmount: # limit tiles from going too high iceHeight = randint(int(
rowAmount/2)-5,int(rowAmount/2)+5) tileFile.write('n') tileFile.write('
nn') for i in range(0,colAmount): tileFile.write('') for j in range(0,
rowAmount): tileFile.write('n') tileFile.write('n') tileFile.write('nn'
) tileFile.close()
```

11. **Display the tile map.** Import everything from `cocos.tiles` and then go into the `startGame` function for that.
    1. At the beginning of your `startGame` function, generate a tile map using the function you defined for that.
    2. Create a new scrolling manager. Do this directly under the line where you add the sprite to its layer.
    3. Create a new layer containing the tiles, which will be loaded from the `levelMap.xml` tile map your `generateTilemap` function generated.
    4. Add the non-solid layer, the solid layer and the sprite layer to the scrolling manager, exactly in this order. You can add a z-position if you want.
    5. Instead of creating the scene from the sprite layer, create it from the scrolling manager.
    6. Your `startGame` function should now look like this:

```
def startGame(): generateTilemap() # fig = Sprite('pingu.png') fig.
position = (8, 500) figLayer = ScrollableLayer() figLayer.add(fig)
# tileLayer = load('levelMap.xml') solidTiles = tileLayer['solid']
nsoliTiles = tileLayer['not_solid'] # scrMang = ScrollingManager()
scrMang.add(nsoliTiles,z=-1) scrMang.add(solidTiles,z=0) scrMang.
add(figLayer,z=1) # gameSc = Scene(scrMang) director.run(gameSc)
```

12. 


**Test your code.** You should test your code often to make sure that the new features you implemented really work.

1. The code in the example should now show some icy landscape behind the penguin. If the penguin looks like it is hovering far over the ice, you didn't do anything wrong, and it will be fixed in the next step.

13. **Add the controls.** The player has many more ways to interact with the program in a 2D game than in a text-based game. A common one includes moving their figure when the correct key is pressed.
    1. Import everything from `cocos.mapcolliders` and from `cocos.actions`. Also import `key` from `pyglet.window`.
    2. "Declare" some global variables. Global variables are shared between functions. You can't really declare variables in Python, but you have to say that a global variable exists in the main code before using it. You can assign 0 as the value because a function will take care of assigning the correct value later. So add under the import expressions:

```
# "declaring" global variables keyboard = 0 scrMang = 0
```

    3. Adjust your `startGame` function:
        1. Say that you use the global variables `keyboard` and `scrMang`. Do this by writing `global keyboard, scrMang` at the top of the function.
        2. Make the window listen to keyboard events.
        3. Tell the figure to act based on a `PlatformerController`. You'll implement that `PlatformerController` soon.
        4. Create a map collider to handle collisions between the solid tiles and the figure.

```
def startGame(): global keyboard, scrMang generateTilemap() # fig =
 Sprite('pingu.png') fig.position = (8, 250) figLayer =
ScrollableLayer() figLayer.add(fig) # tileLayer = load(
'levelMap.xml') solidTiles = tileLayer['solid'] nsoliTiles =
tileLayer['not_solid'] # keyboard = key.KeyStateHandler() director.
window.push_handlers(keyboard) # fig.do(PlatformerController())
mapcollider = RectMapCollider(velocity_on_bump='slide') fig.
```

```
collision_handler = make_collision_handler(mapcollider, solidTiles)
 # scrMang = ScrollingManager() scrMang.add(nsoliTiles,z=-1)
scrMang.add(solidTiles,z=0) scrMang.add(figLayer,z=1) # gameSc =
Scene(scrMang) director.run(gameSc)
```

4. Create a platformer controller. This is what will move the figure according to your keypresses.
    1. Define the platformer controller as a subclass of `Action`.
    2. Define the move speed, the jump speed and the gravity.
    3. Define the `start` function. This function is called once, when the platformer controller is connected to the figure. It should set its speed to 0 both in x and in y direction.
    4. Define the `step` function. It will be repeated while the scene is running.
    5. Tell the `step` function to use the global variables `keyboard` and `scrMang`.
    6. Get and change the velocity. Save the x and the y velocity in separate variables. Set the x velocity to either 1 or -1 (depending on whether the left or right key was pressed) multiplied with the move speed. Add gravity to the y velocity. Multiply it with downtime so it works the same way on slower devices. If the space key is pressed and the figure is standing on the ground, jump by changing y velocity to jump speed.
    7. Calculate to where the figure should move. Then let the collision handler adjust that position if it is inside of a solid tile. Finally, move the figure to the new adjusted position.
    8. Set the focus of the scrolling manager on the figure. This causes the camera to move in a reasonable way when the figure moves.

```
class PlatformerController(Action): global keyboard, scrMang
on_ground = True MOVE_SPEED = 300 JUMP_SPEED = 500 GRAVITY = -1200
def start(self): self.target.velocity = (0, 0) def step(self, dt):
global keyboard, scroller if dt
```