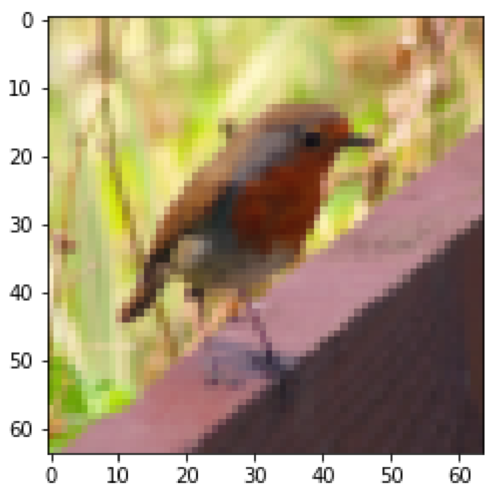
In [14]:
```
import time
import numpy as np
import h5py
import matplotlib.pyplot as plt
import scipy
from PIL import Image
from scipy import ndimage
from dnn_app_utils import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [15]:
```
train_x_orig, train_y, test_x_orig, test_y, classes = load_data()
```

In [16]:
```
# Example of a picture
index = 10
plt.imshow(train_x_orig[index])
print ("y = " + str(train_y[0, index]) + ". It's a " + classes[train_y[0
, index]].decode("utf-8") + " picture.")
```

```
y = 0. It's a non-cat picture.
```



In [17]:
```
# Explore your dataset
m_train = train_x_orig.shape[0]
num_px = train_x_orig.shape[1]
m_test = test_x_orig.shape[0]

print ("Number of training examples: " + str(m_train))
print ("Number of testing examples: " + str(m_test))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) +
", 3)")
print ("train_x_orig shape: " + str(train_x_orig.shape))
print ("train_y shape: " + str(train_y.shape))
print ("test_x_orig shape: " + str(test_x_orig.shape))
print ("test_y shape: " + str(test_y.shape))
```

```
Number of training examples: 209
Number of testing examples: 50
Each image is of size: (64, 64, 3)
train_x_orig shape: (209, 64, 64, 3)
train_y shape: (1, 209)
test_x_orig shape: (50, 64, 64, 3)
test_y shape: (1, 50)
```

In [18]:
```
# Reshape the training and test examples
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T   #
 The "-1" makes reshape flatten the remaining dimensions
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T

# Standardize data to have feature values between 0 and 1.
train_x = train_x_flatten / 255.
test_x = test_x_flatten / 255.

print ("train_x's shape: " + str(train_x.shape))
print ("test_x's shape: " + str(test_x.shape))
```

```
train_x's shape: (12288, 209)
test_x's shape: (12288, 50)
```

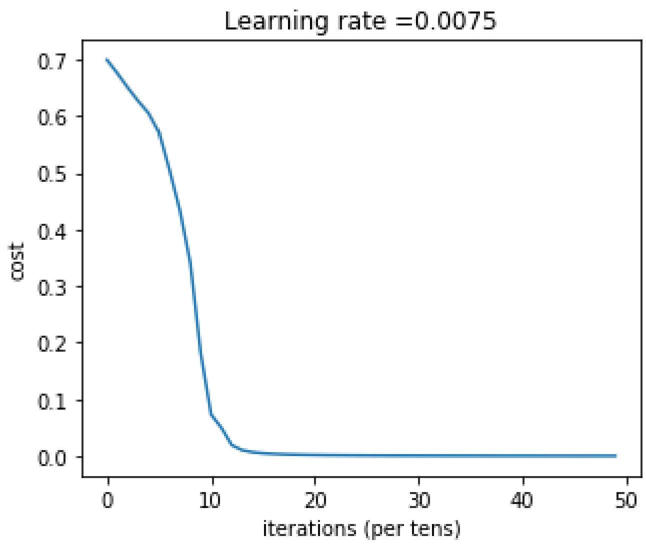
In [19]:
```
### CONSTANTS ###
layers_dims = [12288, 100, 80, 60, 40,20, 7, 5, 1] #  5-layer model
```

In [20]:
```
# GRADED FUNCTION: n_layer_model

def L_layer_model(X, Y, layers_dims, learning_rate=0.0075, num_iteration
s=3000, print_cost=False): #lr was 0.009
    """
    Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->S
IGMOID.

    Arguments:
    X -- data, numpy array of shape (number of examples, num_px * num_px
* 3)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of sha
pe (1, number of examples)
    layers_dims -- list containing the input size and each layer size, o
f length (number of layers + 1).
    learning_rate -- learning rate of the gradient descent update rule
    num_iterations -- number of iterations of the optimization loop
    print_cost -- if True, it prints the cost every 100 steps

    Returns:
    parameters -- parameters learnt by the model. They can then be used
 to predict.
    """

    np.random.seed(1)
    costs = []                         # keep track of cost

    # Parameters initialization.
    ### START CODE HERE ###
    parameters = initialize_parameters_deep(layers_dims)
    ### END CODE HERE ###

    # Loop (gradient descent)
    for i in range(0, num_iterations):

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMO
ID.
        ### START CODE HERE ### (≈ 1 line of code)
        AL, caches = L_model_forward(X, parameters)
        ### END CODE HERE ###

        # Compute cost.
        ### START CODE HERE ### (≈ 1 line of code)
        cost = compute_cost(AL, Y)
        ### END CODE HERE ###

        # Backward propagation.
        ### START CODE HERE ### (≈ 1 line of code)
        grads = L_model_backward(AL, Y, caches)
        ### END CODE HERE ###

        # Update parameters.
        ### START CODE HERE ### (≈ 1 line of code)
        parameters = update_parameters(parameters, grads, learning_rate)
        ### END CODE HERE ###
```

```
        # Print the cost every 100 training example
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" % (i, cost))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

In [21]:
```
parameters = L_layer_model(train_x, train_y, layers_dims, num_iterations
=5000, print_cost=True)
```

```
Cost after iteration 0: 0.699376
Cost after iteration 100: 0.675820
Cost after iteration 200: 0.650465
Cost after iteration 300: 0.626926
Cost after iteration 400: 0.605531
Cost after iteration 500: 0.571469
Cost after iteration 600: 0.504836
Cost after iteration 700: 0.433403
Cost after iteration 800: 0.341125
Cost after iteration 900: 0.181984
Cost after iteration 1000: 0.073775
Cost after iteration 1100: 0.050149
Cost after iteration 1200: 0.019582
Cost after iteration 1300: 0.010558
Cost after iteration 1400: 0.006912
Cost after iteration 1500: 0.005062
Cost after iteration 1600: 0.003936
Cost after iteration 1700: 0.003193
Cost after iteration 1800: 0.002671
Cost after iteration 1900: 0.002285
Cost after iteration 2000: 0.001991
Cost after iteration 2100: 0.001761
Cost after iteration 2200: 0.001576
Cost after iteration 2300: 0.001425
Cost after iteration 2400: 0.001299
Cost after iteration 2500: 0.001192
Cost after iteration 2600: 0.001100
Cost after iteration 2700: 0.001020
Cost after iteration 2800: 0.000950
Cost after iteration 2900: 0.000889
Cost after iteration 3000: 0.000834
Cost after iteration 3100: 0.000785
Cost after iteration 3200: 0.000742
Cost after iteration 3300: 0.000702
Cost after iteration 3400: 0.000666
Cost after iteration 3500: 0.000634
Cost after iteration 3600: 0.000604
Cost after iteration 3700: 0.000577
Cost after iteration 3800: 0.000551
Cost after iteration 3900: 0.000528
Cost after iteration 4000: 0.000507
Cost after iteration 4100: 0.000487
Cost after iteration 4200: 0.000468
Cost after iteration 4300: 0.000451
Cost after iteration 4400: 0.000435
Cost after iteration 4500: 0.000420
Cost after iteration 4600: 0.000405
Cost after iteration 4700: 0.000392
Cost after iteration 4800: 0.000379
Cost after iteration 4900: 0.000368
```




In [22]:
```
pred_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 1.0
```

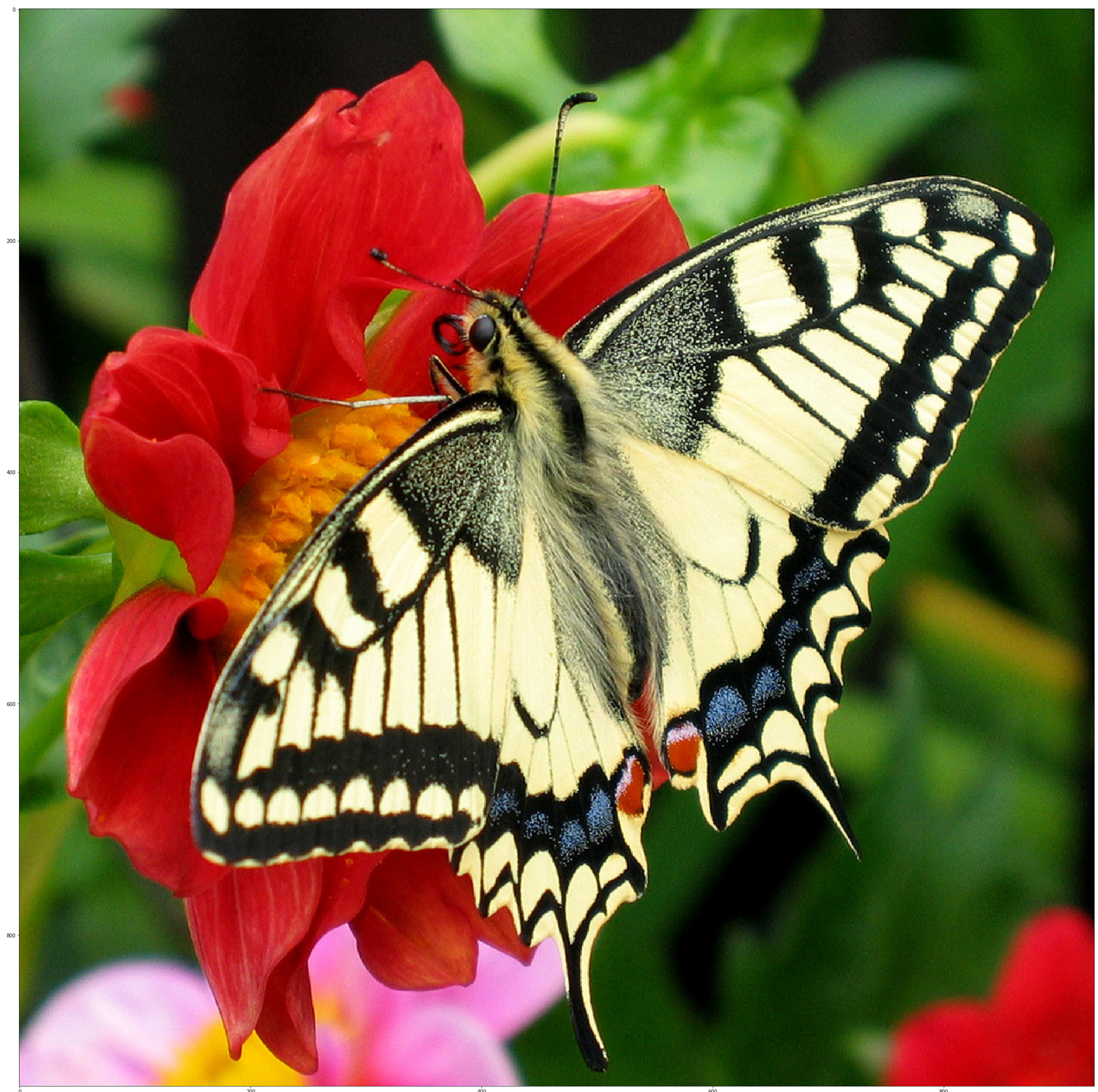
In [23]:
```
pred_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.82
```

In [24]:
```
print_mislabeled_images(classes, test_x, test_y, pred_test)
```

In [29]:
```
## START CODE HERE ##
my_image = "butterfly.jpg" # change this to the name of your image file
my_label_y = [1] # the true class of your image (1 -> cat, 0 -> non-cat)
## END CODE HERE ##

fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
my_image = scipy.misc.imresize(image, size=(num_px,num_px)).reshape((num
_px*num_px*3,1))
my_predicted_image = predict(my_image, my_label_y, parameters)

plt.imshow(image)
print ("y = " + str(np.squeeze(my_predicted_image)) + ", your L-layer mo
del predicts a \"" + classes[int(np.squeeze(my_predicted_image)),].decod
e("utf-8") +  "\" picture.")
```

```
//anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:7: Deprec
ationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0.
Use ``matplotlib.pyplot.imread`` instead.
  import sys
//anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:8: Deprec
ationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.


Accuracy: 0.0
y = 0, your L-layer model predicts a "non-cat" picture.
```



In [ ]: