In [32]:

```
# Import dependencies
import random
import numpy as np
import torch
import torch.nn as nn


# You can find Alfredo's plotting code in plot_lib.py in this directory .
# Download it along with this assignment and keep it in the same directory.
from plot_lib import set_default, show_scatterplot#, plot_bases

from matplotlib.pyplot import plot, title, axis
```

In [33]:

```
# Set up your device
cuda = torch.cuda.is_available()
device = torch.device("cuda:0" if cuda else "cpu")
```

In [34]:

```
# Set up random seed to 1008. Do not change the random seed.
# Yes, these are all necessary when you run experiments!
seed = 1008
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if cuda:
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
```

1. Full, slice, fill

Write a function `warm_up` that returns the 2D tensor with integers below. **Do not use any loops**.

```
1 2 1 1 1 1 2 1 1 1 1 2 1
2 2 2 2 2 2 2 2 2 2 2 2 2
1 2 1 1 1 1 2 1 1 1 1 2 1
1 2 1 3 3 1 2 1 3 3 1 2 1
1 2 1 3 3 1 2 1 3 3 1 2 1
1 2 1 1 1 1 2 1 1 1 1 2 1
2 2 2 2 2 2 2 2 2 2 2 2 2
1 2 1 1 1 1 2 1 1 1 1 2 1
1 2 1 3 3 1 2 1 3 3 1 2 1
1 2 1 3 3 1 2 1 3 3 1 2 1
1 2 1 1 1 1 2 1 1 1 1 2 1
2 2 2 2 2 2 2 2 2 2 2 2 2
1 2 1 1 1 1 2 1 1 1 1 2 1
```

Hint: Use `torch.full` , `torch.fill_` , and the slicing operator.

In [35]:

```
def warm_up():
#    raise NotImplementedError()
    W = torch.full((13,13),1)
    torch.fill_(W[:,1],2)
    torch.fill_(W[:,6],2)
    torch.fill_(W[:,11],2)
    torch.fill_(W[1,:],2)
    torch.fill_(W[6,:],2)
    torch.fill_(W[11,:],2)
    torch.fill_(W[3:5,3:5],3)
    torch.fill_(W[3:5,8:10],3)
    torch.fill_(W[8:10,3:5],3)
    torch.fill_(W[8:10,8:10],3)
    return(W)
# Uncomment line below once you implement this function.
print(warm_up())
```

```
tensor([[1., 2., 1., 1., 1., 1., 2., 1., 1., 1., 1., 2., 1.],
        [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
        [1., 2., 1., 1., 1., 1., 2., 1., 1., 1., 1., 2., 1.],
        [1., 2., 1., 3., 3., 1., 2., 1., 3., 3., 1., 2., 1.],
        [1., 2., 1., 3., 3., 1., 2., 1., 3., 3., 1., 2., 1.],
        [1., 2., 1., 1., 1., 1., 2., 1., 1., 1., 1., 2., 1.],
        [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
        [1., 2., 1., 1., 1., 1., 2., 1., 1., 1., 1., 2., 1.],
        [1., 2., 1., 3., 3., 1., 2., 1., 3., 3., 1., 2., 1.],
        [1., 2., 1., 3., 3., 1., 2., 1., 3., 3., 1., 2., 1.],
        [1., 2., 1., 1., 1., 1., 2., 1., 1., 1., 1., 2., 1.],
        [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
        [1., 2., 1., 1., 1., 1., 2., 1., 1., 1., 1., 2., 1.]])
```

2. To Loop or not to loop

The motivation for the following three sub-questions is to get you to think critically about how to write your deep learning code. These sorts of choices can make the difference between tractable and intractable model training.

2.1. `mul_row_loop`

Write a function `mul_row_loop` , using python loops with simple indexing but no advanced indexing/slicing, that receives a 2D tensor as input and returns a tensor of same size that is

- equal to the input on the first row
- 2 times the input's second row on the second row
- 3 times the input's third row on the third row
- etc..

For instance:

```
>>> t = torch.full((4, 8), 2.0)
>>> t
tensor([[2., 2., 2., 2., 2., 2., 2., 2.],
[2., 2., 2., 2., 2., 2., 2., 2.],
[2., 2., 2., 2., 2., 2., 2., 2.],
[2., 2., 2., 2., 2., 2., 2., 2.]])
>>> mul_row(t)
tensor([[2., 2., 2., 2., 2., 2., 2., 2.],
[4., 4., 4., 4., 4., 4., 4., 4.],
[6., 6., 6., 6., 6., 6., 6., 6.],
[8., 8., 8., 8., 8., 8., 8., 8.]])
```

In [36]:

```python
def mul_row_loop(input_tensor):
#    raise NotImplementedError()
    ret = input_tensor*1
    for i in range(len(input_tensor)):
        for j in range(len(input_tensor[0,:])):
            ret[i,j] = ret[i,j]*(i+1)
    return(ret)

t = torch.full((4, 8), 2.0)
mul_row_loop(t)
```

Out[36]:

```
tensor([[2., 2., 2., 2., 2., 2., 2., 2.],
        [4., 4., 4., 4., 4., 4., 4., 4.],
        [6., 6., 6., 6., 6., 6., 6., 6.],
        [8., 8., 8., 8., 8., 8., 8., 8.]])
```

2.2. `mul_row_fast`

Write a second version of the same function named `mul_row_fast` which uses tensor operations and no looping.

Hint: Use broadcasting and `torch.arange` , `torch.view` , and `torch.mul` .

In [37]:

```
def mul_row_fast(input_tensor):
#    raise NotImplementedError()
    T = len(input_tensor)
    other = torch.arange(1,T+1).view(T,1)
    ret = torch.mul(input_tensor,other)
    return(ret)
t = torch.full((4, 8), 2.0)
print(mul_row_fast(t))
```

```
tensor([[2., 2., 2., 2., 2., 2., 2., 2.],
        [4., 4., 4., 4., 4., 4., 4., 4.],
        [6., 6., 6., 6., 6., 6., 6., 6.],
        [8., 8., 8., 8., 8., 8., 8., 8.]])
```

2.3. `times`

Write a function `times` which takes a 2D tensor as input and returns the run times of `mul_row_loop` and `mul_row_fast` on this tensor, respectively. Use `time.perf_counter` .

Use `torch.ones` to create a 2D tensor of size (1000, 400) full of ones and run `times` on it (there should be more than two orders of magnitude difference).

In [38]:

```
from time import clock
def times(input_tensor):
#    raise NotImplementedError()
    t1 = clock()
    ret1 = mul_row_loop(input_tensor)
    t2 = clock()
    ret2 = mul_row_fast(input_tensor)
    t3 = clock()
    return t2-t1,t3-t2


# Uncomment lines below once you implement this function.
input_tensor = torch.ones(1000,400)
time_1, time_2 = times(input_tensor)
print('{}, {}'.format(time_1, time_2))
```

```
6.605997, 0.028266
```

3. Non-linearities

In this section, we explore similar concepts to Lab 1 and get comfortable initializing modules like nn.Linear and using non-linearities in PyTorch.

3.1. ReLU

ReLU (Rectified Linear Unit) is a non-linear activation fuction defined as:

$$y = \max(0, x)$$

Define a fully connected neural network `linear_fc_relu` which:

- takes 2 dimensional data as input and passes it through linear modules (`torch.nn.Linear`)
- has one hidden layer of dimension 5
- has output dimension of 2
- has ReLu as an activation function

Create a tensor with input data X of size (100, 2) using `torch.randn` .

Following the example in [https://github.com/Atcold/pytorch-Deep-Learning-Minicourse/blob/master/02-space_stretching.ipynb (https://github.com/Atcold/pytorch-Deep-Learning-Minicourse/blob/master/02-space_stretching.ipynb)](https://github.com/Atcold/pytorch-Deep-Learning-Minicourse/blob/master/02-space_stretching.ipynb), visualize the output of passing `X` through the neural network `linear_fc_relu` .

You can find Alfredo's plotting code in plot_lib.py in this directory. Download it along with this assignment and keep it in the same directory.

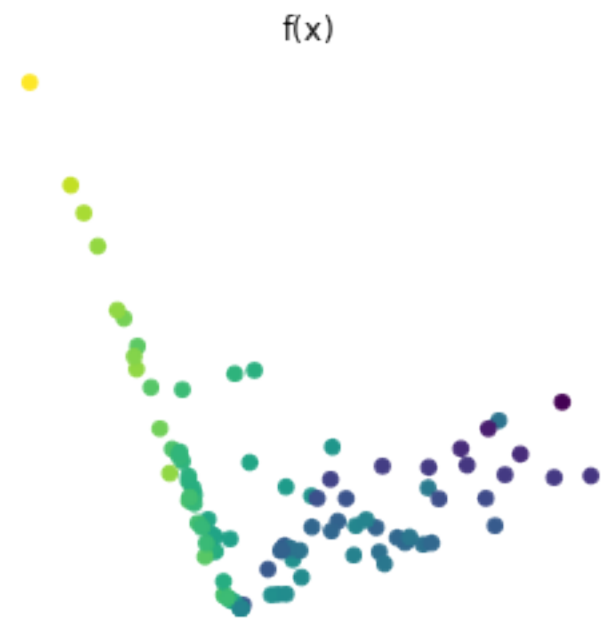
In [39]:

```
# Input data
X = torch.randn(100,2)
```

In [40]:

```
# create 1-layer neural networks with ReLU activation
colors = X[:, 0]

model = nn.Sequential(
    nn.Linear(2, 5, bias=False),
    nn.ReLU(),
    nn.Linear(5,2)
)
# linear_fc_relu = TODO
with torch.no_grad():
    linear_fc_relu = model(X)
# Visualize: TODO
show_scatterplot(linear_fc_relu, colors, title = 'f(x)')
```



3.2. Sigmoid

The sigmoid function is another popular choice for a non-linear activation function which maps its input to values in the interval $(0, 1)$. It is formally defined as:

$$\sigma(x) = \frac{1}{1 + exp[-x]}$$

Define a new neural network `linear_fc_sigmoid` which is the same architecture as in part `3.1.` but with a sigmoid unit instead of ReLU.

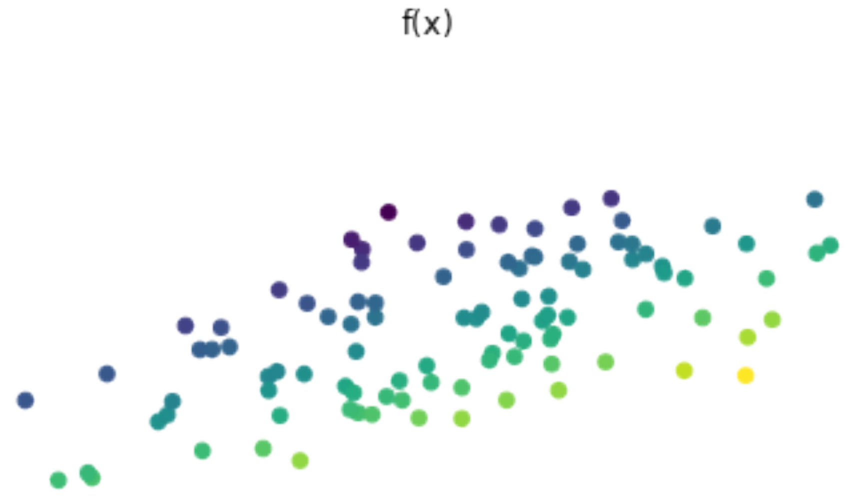
Using the same X as in part `3.1` , visualize the output of passing `X` through the neural network `linear_fc_sigmoid` .

In [41]:

```
# create 1-layer neural networks with Sigmoid activation


model_sig = nn.Sequential(
    nn.Linear(2, 5, bias=False),
    nn.Sigmoid(),
    nn.Linear(5,2)
)

# linear_fc_sigmoid = TODO
with torch.no_grad():
    linear_fc_sigmoid = model_sig(X)
# Visualize: TODO
show_scatterplot(linear_fc_sigmoid, colors, title = 'f(x)')
```



In [ ]:

```

```