1. Provide example code and necessary elaborations for demonstrating the advantages of Dynamic Scoping in using Perl to implement the simplified Monopoly game as compared to the corresponding codes in Python.

   Below are the examples of `upgradeLand` and `payDue` function in respective languages.

```perl
my $self = shift;
my $land_level = shift;
local $Player::due = 0;
if($main::cur_player -> {money} < ($upgrade_fee[$land_level] * $hfr)){
  print "You do not have enough money to upgrade the land!";
}else{
  local $Player::due = $upgrade_fee[$land_level];
  local $Player::handling_fee_rate = $hfr;
  $self -> {level} += 1;
}
$main::cur_player->payDue();
```

Listing 1: Perl version of `upgradeLand`

```perl
my $self = shift;
$self->{money} += $income * (1 - $tax_rate);
$self->{money} -= $due * (1 + $handling_fee_rate);
```

Listing 2: Perl version of `payDue`

```python
Player.income = 0
Player.tax_rate = 0
Player.due = 0
Player.handling_fee_rate = 0
land_level = self.level
if(cur_player.money < self.upgrade_fee[land_level]):
  print("You do not have enough money to upgrade the land!")
else:
  Player.income = 0
  Player.tax_rate = 0
  Player.due = self.upgrade_fee[land_level]
  Player.handling_fee_rate = 0.1
  self.level += 1
cur_player.payDue()
```

Listing 3: Python version of `upgradeLand`

```python
self.money += Player.income * (1 - Player.tax_rate)
self.money -= Player.due * (1 + Player.handling_fee_rate)
```

Listing 4: Python version of `payDue`

As you can see, in the Python version of `upgradeLand`, we have to set the variables needed for `payDue` every time, as python does not support dynamic scoping, meaning the variable will find its declaration in increasingly enclosing scopes, and `payDue` is in the `Player` class, we have to change the variables in the `Player` class, such that we can calculate the appropriate value.

On the other hand, in the Perl version of `upgradeLand`, we only masked `due` and `handling_fee_rate`, this is because Perl is a dynamically scoped language, meaning the variable will find its declaration in calling sequences, in this case, `income` is equal to 0, which does not affect our calculation in `payDue`, therefore we don't need to reinitialize all the variables.

In this case, dynamic scoping can help us calculate the correct result using the appropriate value without affecting the original case.

2. Discuss the keyword local in Perl (e.g. its origin, its role in Perl, and real practical applications of it) and giving your own opinions.

   – `local`'s origin

   Before Perl 5(the version used in this assignment), dynamic scoping was the only scoping method in Perl. In Perl 5, the creators introduced `my` to enable the static scoping ability of Perl, and `local` is still used for dynamic scoping.

   – `local`'s role in Perl

   `local` is used to make a variable used dynamic scoping instead of static scoping. As dynamic scoping's variable will find its declaration in calling sequences, meaning if we declare the variable with the same name inside the calling sequence, we will not affect the original value of the variable.

   – `local`'s real practical applications

   In the monopoly game, `local` happened many times, as a lot ot the operation need to use `payDue` function to manipulate `money` variable of the player, using `local`, we can use `payDue` function with `local`-ed variable without messing the original case(i.e. Fixed cost for every round).

   – My opinion

   As a person which used language only support static scoping, use of `local` is somewhat confusing, retrieving the value of the variable by the function who called it. But after using `local`, I have to admit that `local` is useful, when writing the Python version of monopoly, I have to check and make sure that all variable needed for `payDue` had set properly, but when writing the Perl version of the program, as sometimes we need to only to mask some value, it make the program more flexible. But in the end, do I hate using `local`(Perl)? Not really. Do I want to start using `local`(Perl) from now on? Not really, too.