Search Algorithms: Object-Oriented Implementation (Part E)

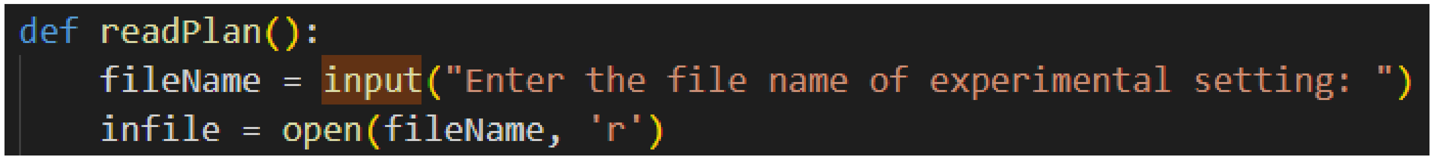
Contents

Adding More Algorithms and Classes

- (TODO) Add 3 new Search Algorithms to the existing code:
  - random-restart
  - stochastic hill climbing
  - simulated annealing
- (TODO) Also, modify the class structure for expandability/extensibility
  - To easily add various types of algorithms
- (TODO) Add configuration files
  - Use a configuration file where the problem to be solved, the solution algorithm, and various parameters are predefined
  - As a result, the code that interacts with the user was modified, and logic for parsing and processing the configuration file was added

The New Main Program

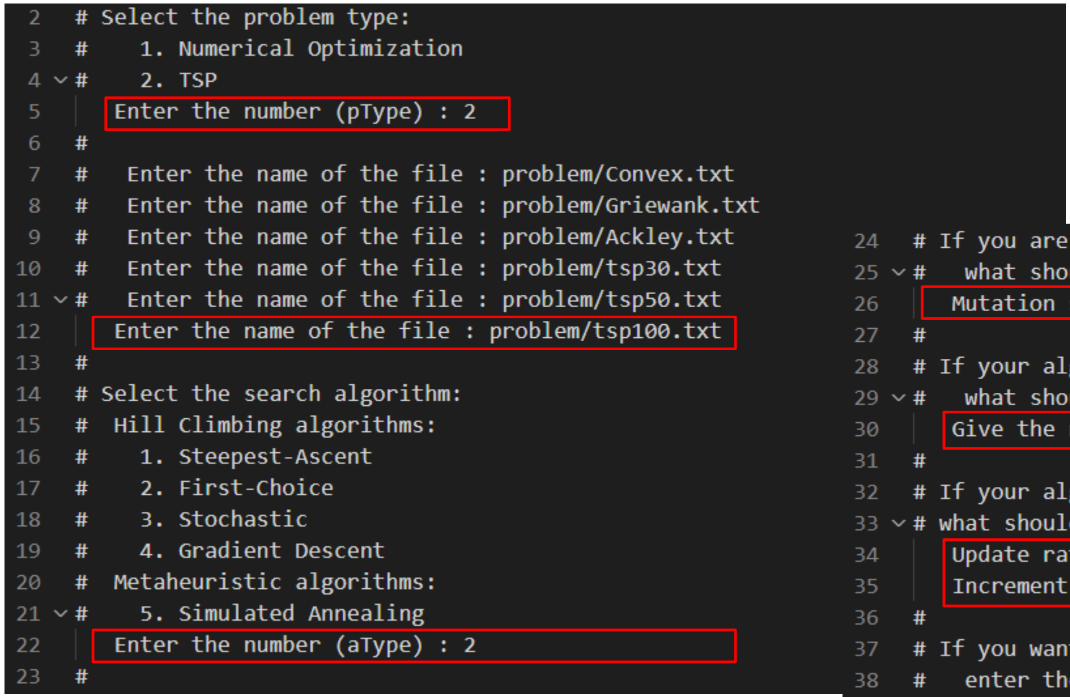
- The main program is changed significantly to meet the new requirements
  - The program should be able to support multiple experiments requested by the user
    - **Existing Program**: Returns the calculated solution by executing the user-selected algorithm once (Note: Numerous iterative calculations are performed during this single execution)
    - **New Requirement**: The program will be modified to execute the user-selected algorithm N times (numExp) and return the **best solution** (bestSolution) among them!
  - Experimental settings and other information should be read from a setup file provided by the user
    - Existing Program: Receives 3 pieces of information from the user via a terminal-based user interface: (1) Problem Type (Numeric/TSP), (2) Solution Algorithm (SA, FC, GD), (3) Filename where the problem is defined
    - New Requirement: To receive the 3 pieces of information above and various parameter values (delta, limitStuck, dx, numRestart, limitEval, numExp) from the user, the program will receive only the name of the setup file (where all information is pre-entered) and process the contents of the setup file to set the various parameters!



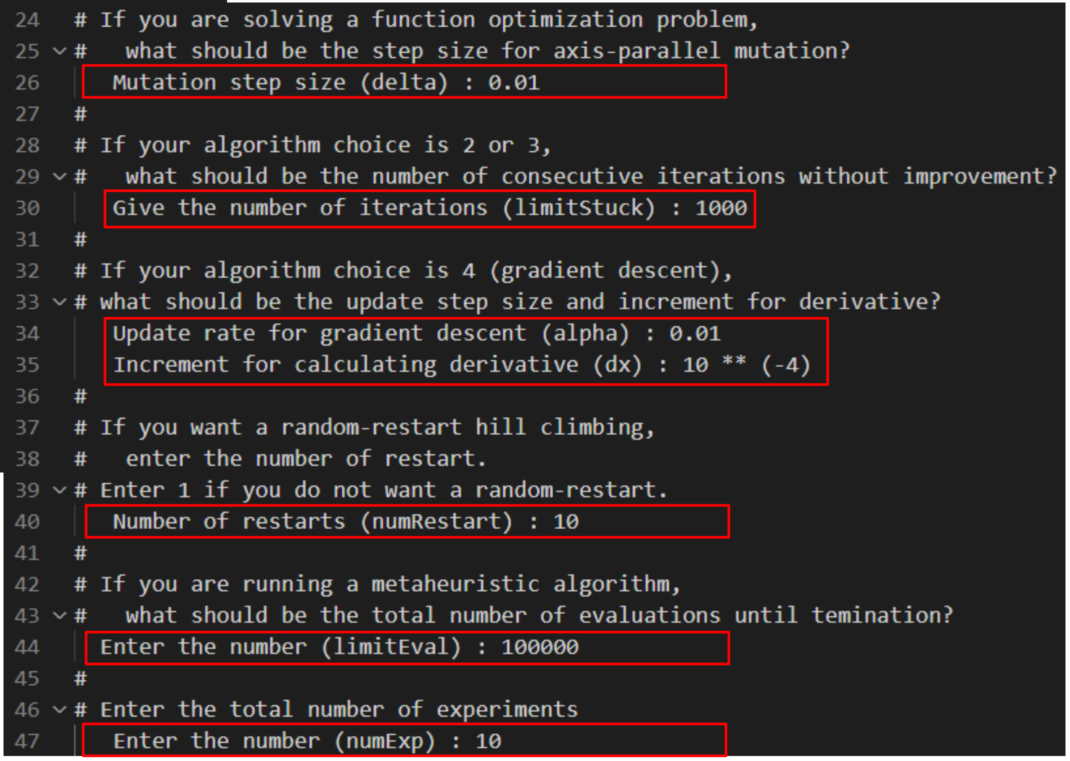
```python
def readPlan():
    fileName = input("Enter the file name of experimental setting: ")
    infile = open(fileName, 'r')
```

Adding More Algorithms and Classes

- Modified User Interface and setup file:
  - Modified User Interface: The user only enters the setup file name in the terminal
  - Text File Example (Note: Lines starting with # are ignored):



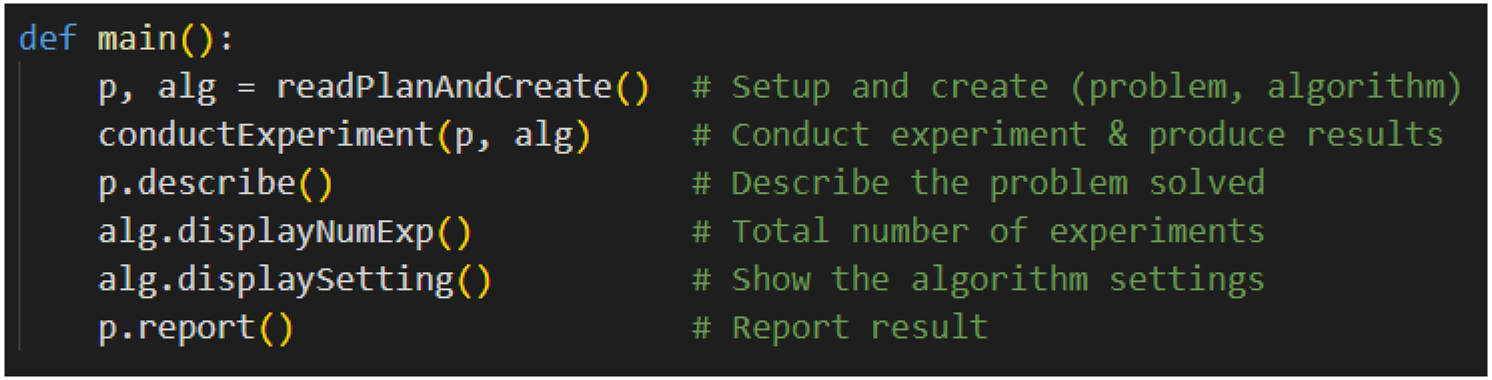
```
# Select the problem type:
#    1. Numerical Optimization
#    2. TSP
   Enter the number (pType) : 2
#
#   Enter the name of the file : problem/Convex.txt
#   Enter the name of the file : problem/Griewank.txt
#   Enter the name of the file : problem/Ackley.txt
#   Enter the name of the file : problem/tsp30.txt
#   Enter the name of the file : problem/tsp50.txt
   Enter the name of the file : problem/tsp100.txt
#
# Select the search algorithm:
#  Hill Climbing algorithms:
#    1. Steepest-Ascent
#    2. First-Choice
#    3. Stochastic
#    4. Gradient Descent
#  Metaheuristic algorithms:
#    5. Simulated Annealing
   Enter the number (aType) : 2
#
```



```
# If you are solving a function optimization problem,
#   what should be the step size for axis-parallel mutation?
   Mutation step size (delta) : 0.01
#
# If your algorithm choice is 2 or 3,
#   what should be the number of consecutive iterations without improvement?
   Give the number of iterations (limitStuck) : 1000
#
# If your algorithm choice is 4 (gradient descent),
# what should be the update step size and increment for derivative?
   Update rate for gradient descent (alpha) : 0.01
   Increment for calculating derivative (dx) : 10 ** (-4)
#
# If you want a random-restart hill climbing,
#   enter the number of restart.
# Enter 1 if you do not want a random-restart.
   Number of restarts (numRestart) : 10
#
# If you are running a metaheuristic algorithm,
#   what should be the total number of evaluations until temination?
  Enter the number (limitEval) : 100000
#
# Enter the total number of experiments
   Enter the number (numExp) : 10
```

The New Main Program

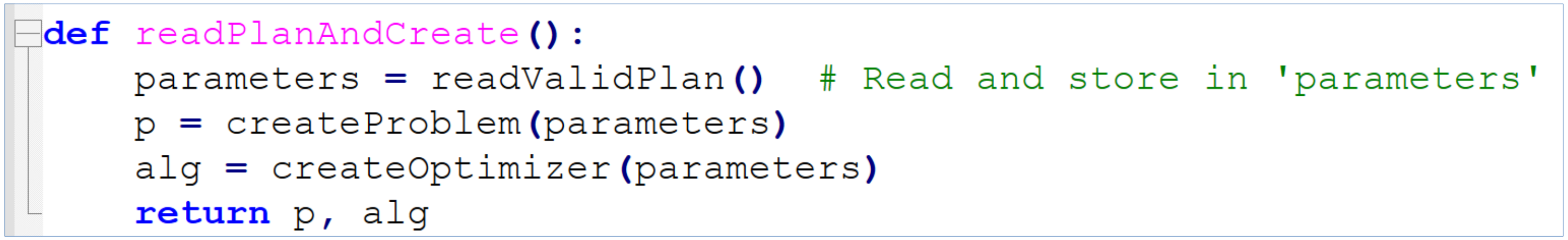
- **`main()`**:
  - (**`readPlanAndCreate`**) Reads information from a setup file and
    - creates a problem **`p`** (**`Problem`** object) to be solved, and
    - creates an optimizer **`alg`** (**`Optimizer`** object) to be used
  - Conducts experiments and obtains the result (**`conductExperiment`**)
  - Describes the problem just solved (**`p.describe`**)
  - Shows the settings of experiment (**`alg.displayNumExp`** and **`alg.displaySetting`**)
  - Reports the result of experiment (**`p.report`**)



```
def main():
    p, alg = readPlanAndCreate()  # Setup and create (problem, algorithm)
    conductExperiment(p, alg)     # Conduct experiment & produce results
    p.describe()                  # Describe the problem solved
    alg.displayNumExp()           # Total number of experiments
    alg.displaySetting()          # Show the algorithm settings
    p.report()                    # Report result
```

The New Main Program

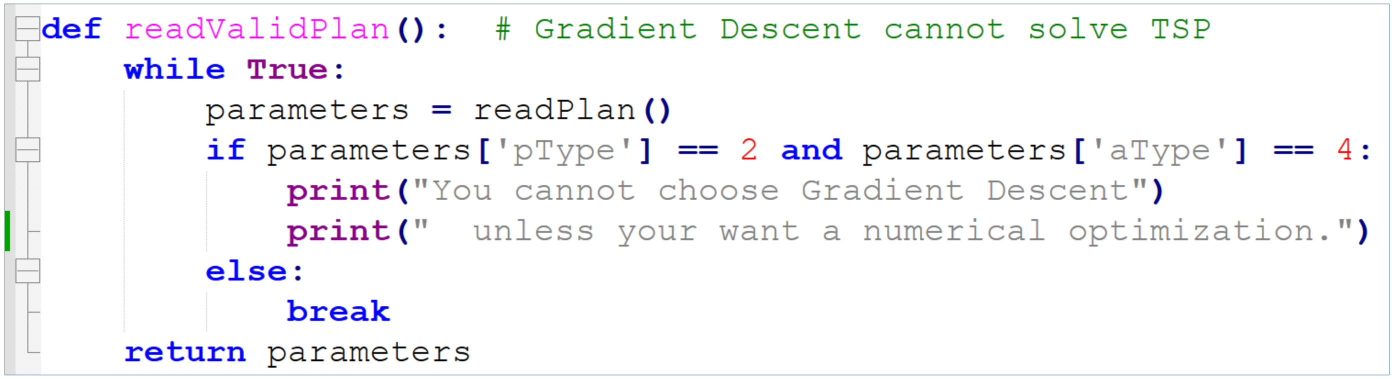
- **`readPlanAndCreate`**():
  - Reads setup information from a file and stores them in the dictionary **`parameters`** (**`readValidPlan`**)
  - Creates **`Problem`** object and store it in **`p`** (**`createProblem`**)
  - Creates **`Optimizer`** object and store it in **`alg`** (**`createOptimizer`**)
  - Returns **`p`** and **`alg`**



```
def readPlanAndCreate():
    parameters = readValidPlan()   # Read and store in 'parameters'
    p = createProblem(parameters)
    alg = createOptimizer(parameters)
    return p, alg
```

The New Main Program

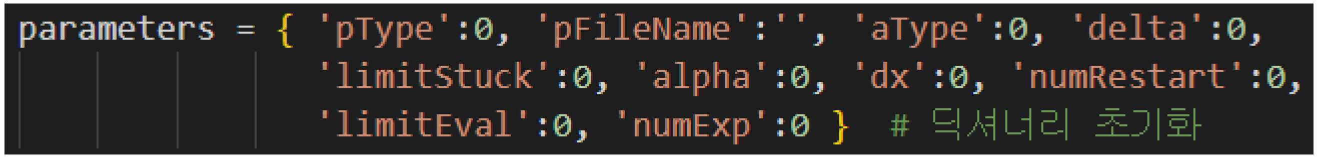
- **`readValidPlan()`**:
  - Reads setup information from a file and stores them in the dictionary **`parameters`** (**`readPlan`**)
  - Keeps querying the user if gradient descent is chosen for TSP
    - If the Gradient Descent solution is selected for the TSP problem (Error!), repeatedly call the **`readPlan`** function to receive a different setup file.
      - ✓ TSP probem : parameters[‘pType’] = 2
      - ✓ Grad Desc. algorithm : parameters[‘aType’] = 4
  - Returns **`parameters`**



```python
def readValidPlan():  # Gradient Descent cannot solve TSP
    while True:
        parameters = readPlan()
        if parameters['pType'] == 2 and parameters['aType'] == 4:
            print("You cannot choose Gradient Descent")
            print("  unless your want a numerical optimization.")
        else:
            break
    return parameters
```

The New Main Program

- **`readPlan()`**:
  - Obtains a file name from the user # setup 파일의 이름을 입력 받기
  - Prepares a dictionary variable **`parameters`** to store the information

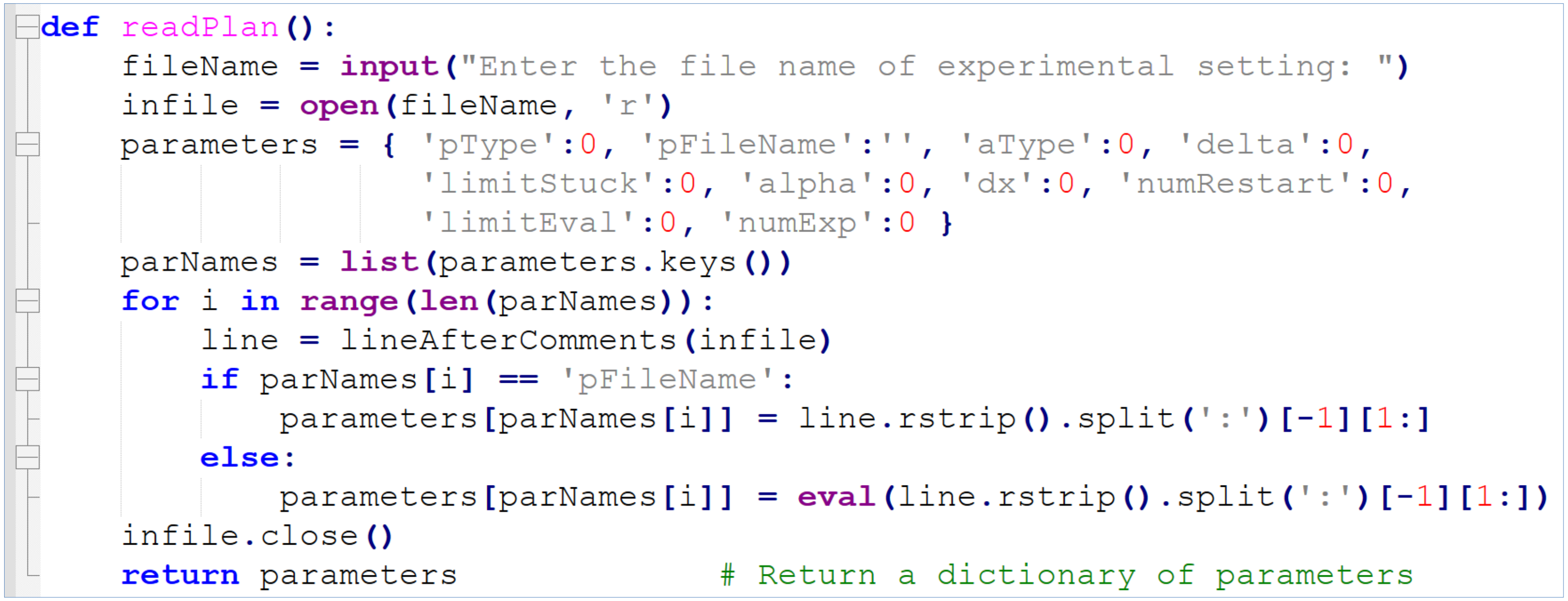


```
parameters = { 'pType':0, 'pFileName':'', 'aType':0, 'delta':0,
               'limitStuck':0, 'alpha':0, 'dx':0, 'numRestart':0,
               'limitEval':0, 'numExp':0 }  # 딕셔너리 초기화
```

  - Fills out **`parameters`** dictionary by reading the given setup file
    - All the information are numeric values except the name of the file containing specifics of the target problem (ex: `problem/Convex.txt`)
    - When reading the file line-by-line (**`lineAfterComments`**), lines beginning with '#' are all skipped
  - Returns **`parameters`** dictionary

The New Main Program

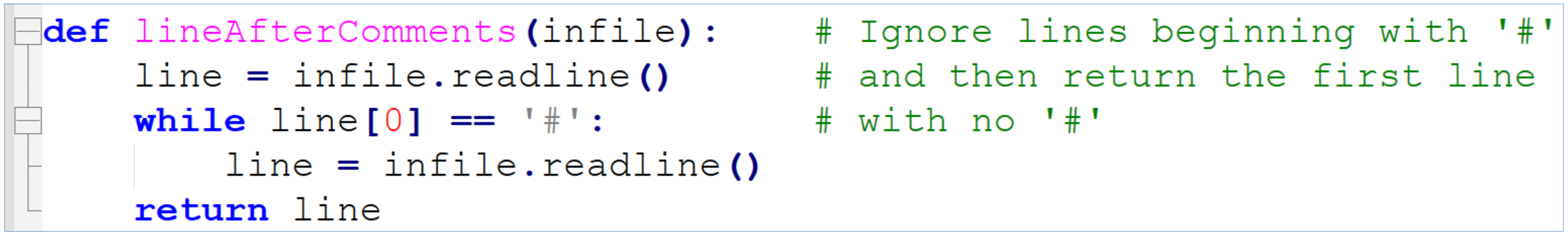
- **`readPlan()`**:



```python
def readPlan():
    fileName = input("Enter the file name of experimental setting: ")
    infile = open(fileName, 'r')
    parameters = { 'pType':0, 'pFileName':'', 'aType':0, 'delta':0,
                   'limitStuck':0, 'alpha':0, 'dx':0, 'numRestart':0,
                   'limitEval':0, 'numExp':0 }
    parNames = list(parameters.keys())
    for i in range(len(parNames)):
        line = lineAfterComments(infile)
        if parNames[i] == 'pFileName':
            parameters[parNames[i]] = line.rstrip().split(':')[-1][1:]
        else:
            parameters[parNames[i]] = eval(line.rstrip().split(':')[-1][1:])
    infile.close()
    return parameters             # Return a dictionary of parameters
```

The New Main Program

- **`lineAfterComments()`**:
  - Skips the lines beginning with the symbol ‘#’
  - Returns the line that doesn’t begin with ‘#’

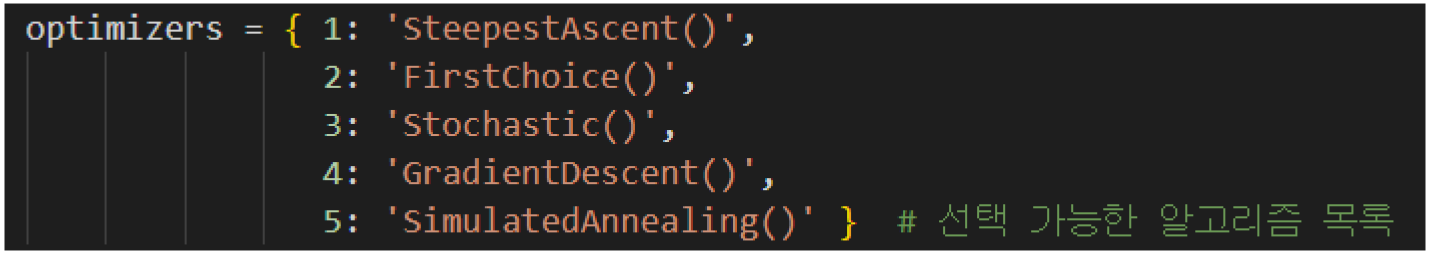


```
def lineAfterComments(infile):     # Ignore lines beginning with '#'
    line = infile.readline()       # and then return the first line
    while line[0] == '#':          # with no '#'
        line = infile.readline()
    return line
```

- **`createProblem(parameters)`**:
  - Creates a **`Numeric`** or **`Tsp`** object and store in **`p`** depending on the type of problem chosen (i.e., `parameters['pType’])`)
  - Sets some relevant variables of **`p`** in the class hierarchy according to the values in **`parameters`** (**`p.setVariables`**)
  - Returns a specific problem instance **`p`**

The New Main Program

- **`createOptimizer(parameters)`**:
  - Prepares a dictionary **`optimizers`** of algorithm class names (**`optimizers`**) that can be indexed by **`aType`** `(= parameters['aType'])`

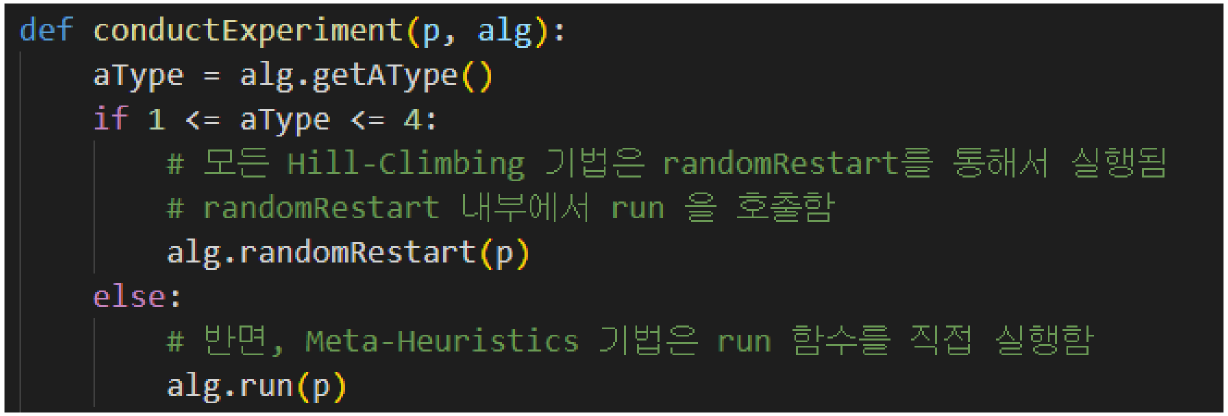


```
optimizers = { 1: 'SteepestAscent()',
               2: 'FirstChoice()',
               3: 'Stochastic()',
               4: 'GradientDescent()',
               5: 'SimulatedAnnealing()' }  # 선택 가능한 알고리즘 목록
```

  - Creates an object **`alg`** of the targeted algorithm by applying the **`eval`** function to the string of the name of the algorithm class (**`eval(optimizers[aType])`**)
  - Sets the class variables of **`alg`** with the values in **`parameters`** (**`alg.setVari-ables`**)
  - Returns **`alg`** as the created optimizer object

The New Main Program

- **`conductExperiment(p, alg)`**:
  - Solves the problem **`p`** with the chosen optimizer **`alg`** and collects the result of each individual experiment
    - If the chosen algorithm is a hill climber, then the random restart algorithm is called (**`alg.randomRestart`**)
    - Otherwise, its **`run`** method is called (**`alg.run`**)



```
def conductExperiment(p, alg):
    aType = alg.getAType()
    if 1 <= aType <= 4:
        # 모든 Hill-Climbing 기법은 randomRestart를 통해서 실행됨
        # randomRestart 내부에서 run 을 호출함
        alg.randomRestart(p)
    else:
        # 반면, Meta-Heuristics 기법은 run 함수를 직접 실행함
        alg.run(p)
```

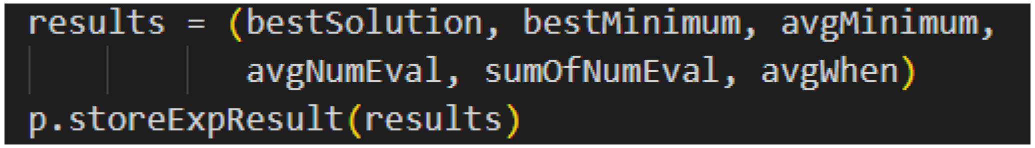
  - Repeats experiment multiple times (**`numExp = parameters[numExp]`**) if requested and collects the results
  - Finally, stores the summary of the best result (**`p.storeExpResult`**)

Changes to 'Problem' Class

- For recording the results of experiments, the **`Problem`** class is revised to have the following additional variables:
  - **`pFileName`**: name of the file containing problem specifics
  - The following 6 values are calculated in the `main.py` by `conductExperiment` 때 and are stored in the `problem` instance by calling `p.storeExpResult`
  - **`bestSolution`**: The best solution found so far
    - best solution found in *n* different experiments (`n = numExp = parameters[numExp]`)
  - **`bestMinimum`**: objective value of **`bestSolution`**
  - **`avgMinimum`**: average objective value of the best solutions obtained from *n* experiments (**`sumOfMinimum`** average)
    - **`sumOfMinimum`**: Calculated in main.py, where the objective value obtained during each iteration over **`numExp`** is called **`newMinimum`**, and **`sumOfMinimum`** += **`newMinimum`**
  - **`avgNumEval`**: average number of evaluations made in *n* different experiments
  - **`sumOfNumEval`**: total number of evaluations made all through *n* experiments
  - **`avgWhen`**:
    - average iteration when the best solution first appears in *n* experiments
    - A meaningful value only in **`MetaHeuristics`**-based algorithms
    - **`sumOfWhen`** += **`alg.getWhenBestFound`**(), and **`avgWhen`** is the average value of **`sumOfWhen`**

Changes to ‘Problem’ Class

- Accordingly, several new methods are added to handle those variables
  - Three new accessors **`getSolution`**, **`getValue`**, **`getNumEval`** are necessary for **`conductExperiment`** of the main program to conduct multiple experiments
  - The new method **`storeExpResult`** is also necessary for **`conductExperiment`** to store the result of experiment after finishing all the experiments

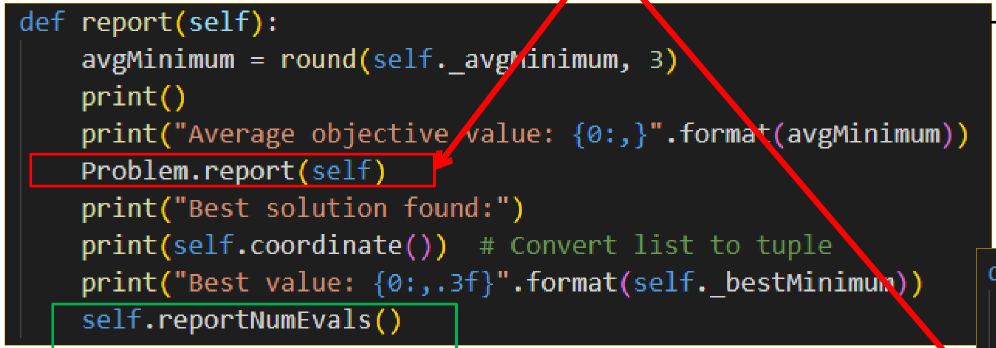


```
results = (bestSolution, bestMinimum, avgMinimum,
           avgNumEval, sumOfNumEval, avgWhen)
p.storeExpResult(results)
```

Part of `conductExperiment` function in `main.py`

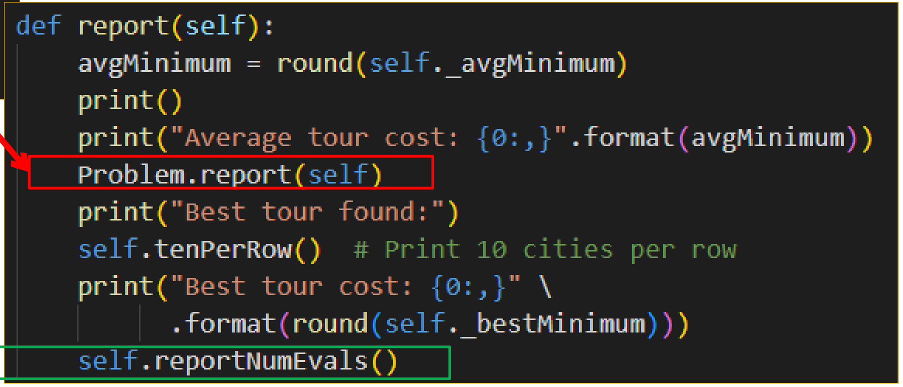
Changes to 'Problem' Class

- The **`report`** method has been revised to display the summary result of multiple experiments in an organized fashion
  - **`report`** in the base class prints messages that are common to both numerical optimization problem and TSP
  - Those in the subclasses print messages specific to the type of problem just solved
  - Therefore, the **`report`** methods in the **`Numeric`** and **`TSP`** classes call the report method of the parent class (**`Problem.report(self)`**) and then print their respective specialized information.



report method from Numeric

```python
def report(self):
    avgMinimum = round(self._avgMinimum, 3)
    print()
    print("Average objective value: {0:,}".format(avgMinimum))
    Problem.report(self)
    print("Best solution found:")
    print(self.coordinate())  # Convert list to tuple
    print("Best value: {0:,.3f}".format(self._bestMinimum))
    self.reportNumEvals()
```



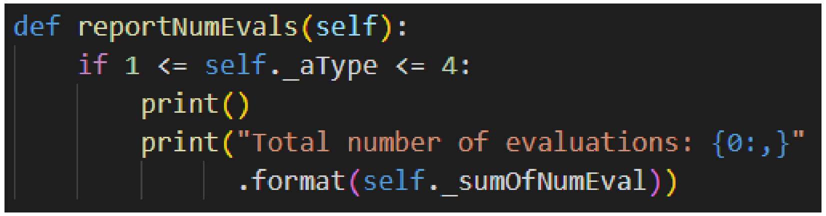
report method from TSP

```python
def report(self):
    avgMinimum = round(self._avgMinimum)
    print()
    print("Average tour cost: {0:,}".format(avgMinimum))
    Problem.report(self)
    print("Best tour found:")
    self.tenPerRow()  # Print 10 cities per row
    print("Best tour cost: {0:,}" \
          .format(round(self._bestMinimum)))
    self.reportNumEvals()
```

Refer to the next page

Changes to 'Problem' Class

- The **`reportNumEvals`** method prints out the <u>total number of evaluations</u> regardless of the problem type
  - However, it does nothing when the algorithm used is MetaHeuristics (simulated annealing or GA) because the number of evaluations for them is predetermined



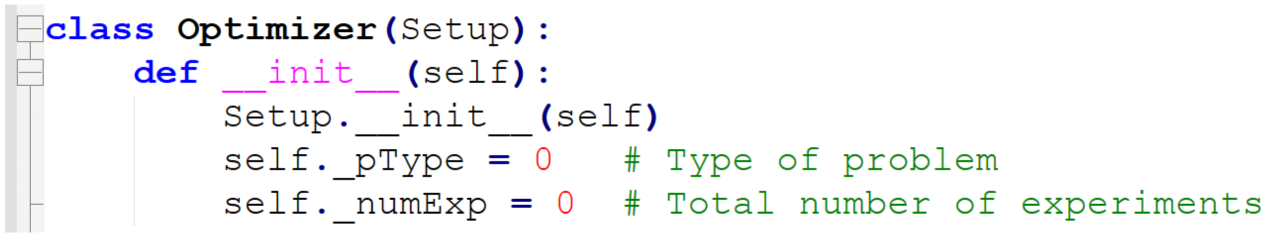
```python
def reportNumEvals(self):
    if 1 <= self._aType <= 4:
        print()
        print("Total number of evaluations: {0:,}"
              .format(self._sumOfNumEval))
```

```
1. Steepest-Ascent
2. First-Choice
3. Stochastic
4. Gradient Descent
```

  - It is separated from **`report`** because we want the result messages printed out in some appropriate order when they are mixed together with the messages generated from the subclasses
  - Call to **`reportNumEvals`** is made within **`report`** of the subclasses at its last line (Referring to the previous page)

‘Optimizer’ Class

- **`Optimizer`** has two variables **`pType`** and **`numExp`** to store common information about experimental settings:
  - The methods in the class hierarchy of **`Optimizer`** are extended versions of those previously existed in **`HillClimbing`**
  - New accessor methods **`getWhenBestFound`** (in **`MetaHeuristics`**) and **`getNumExp`** (in **`Optimizer`**) are added for being used by the **`conductExperiment`** function in the main program



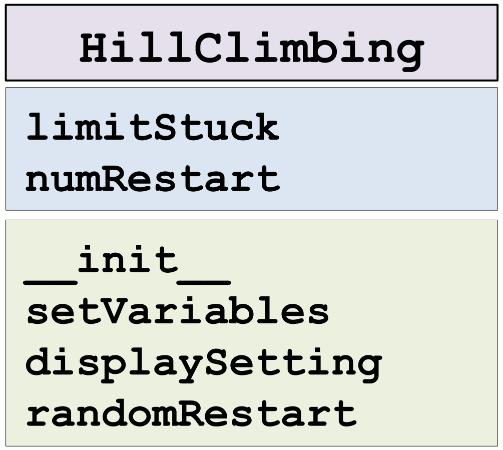
```
class Optimizer(Setup):
    def __init__(self):
        Setup.__init__(self)
        self._pType = 0    # Type of problem
        self._numExp = 0   # Total number of experiments
```

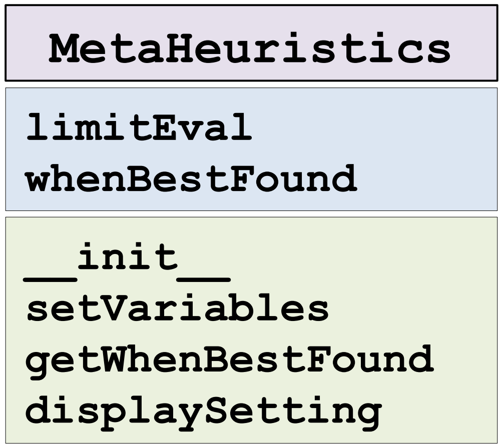
- Both ‘`random.py`’ and ‘`math.py`’ should be imported to the ‘`optimizer.py`’ file
  - the methods for stochastic hill climbing and simulated annealing algorithms need them

‘Optimizer’ Class

- **`HillClimbing`** now has only two variables: **`limitStuck`** and **`numRestart`**
  - **`pType`** has moved up to **`optimizer`**

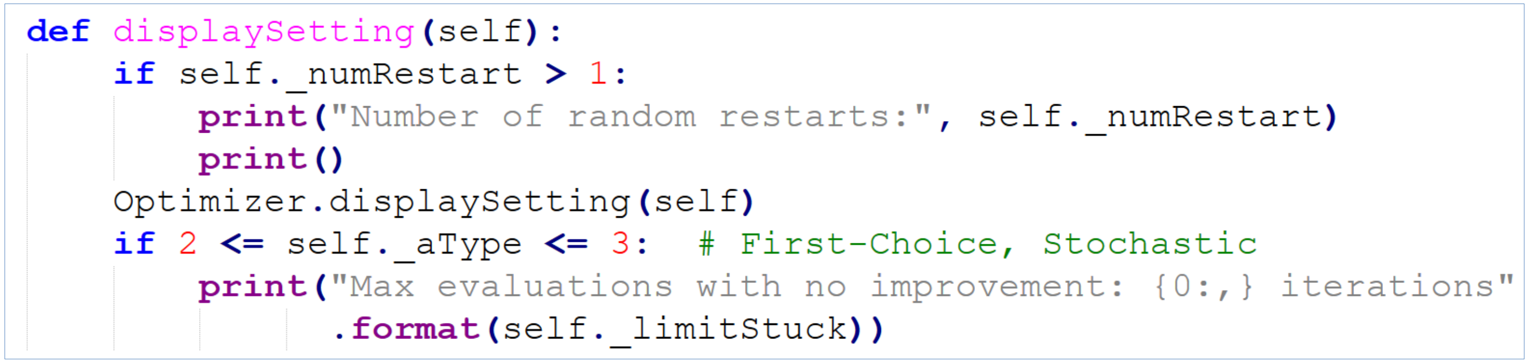
- **`MetaHeuristics`** is a parent of **`SimulatedAnnealing`** and **`GA`**
  - **`MetaHeuristics`** has two variables: **`limitEval`** and **`whenBestFound`**
  - **`displaySettings`** method prints out **`limitEval`** (total number of evaluations until termination)





Changes to ‘HillClimbing’ Class

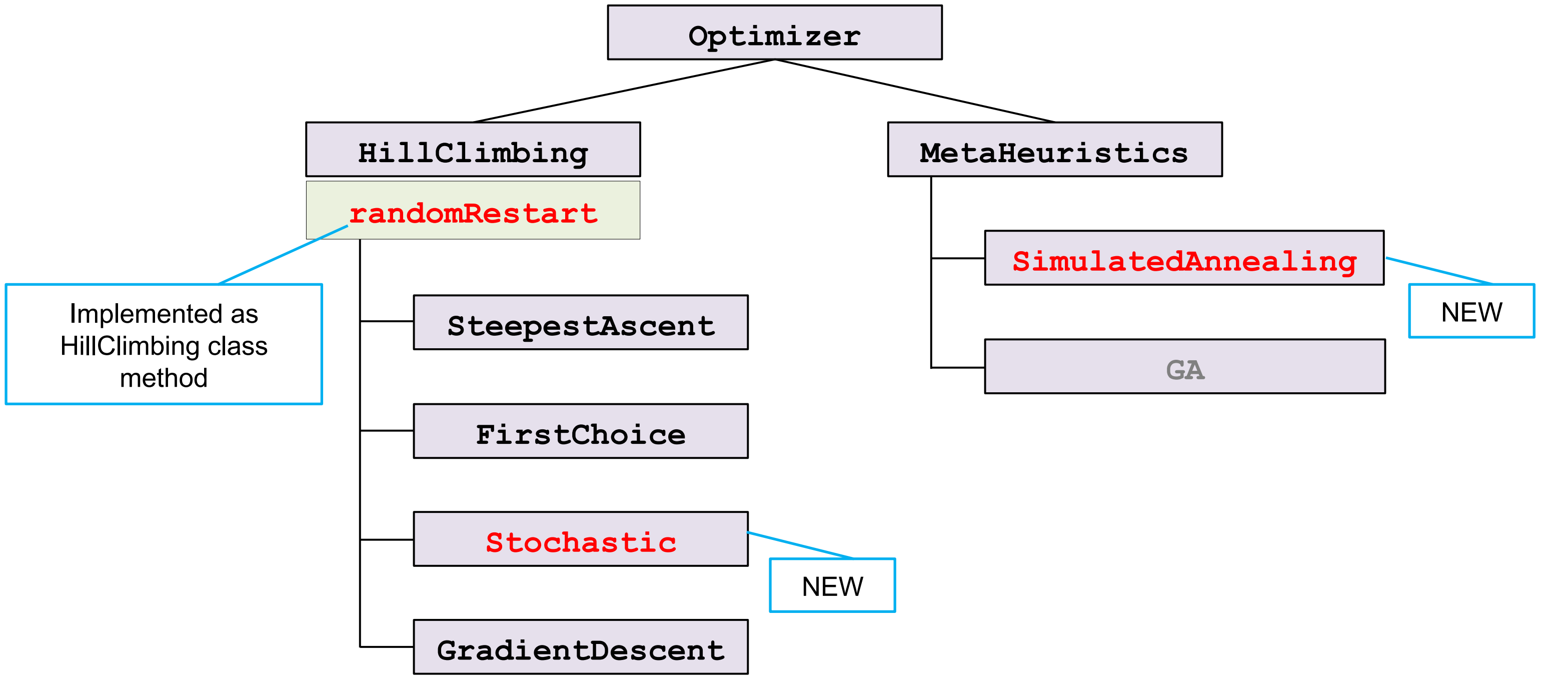
- Changes of variables:
  - Under the new **`Optimizer`** class hierarchy, the variable **`pType`** is moved up from **`HillClimbing`** to **`Optimizer`**
  - A variable **`numRestart`** is newly added (the **`setVariables`** method is revised accordingly)
- Changes to the **`displaySetting`** method:
  - The printing of mutation step size is moved up to **`Optimizer`**
  - Now it prints out setting information related only to the variables of its own class **`HillClimbing`**



```
def displaySetting(self):
    if self._numRestart > 1:
        print("Number of random restarts:", self._numRestart)
        print()
    Optimizer.displaySetting(self)
    if 2 <= self._aType <= 3:  # First-Choice, Stochastic
        print("Max evaluations with no improvement: {0:,} iterations"
              .format(self._limitStuck))
```

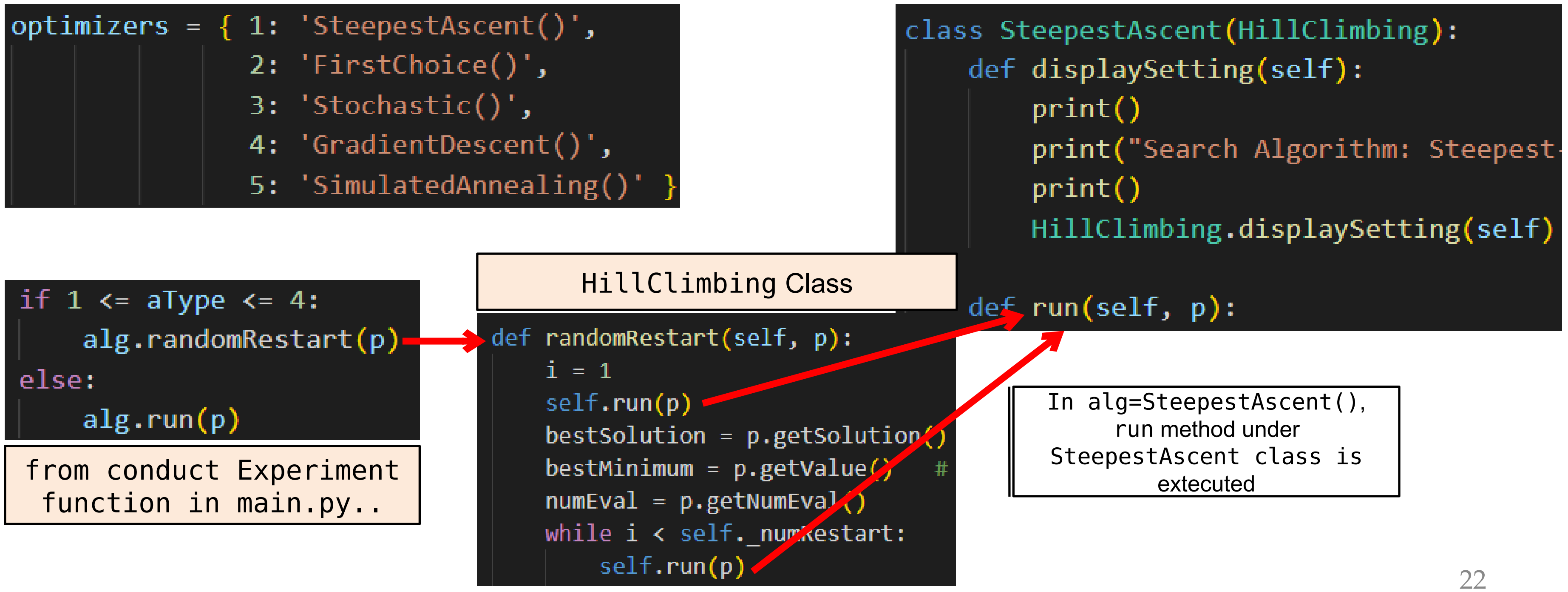
Adding three search algorithm

- (TO-DO) Add 3 new Search Algorithms to the existing code :
  - `HillClimbing > Stochastic`(stochastic hill climbing) and `randomRestart`, then `MetaHeuristics > SimulatedAnnealing`
  - Note that **`randomRestart`** is not implemented as a separate class but as a method within the **`HillClimbing`** class.



Changes to ‘HillClimbing’ Class for randomRestart

- A new method **`randomRestart`** is added to HillClimbing class
  - It keeps calling **`self.run`** for a given number of times (**`self._numRestart`**), and stores the best solution found (**`p.storeResult(bestSolution, bestMinimum)`**)
  - HillClimbing overall call sequence:
    - Note: **`MetaHeuristics`** do not use the **`randomRestart`** method



```python
optimizers = { 1: 'SteepestAscent()',
               2: 'FirstChoice()',
               3: 'Stochastic()',
               4: 'GradientDescent()',
               5: 'SimulatedAnnealing()' }
```

```python
if 1 <= aType <= 4:
    alg.randomRestart(p)
else:
    alg.run(p)
```

from conduct Experiment function in main.py..

HillClimbing Class

```python
def randomRestart(self, p):
    i = 1
    self.run(p)
    bestSolution = p.getSolution()
    bestMinimum = p.getValue()    #
    numEval = p.getNumEval()
    while i < self._numRestart:
        self.run(p)
```

```python
class SteepestAscent(HillClimbing):
    def displaySetting(self):
        print()
        print("Search Algorithm: Steepest
        print()
        HillClimbing.displaySetting(self)

    def run(self, p):
```

In alg=SteepestAscent(), run method under SteepestAscent class is extectuted

Stochastic Hill Climbing (Stochastic Class)

- The algorithm of stochastic hill climbing is the <u>same as first-choice</u> hill climbing <u>except the way a successor is chosen</u>
  - <u>First-choice</u> hill climbing generates a single successor randomly
  - <u>Stochastic hill climbing</u> generates multiple neighbors and then selects one from them at random by a probability proportional to the solution quality
  - The algorithm is implemented as the **`run`** method as before

<`SteepestDescent` `run` method>

- Generate 2N neighbor solutions and select the single best solution using the `bestOf` function,
- Compare with current solution
- Choose if better; otherwise terminate

<`First-Choice` `run` method>

- <u>Randomly generate 1 neighbor solution,</u>
- <u>compare with `current_solution`</u>
  - **Choose if better,**
  - **Otherwise, terminate if `self._limitStuck` consecutive bad solutions are encountered**

<`Stochastic` `run` method>

- Generate 2N neighbor solutions
- Select one randomly using the <u>`stochasticBest`</u>,
- <u>compare with `current_solution`</u>
  - **Choose if better,**
  - **Otherwise, terminate if `self._limitStuck` consecutive bad solutions are encountered**

CAUTION!! The actual implementation proceeds this way (like **First-Choice**, by selecting only solutions better than the current one). This is because, in this lecture, **HillClimbing** is assumed to be a category that includes techniques that only advance toward a better side...

Stochastic Hill Climbing (Stochastic Class)

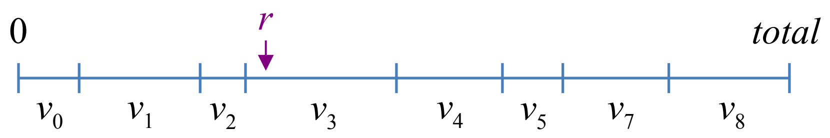
- `stochasticBest(self, neighbors, p):`
  - A function that probabilistically selects one solution from **2N neighbor** solutions
  - It does not select completely randomly, but assigns a probability proportional to the **solution quality**
  - Ex: Situation during the *k*-th iteration in a problem that aims to minimize a given expression
    - `current soln : x = ...`
    - Two neighbor solutions, *y* and *z*, were generated, and their *obj value* are as follows:
      - ✓ `f(y) = 15`
      - ✓ `f(z) = 5`
    - Since it is a minimize problem, solution *z* is better ⇒ Assign a higher selection **probability** to the better solution
      - ✓ Probability of selecting solution y : $\frac{\frac{1}{15}}{\frac{1}{15}+\frac{1}{5}} = \frac{\frac{1}{15}}{\frac{4}{15}} = \frac{1}{4}$ = 25%
      - ✓ Probability of selecting solution z : $\frac{\frac{1}{5}}{\frac{1}{15}+\frac{1}{5}} = \frac{\frac{3}{15}}{\frac{4}{15}} = \frac{3}{4}$ = 75%

Stochastic Hill Climbing (Stochastic Class)

- `stochasticBest(self, neighbors, p)`:
  - Probabilistic selection: How to select *y* with 0.25 probability and *z* with 0.75 probability??
  - [1] Use a probability value between 0 and 1 to search sequentially}
    - Generate a **random value** between [0, 1] and store it in *r*
    - If the value of *r* is less than (or equal to) 0.25, select *y*, otherwise, select *z*
  - [2] Use the **`numpy`** library
    - `my_choice = np.random.choice(a=[x,y], p=[0.25,0.75])`
    - Note : `numpy random` documentation
  - however, ...
    - If f(y)=1 and f(z)=-1, the denominator becomes 0 (***divide-by-zero***.)
    - Depending on the situation, a negative probability might result.
    - What is the implementation method that **considers general situations**? (*Next page*)

Stochastic Hill Climbing (Stochastic Class)

- **`stochasticBest(self, neighbors, p)`**:
  - Implementation method considering the general situation:
  - Obtains a list of evaluation values of **neighbors** and store in **valuesForMin**
    - (Original problem) **Smaller values are better**
    - However, to make the probability-based selection calculation easier, the following operation is performed
  - Converts the list to the one in which larger values are better (**`valuesForMax`**)
    - Changes the values so that the larger the value, the higher the selection probability
    - Each original value is subtracted from a <u>large enough value</u> (ex. max(`valuesForMin`)+1 to avoid zero)
    - The converted values are stored in the **valuesForMax** list
  - Chooses at random from **`valuesForMax`** with probability proportional to its value



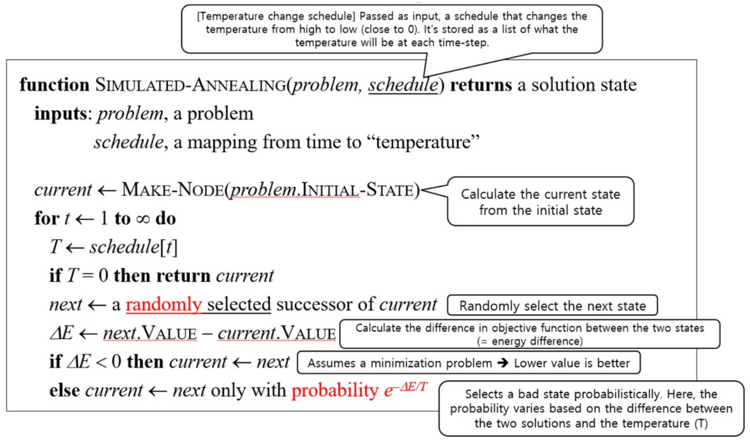
  - Returns the chosen neighbor together with its original evaluation value

예) 9 solutions were made, and v_0 to v_8 are the **`valuesForMax`**. A value is drawn randomly from the list. After generating a random number **`r`** within [0, sum(valuesForMax)), one solution is selected along with its index, like in the example

Simulated Annealing (SimulatedAnnealing Class)

- **Working Principle**
  - Generates one **next solution** randomly (just like **first-choice**)
  - Selects the **next solution** if it is **better** than the current solution.
  - Selects the **next solution probabilistically** even if it is **worse** than the current solution.
    - Uses the probability `exp(–dE/t)` for selection

- Selects a *bad solution* with a **random probability** and, through this, can find a **better solution**.
- **Note:** *Gradually reduce* the probability of selecting a bad solution.



[Temperature change schedule] Passed as input, a schedule that changes the temperature from high to low (close to 0). It's stored as a list of what the temperature will be at each time-step.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.VALUE – current.VALUE
    if ΔE < 0 then current ← next
    else current ← next only with probability $e^{-\Delta E/T}$
```

Calculate the current state from the initial state

Randomly select the next state

Calculate the difference in objective function between the two states (= energy difference)

Assumes a minimization problem ➔ Lower value is better

Selects a bad state probabilistically. Here, the probability varies based on the difference between the two solutions and the temperature (T)

- **Additional Member Variable**: There is one variable **`numSample`** storing the number of samples used to heuristically determine an initial temperature (초기 온도값)
  - It is currently preset to 100 (in **`SimulatedAnnealing`** Class)
  - Used in **`initTemp`** function (*See next slide*)
  - Used to generate the initial temperature value.

Simulated Annealing (SimulatedAnnealing Class)

- `initTemp(self, p)`:
  - Working method
    - Calculates the temperature `t` such that `exp(-dE/t)` $= 0.5$, and returns `t`
    - That is, it finds the temperature **t** where the probability of selecting a **bad solution** is **0.5** and returns it.
    - **Why?** It is reasonable to use a **t** that satisfies the above condition as the initial temperature.
  - Implementation
    - Takes k (= `self._numSample`) random samples and their neighbors from the domain of problem `p`
    - The process of generating an arbitrary initial value, randomly generating its neighboring solution, recording the difference, and calculating the average difference value is repeated k times.
    - Calculates the average difference between two solutions (= **dE** average value)
    - Sets the initial **t** value using the calculated average value of **dE** such that the probability of selecting a bad solution is 0.5 initially.

Simulated Annealing (SimulatedAnnealing Class)

- **`initTemp(self, p)`**:
  - **Working method**
    - Calculates the temperature **`t`** such that **`exp(-dE/t)`** $= 0.5$, and returns **`t`**
    - That is, it finds the temperature **t** where the probability of selecting a **bad solution** is **0.5** and returns it.
  - **Implementation**
    - Implemented as following:
      - ✓ For `m`-th iteration in `range(self._numSample)`:
        - » generate a random initial solution and evaluate (= `v0`)
        - » generate a random mutation and evaluate (= `v1`)
        - » calc `diffs(m) = abs(v1-v0)`
      - ✓ Calculates the average **`dE`** of their differences
        - » `dE = sum( diffs ) / self._numSample`
      - ✓ `return t = dE / math.log(2) # Initial Temperature Setting. Why is it this way? (Next Page)`

Simulated Annealing (SimulatedAnnealing Class)

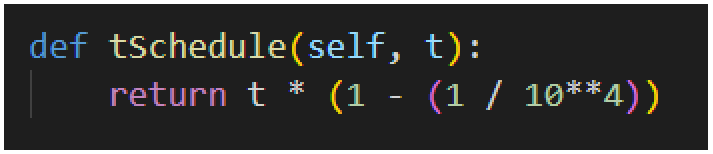
- `initTemp(self, p)`:
  - Find the temperature **T** at which the probability of selecting a **bad solution** becomes **0.5**.
  - Probability of selecting a bad **`soln`** (solution) : $p = e^{-dE/T}$
  - Find **T** where **p=0.5**
    - $p = \frac{1}{2} = e^{-dE/T}$
    - $\log\left(\frac{1}{2}\right) = \log e^{-dE/T} = -\frac{dE}{T}$
    - $-\log(2) = -\frac{dE}{T}$
    - $T = \frac{dE}{\log(2)}$
    - dE = The difference in value calculated probabilitstically

Simulated Annealing (SimulatedAnnealing Class)

- `initTemp(self, p)`:
  - **Summary of Operation**: Find the initial temperature **`t`** at which the probability of selecting a bad solution becomes 0.5.
    - Repeat for `numSample` times
      - ✓ Generate a random solution $c0$ and one random mutant (next solution) $c1$ from it.
      - ✓ Calculate the obj value $v0$ for $c0$ and the obj value $v1$ for $c1$.
    - dE = average of $abs(v1 - v0)$
      - ✓ That is, dE calculates the average difference between **two objective values**.
      - ✓ $t = dE \,/\, math.\log(2)$ you can find the initial temperature **`t`** where the probability of selecting a bad solution becomes **0.5**.

Simulated Annealing (SimulatedAnnealing Class)

- **`tSchedule(self, t)`**:
  - Calculates the next temperature using a simple formula, and returns it



```python
def tSchedule(self, t):
    return t * (1 - (1 / 10**4))
```

  - The temperature schedule values can be used in the form of a pre-calculated **list (array)**, or it can be implemented, as shown above, to take the **current temperature** as input and calculate the **next temperature**.

Simulated Annealing (SimulatedAnnealing Class)

- **`run()`** method explanation:
  - Starts from a random initial point
    - The randomly selected initial solution is recorded as the **best solution**, and this record is **updated** whenever a better solution is found.
  - The solution is **iteratively updated**, and the process terminates when the temperature (**t, temperature**) reaches **0** or the number of iterations reaches **`self._limitEval`**.
    - it uses another variable **`whenBestFound`** to record when the best-so-far solution has first been found (it records the iteration counter value at that time)
  - The temperature decreases every iteration according to an annealing schedule (**`self.tSchedule(t)`**)
    - Note: As temperature gets lower the probability of bad solution get selected decreases.
  - The initial temperature is heuristically determined so that the probability of accepting a worse neighbor becomes 0.5 initially (**`self.initTemp(p)`**)
  - Generate a random neighboring solution based on the **current solution**,
    - If the neighbor is a **better solution**, it is **always selected**. If not:
      - ✓ The probability of accepting a worse neighbor is **`exp(-dE/t)`**,
      - ✓ **`dE`** is the difference of the evaluation values (= **`valueNext - valueCurrent`**)
      - ✓ **`t`** is the current temperature