Evaluate this! - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

This expression evaluates as True: `not 1 and True`: False
This expression evaluates as True: `1 is not 1`: False
This expression evaluates as True: `bool([]) == bool({})`: True
This expression evaluates as True: `bool(1)`: True

This expression evaluates as None: `print(1)`: True
This expression evaluates as None: `(lambda x: None)(True)`: True
This expression evaluates as None: `'None' == None`: False
This expression evaluates as None: `None`: True

This expression evaluates as 1: `int('1')`: True
This expression evaluates as 1: `'3' - '2'`: False
This expression evaluates as 1: `int(True)`: True
This expression evaluates as 1: `int(2.9-1)`: True

Loops - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

To achieve the same semantic result, any `for` loop can be rewritten as a `while` loop.: True
It is possible to nest `while` blocks inside one another.: True
Python coerces any expression to a boolean value when used as an `if` condition.: True
It can happen that a `while` loop is run zero times.: True

A `while` loop may never stop.: True
A `for` loop will always execute at least once.: False
`while` loops inherently have a higher chance of problematic behavior than `for` loops.: True
Every `x` in `for x in enumerate(range(10))` is a tuple.: True

Type coercion refers to the implicit conversion of one type to another: True
It is possible to nest `while` blocks inside one another.: True
Every `x` in `for x in enumerate(range(10))` is a tuple.: True
`for` loops cannot be nested.: False

Functions - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

Every function has a return value.: True
Any function accepts at least one parameter.: False
Functions cannot have optional parameters.: False
Functions enable decomposition.: True

A function without any explicit return statements returns `None`: True
Function parameters with a default value are optional.: True
Any function accepts at least one parameter.: False
If called multiple times, a function is guaranteed to have the same return value every time.: False

---

Calling the same function twice always is guaranteed to have the same effect both times.: False
Functions cannot have optional parameters.: False
Function parameters with a default value are optional.: True
Some function don't return any kind of value, not even `None`.: False

---

If / Elif / Else - 3 points, 1 out of 2 tasks

Implement a function `normalize` which takes three numbers as parameters `number`, `lower` and `upper`. If `number` is smaller than `lower`, the function should return `lower`. If `number` is larger than `upper`, it should return `upper`. Otherwise, the function should return `number`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```python
def normalize(number, lower, upper):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```python
print( normalize(1.5, 0, 1) )
print( normalize(15, 10, 20) )
```

The calls above should produce the following output:

```
1
15
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def normalize(number, lower, upper):
    if number > upper:
        return upper
    elif number < lower:
        return lower
    return number

    # or:
    #return min(upper, max(lower, number))

# examples
print( normalize(1.5, 0, 1) )
print( normalize(15, 10, 20) )
```

Implement a function `check` which takes two non-negative integers as parameters `speed` and `limit`. If `speed` is greater than `limit`, the function should return `False`, otherwise it should return `True`. However, if `limit` is `0`, the function should return `True` whatever the value of `speed`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def check(speed, limit):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( check(130, 90) )
print( check(170, 0) )
```

The calls above should produce the following output:

```
False
True
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def check(speed, limit):
    if limit == 0:
        return True
    if speed > limit:
        return False
    return True

# examples
print( check(130, 90) )
print( check(170, 0) )
```

DÃ©jÃ vu - 6 points, 1 out of 3 tasks

Implement a function `length` which takes a single parameter `iterable`, which can be a list, tuple or string. The function should compute the length of `iterable` recursively.

For example, `[1, 2, [3, 4], True]`, `("a", 1, 2, None)` and `"good"` all have a length of `4`.

To obtain full points, your solution *must* be recursive. An iterative solution that works correctly will be awarded 50% of the points. You may *not* use the `len()` function!

You may assume that your function will only be called with lists, tuples or strings.

Use the following template:

```
def length(iterable):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( length([1, 2, [3, 4]]) )
print( length(("a", 1, 2, None)) )
print( length("oh dear") )
```

The calls above should produce the following output:

```
3
4
7
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def length(iterable):
    if iterable in ["", [], ()]:
        return 0
    return 1 + length(iterable[1:])

# examples
print( length([1, 2, [3, 4]]) )
print( length(("a", 1, 2, None)) )
print( length("oh dear") )
```

Implement a function `change` which takes a single positive integer parameter `pennies` and returns a tuple containing 5 integers representing the lowest possible number of dollars, quarters, dimes, nickels and pennies necessary to add up to the number of `pennies` provided. Dollars have a value of 100, a quarter is 25 pennies, a dime is 10 pennies and a nickel is 5 pennies.

For example, 162 pennies can be represented using 1 dollar, 2 quarters, 1 dime and 2 pennies.

You may assume that your function will only be called with non-negative integers.

Use the following template:

```
def change(pennies):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( change(162) )
print( change(3) )
```

The calls above should produce the following output:

```
(1, 2, 1, 0, 2)
(0, 0, 0, 0, 3)
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def change(pennies):
    dollars = pennies // 100
    pennies = pennies % 100
    quarters = pennies // 25
    pennies = pennies % 25
    dimes = pennies // 10
    pennies = pennies % 10
    nickels = pennies // 5
    pennies = pennies % 5
    return (dollars, quarters, dimes, nickels, pennies)

# examples
print( change(162) )
print( change(3) )
```

Implement a function `sum_divisibles` which takes two parameters, a positive integer `divisor` and a non-negative integer `limit`. The function should compute the sum of all values from `0` up to and including `limit` which are divisible by `divisor`.

For example, given a `limit` of `17` and a `divisor` of `4`, the return value should be `40` (which results from `4 + 8 + 12 + 16`).

You may assume that your function will only be called with non-negative integers.

Use the following template:

```
def sum_divisibles(limit, divisor):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( sum_divisibles(5, 2) )
print( sum_divisibles(11, 5) )
```

The calls above should produce the following output:

```
6
15
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def sum_divisibles(limit, divisor):
    res = 0
    for n in range(limit+1):
        if n % divisor == 0:
            res += n
    return res

# examples
print( sum_divisibles(5, 2) )
print( sum_divisibles(11, 5) )
```

Loops - 5 points, 1 out of 2 tasks

Implement a function `product` which takes two lists of numbers as parameters `xs` and `ys`. For every value *x* in `xs`, the function should multiply *x* with every value in `ys` and print each result, one per line. Note that this function does not return anything.

Important: even though there are various ways to solve this task, you **must** use at least one `for` loop to solve this problem and you may not use `range()`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def product(xs, ys):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( product([5], [10, 11]) )
print( product([2, 3, 4], [1, 10, 20]) )
```

The calls above should produce the following output:

```
50
55
None
2
20
40
3
30
60
4
40
80
None
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def product(xs, ys):
    for x in xs:
        for y in ys:
            print(x*y)

# examples
print( product([5], [10, 11]) )
print( product([2, 3, 4], [1, 10, 20]) )
```

Implement a function `print_range` which takes three numbers as parameters `start`, `stop` and `step`. The function should *print* numbers beginning at `start` in increments of `step` up to (but not including) `stop`, one number per line. You can assume that `start` will never be larger than `stop` and that `step` is always positive. Note that this function does not return anything.

Important: even though there are various ways to solve this task, you **must** use a while loop to solve this problem and you may not use `range()`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def print_range(start, stop, step):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( print_range(0, 5, 1) )
print( print_range(5, 20, 3) )
```

The calls above should produce the following output:

```
0
1
2
3
4
None
5
8
11
14
17
None
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def print_range(start, stop, step):
    current = start
    while current < stop:
        print(current)
        current += step

# examples
print( print_range(0, 5, 1) )
print( print_range(5, 20, 3) )
```

Functions (1) - 6 points, 1 out of 2 tasks

Implement a function `add` which can take two numbers as parameters `a` and `b`. Both parameters are optional and shall be `0` by default. The function should return the sum of `a` and `b`.

Important: In this task, you must write the function signature yourself. Make sure that your function is called `add`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def ... # Write the function signature and implement the function
```

The following example illustrates how the solution function may be called:

```
print( add(2, 2) )
print( add() )
```

The calls above should produce the following output:

```
4
0
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def add(a=0, b=0):
    return a+b

# examples
print( add(2, 2) )
print( add() )
```

Implement a function `multiply` which takes a number as parameter `n` and, optionally, another number as parameter `factor`, which shall be `1` by default. The function should return `n` multiplied by `factor`.

Important: In this task, you must write the function signature yourself. Make sure that your function is called `multiply`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def ... # Write the function signature and implement the function
```

The following example illustrates how the solution function may be called:

```
print( multiply(2, 2) )
print( multiply(2) )
```

The calls above should produce the following output:

```
4
2
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def multiply(n, factor=1):
    return n*factor

# examples
print( multiply(2, 2) )
print( multiply(2) )
```

Data structures - 7 points, 1 out of 3 tasks

Implement a function `duplicate_every` which takes two parameters: a list `l` and a positive integer `n`. The function should return a list containing the elements from `l` but with every `n`'th element occuring twice. For example, if `n` is 3, every third element should be repeated once.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def duplicate_every(l, n):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( duplicate_every([1, 3, 4, 5], 2) )
print( duplicate_every([1, 4, 5, 4, 3, 2, 1], 3) )
```

The calls above should produce the following output:

```
[1, 3, 3, 4, 5, 5]
[1, 4, 5, 5, 4, 3, 2, 2, 1]
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def duplicate_every(l, n):
    res = []
    for i, item in enumerate(l):
        if (i+1) % n == 0:
            res.append(item)
        res.append(item)
    return res

# examples
print( duplicate_every([1, 3, 4, 5], 2) )
print( duplicate_every([1, 4, 5, 4, 3, 2, 1], 3) )
```

Implement a function `sort_dict_values` which takes a dictionary as the only parameter `d`. The given dictionary can contain an arbitrary number of key/value pairs, but all of the values will be lists containing values of the same type. The function should return a dictionary with the same key/value pairs, but where each value has been sorted (using Python's built-in `sorted()` or `.sort()` functions).

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def sort_dict_values(d):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( sort_dict_values({"a": [3, 1, 2]}) )
print( sort_dict_values({1: ["z", "az"], 2: [1]}) )
```

The calls above should produce the following output:

```
{'a': [1, 2, 3]}
{1: ['az', 'z'], 2: [1]}
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def sort_dict_values(d):
    res = {}
    for key, value in d.items():
        res[key] = sorted(value)
    return res

# examples
print( sort_dict_values({"a": [3, 1, 2]}) )
print( sort_dict_values({1: ["z", "az"], 2: [1]}) )
```

Implement a function `remove_every` which takes two parameters: a list `l`(containing arbitrary values) and a positive integer `n`. The function should return a list containing the elements from `l` but with every `n`'th element removed. For example, if `n` is 3, every third element should be removed.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def remove_every(l, n):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( remove_every([1, 2, 3, 4, 5], 2) )
print( remove_every([1, 2, 3, 4, 5, 4, 3, 2, 1], 3) )
```

The calls above should produce the following output:

```
[1, 3, 5]
[1, 2, 4, 5, 3, 2]
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def remove_every(l, n):
    res = []
    for i, item in enumerate(l):
        if (i+1) % n == 0:
            continue
        res.append(item)
    return res

# examples
print( remove_every([1, 2, 3, 4, 5], 2) )
print( remove_every([1, 2, 3, 4, 5, 4, 3, 2, 1], 3) )
```

Functions (2) - 6 points, 1 out of 3 tasks

Implement a function `add` which takes a number `n` as the only parameter. `add` should return a function that takes exactly one parameter (also a number). The returned function should add `n` to the value passed into it and return the result.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def add(n):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( add(3)(10) )
print( add(-5)(15) )
```

The calls above should produce the following output:

```
13
```

```
10
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def add(n):
    def f(x):
        return n + x
    return f

    # or
    #return lambda x: n + x

# examples
print( add(3)(10) )
print( add(-5)(15) )
```

Implement a function `apply` which takes three parameters: a function `f1`, a function `f2`, and a value `value`. These parameters will always be chosen such that the function `f1` takes one parameter of the same type as `value` and `f2` will take one parameter of whatever type the return value of `f1` will be.

The function `apply` should do the following:

- Call `f1` with `value` as the parameter, which will result in some return value
- Call `f2` with the return value of the previous call as the parameter, which will result in another value
- Return the resulting value

For example, calling `apply(min, chr, [101, 97, 99])` should first call `min` with `[101, 97, 99]` as the parameter, which will result in `97`. Then, `chr` should be called with `97` as the parameter, resulting in `a`, which shall be returned.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def apply(f1, f2, value):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( apply(min, chr, [101, 97, 99]) )
print( apply(max, lambda x: x*2, [2, 50, 3]) )
print( apply(sorted, lambda s: s[0].upper(), "what") )
```

The calls above should produce the following output:

```
a
100
A
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def apply(f1, f2, value):
    return f2(f1(value))

# examples
print( apply(min, chr, [101, 97, 99]) )
print( apply(max, lambda x: x*2, [2, 50, 3]) )
print( apply(sorted, lambda s: s[0].upper(), "what") )
```

Implement a function `apply` which takes two parameters:

- a function `f`, which shall always be a function that takes exactly one parameter
- a list `values`

The function `apply` should iterate through `values` and for each element call the function `f` with the element as the parameter. The return value of `apply` should be a list containing the results of all the calls.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def apply(f, values):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( apply(max, [[1,2], [4.5, 1, 3]]) )
print( apply(lambda x: x.upper(), ["hello", "world"]) )
```

The calls above should produce the following output:

```
[2, 4.5]
['HELLO', 'WORLD']
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def apply(f, values):
    res = []
    for v in values:
        res.append(f(v))
    return res

    # or simply
    #return [f(v) for v in values]

# examples
print( apply(max, [[1,2], [4.5, 1, 3]]) )
print( apply(lambda x: x.upper(), ["hello", "world"]) )
```

Riddle me this! - 21 points, 1 out of 3 tasks

You have been tasked with programming a simulator for number lotteries.

Implement a function `lottery` according to these implementation instructions:

- `lottery` takes three parameters:
    - an integer `limit` specifying the largest number that can be drawn in this lottery
    - a list `guess` of *n* unique integer numbers between 1 and (including) `limit`; this is the guess provided by the player. *n* will always be larger than 0
    - a number `prize` indicating the maximum amount that can be won in the lottery
- `lottery` shall then perform the following procedure:
    - draw *n* unique random numbers in the range from (and including) 1 to (and including) *limit*. *n* is implied by the length of the guess provided by the player.
    - check how many numbers in `guess` appear in the random draw
    - calculate the payout according to the following rule: For an exact match, the whole prize is paid out. If one number differs, half of the prize is paid out. If two numbers differ, a quarter of the prize is paid out, and so on. If none of the numbers match, the prize money is `0`.
- In the end `lottery` should return three values: the randomly generated numbers in ascending order, the number of matches, and the payout.

Note: Make sure you return the correct number of return values. Partial points are awarded even if not all return values are correct, but only if all three values are present.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
import random
# this line of code makes it so that calls to random always produce the same
# successive values so that the examples below always produce the same results
random.seed(7)

def lottery(limit, guess, prize):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( lottery(52, [4, 8, 15, 16, 23, 42], 1000000) ) # 2 matching
print( lottery(3, [1, 2, 3], 1000000) )               # inevitable perfect match
print( lottery(10000, [1, 2, 3], 1000000) )           # zero matching
```

The calls above should produce the following output:

```
([4, 5, 10, 21, 26, 42], 2, 62500.0)
([1, 2, 3], 3, 1000000)
([951, 8314, 9549], 0, 0)
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env
```

```python
import random
# this line of code makes it so that calls to random always produce the same
# successive values so that the examples below always produce the same results
random.seed(7)

def lottery(limit, guess, prize):
    match = 0
    numbers = []
    while len(numbers) < len(guess):
        n = random.randint(1, limit)
        if n not in numbers:
            numbers.append(n)
    for number in guess:
        if number in numbers:
            match += 1
    payout = prize
    for n in range(len(numbers) - match):
        payout /= 2
    if match == 0:
        payout = 0
    numbers.sort()
    return numbers, match, payout

    # or if you already know Python:
    #numbers = sorted(random.sample(range(1, limit+1), k=len(guess)))
    #match = len(set(numbers) & set(guess))
    #payout = 0 if not match else prize*(1/2**(len(guess)-match))
    #return numbers, match, payout

# examples
print( lottery(52, [4, 8, 15, 16, 23, 42], 1000000) ) # 2 matching
print( lottery(3, [1, 2, 3], 1000000) )               # inevitable perfect match
print( lottery(10000, [1, 2, 3], 1000000) )           # zero matching
```

You have been tasked with programming a billing system for hotels. A hotel offers a variety of products (rooms and additional services), for which the prices are stored in a dictionary. A hotel booking can comprise an arbitrary positive number of products.

Implement a function `booking` according to these implementation instructions:

- `booking` takes two parameters: the first is a dictionary containing the prices as shown in the example `milton_hotel`; the second is a list of products booked.
- Depending on which products are booked together, various discounts are applied:
    - For each executive room, one parking space is offered for free
    - For each suite room, one parking space and one breakfast are offered for free
    - If at least 3 rooms of any type are booked, all room prices are reduced by 10%
- The function should return three values: the total price of the booking before any discounts, the total discount, and the total price after subtracting the discount.
- Of course, your function should work with arbitrary prices as specified in the dictionary, but you can assume that the keys given in the example will always be present in the dictionary.
- Note that discounts are only applied if the free product has actually been booked. For example, if an executive room is booked without a parking space, then there is no discount on that booking.

Note: Do *not* lookup prices in `milton_hotel`! You need to look them up in the function paramter `pricing`!

Note: Make sure you return the correct number of return values. Partial points are awarded even if not all return values are correct, but only if all three values are present.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```python
milton_hotel = {
    "standard":  120.00,
    "executive": 160.00,
    "suite":     320.00,
    "breakfast": 25.00,
    "parking":   30.00,
}

def booking(pricing, products):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```python
print( booking(milton_hotel, [ # no discounts
    "executive", "breakfast"
]))
print( booking(milton_hotel, [ # one free parking
    "standard", "executive", "breakfast", "breakfast", "parking"
]))
print( booking(milton_hotel, [ # two free parking (one applied) and one free breakfast plus 10% off all rooms
    "standard", "executive", "suite", "parking",
    "breakfast", "breakfast", "breakfast"
]))
```

The calls above should produce the following output:

```
(185.0, 0.0, 185.0)
(360.0, 30.0, 330.0)
(705.0, 115.0, 590.0)
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```python
#!/usr/bin/env

milton_hotel = {
    "standard":  120.00,
    "executive": 160.00,
    "suite":     320.00,
    "breakfast": 25.00,
    "parking":   30.00,
}

def booking(pricing, products):
    total = 0
    for product in products:
        total += pricing[product]
    executives = products.count("executive")
    suites = products.count("suite")
    rooms = executives + suites + products.count("standard")
    parkings = products.count("parking")
    breakfasts = products.count("breakfast")
    room_types = ["standard", "executive", "suite"]
    room_discount = 0
    if rooms > 2:
        for product in products:
            if product in room_types:
                room_discount += 0.1 * pricing[product]
    free_breakfasts = min(breakfasts, suites)
    free_parkings = min(parkings, suites + executives)
    discount = (
      free_breakfasts * pricing["breakfast"] +
      free_parkings * pricing["parking"] +
      room_discount)
    return total, discount, total - discount

# examples
print( booking(milton_hotel, [ # no discounts
    "executive", "breakfast"
]))
print( booking(milton_hotel, [ # one free parking
    "standard", "executive", "breakfast", "breakfast", "parking"
]))
print( booking(milton_hotel, [ # two free parking (one applied) and one free breakfast plus 10% off all rooms
    "standard", "executive", "suite", "parking",
    "breakfast", "breakfast", "breakfast"
]))
```

You have been tasked with writing a parser for a simple programming language. The language consists only of three kinds of expressions:

- Integer variable assignments, like `x = 2` or `abc=123`
- Function calls, like `f(x, abc)` or `do_stuff ()`
- Comments, which are anything following the first octothorpe (`#`) on a line

Implement a function `parse` that takes a string and returns three values:

- All variables as a dictionary, where the keys are variable names and the values are the values assigned to each variable. Make sure to convert the string numbers to integers!
- All function calls as a list of tuples, where each tuple has two elements: the first element is the name of the function being called and the second element is a list containing all the parameter names.
- All comments as a list of strings.

Observe the following implementation details:

- There will only be one variable assignment or function call per line
- There can be empty lines in the input string
- If a variable is re-declared, the old assignment is lost
- Function calls may pass zero parameters, in which case the corresponding parameter list in the output should be empty.
- In principle, variable names and function calls can contain any characters except whitespace, commas, `=`, `(` and `)`
- Whitespace around variable names, variable values, function names and function parameters should be stripped.
- For comments, the octothorpe (`#`) and any leading or trailing whitespace should be stripped
- Whitespace can be stripped using Python's `strip()` function. E.g.: `" abc ".strip()` is `"abc"`

Note: Make sure you return the correct number of return values. Partial points are awarded even if not all return values are correct, but only if all three values are present. If you are unable to parse all kinds of expressions, at least return those which you can.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```python
def parse(program):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```python
print( parse("""x = 1""") )
print( parse("""  y=2  """) )
print( parse("""fun()""") )
print( parse("""do_stuff (abc)""") )
print( parse("""# comment   """) )
print( parse("""
x = 1
y=   2
y=   3       # y=2 is now lost

fun( x,   y) #a comment after a #, including this last part: fun(x)
fun(a,abc)
"""))
```

The calls above should produce the following output:

```
({'x': 1}, [], [])
({'y': 2}, [], [])
({}, [('fun', [])], [])
({}, [('do_stuff', ['abc'])], [])
({}, [], ['comment'])
({'x': 1, 'y': 3}, [('fun', ['x', 'y']), ('fun', ['a', 'abc'])], ['y=2 is now lost', 'a comment after a #, including this last part: fun(x)'])
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def parse(program):
    variables = {}
    calls = []
    comments = []
    for line in program.splitlines():
        # extract comment
        octothorpe = line.find('#')
        if octothorpe != -1:
            comments.append(line[octothorpe+1:].strip())
            line = line[:octothorpe]
        # detect variable declaration
        assignment = line.find('=')
        if assignment != -1:
            var, value = line.split("=")
            variables[var.strip()] = int(value.strip())
        # detect function call
        fun = line.find('(')
        if fun != -1:
            name, rest = line.split("(")
            name = name.strip()
            rest = rest.strip()
            rest = rest[:-1].strip() # remove ")"
            parameters = rest.split(",")
            clean_parameters = []
            for var in parameters:
                var = var.strip()
                if var != "":
                    clean_parameters.append(var.strip())
            calls.append((name, clean_parameters))

    return variables, calls, comments

# examples
print( parse("""x = 1""") )
print( parse("""  y=2  """) )
print( parse("""fun()""") )
print( parse("""do_stuff (abc)""") )
print( parse("""# comment   """) )
print( parse("""
x = 1
y=   2
y=   3       # y=2 is now lost

fun( x,   y) #a comment after a #, including this last part: fun(x)
fun(a,abc)
"""))
```

Evaluate this! - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

This expression evaluates as True: `not 1 and True`: False
This expression evaluates as True: `1 is not 1`: False
This expression evaluates as True: `bool([]) == bool({})`: True
This expression evaluates as True: `bool(1)`: True

This expression evaluates as None: `print(1)`: True
This expression evaluates as None: `(lambda x: None)(True)`: True
This expression evaluates as None: `'None' == None`: False
This expression evaluates as None: `None`: True

This expression evaluates as 1: `int('1')`: True
This expression evaluates as 1: `'3' - '2'`: False
This expression evaluates as 1: `int(True)`: True
This expression evaluates as 1: `int(2.9-1)`: True

Loops - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

To achieve the same semantic result, any `for` loop can be rewritten as a `while` loop.: True
It is possible to nest `while` blocks inside one another.: True
Python coerces any expression to a boolean value when used as an `if` condition.: True
It can happen that a `while` loop is run zero times.: True

A `while` loop may never stop.: True
A `for` loop will always execute at least once.: False
`while` loops inherently have a higher chance of problematic behavior than `for` loops.: True
Every `x` in `for x in enumerate(range(10))` is a tuple.: True

Type coercion refers to the implicit conversion of one type to another: True
It is possible to nest `while` blocks inside one another.: True
Every `x` in `for x in enumerate(range(10))` is a tuple.: True
`for` loops cannot be nested.: False

Functions - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

Every function has a return value.: True
Any function accepts at least one parameter.: False
Functions cannot have optional parameters.: False
Functions enable decomposition.: True

A function without any explicit return statements returns `None`: True
Function parameters with a default value are optional.: True
Any function accepts at least one parameter.: False
If called multiple times, a function is guaranteed to have the same return value every time.: False

---

Calling the same function twice always is guaranteed to have the same effect both times.: False
Functions cannot have optional parameters.: False
Function parameters with a default value are optional.: True
Some function don't return any kind of value, not even `None`.: False

---

If / Elif / Else - 3 points, 1 out of 2 tasks

Implement a function `normalize` which takes three numbers as parameters `number`, `lower` and `upper`. If `number` is smaller than `lower`, the function should return `lower`. If `number` is larger than `upper`, it should return `upper`. Otherwise, the function should return `number`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```python
def normalize(number, lower, upper):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```python
print( normalize(1.5, 0, 1) )
print( normalize(15, 10, 20) )
```

The calls above should produce the following output:

```
1
15
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```python
#!/usr/bin/env

def normalize(number, lower, upper):
    if number > upper:
        return upper
    elif number < lower:
        return lower
    return number

    # or:
    #return min(upper, max(lower, number))
```

```
# examples
print( normalize(1.5, 0, 1) )
print( normalize(15, 10, 20) )
```

Implement a function `check` which takes two non-negative integers as parameters `speed` and `limit`. If `speed` is greater than `limit`, the function should return `False`, otherwise it should return `True`. However, if `limit` is `0`, the function should return `True` whatever the value of `speed`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def check(speed, limit):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( check(130, 90) )
print( check(170, 0) )
```

The calls above should produce the following output:

```
False
True
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def check(speed, limit):
    if limit == 0:
        return True
    if speed > limit:
        return False
    return True

# examples
print( check(130, 90) )
print( check(170, 0) )
```

DÃ©jÃ vu - 6 points, 1 out of 3 tasks

Implement a function `length` which takes a single parameter `iterable`, which can be a list, tuple or string. The function should compute the length of `iterable` recursively.

For example, `[1, 2, [3, 4], True]`, `("a", 1, 2, None)` and `"good"` all have a length of `4`.

To obtain full points, your solution *must* be recursive. An iterative solution that works correctly will be awarded 50% of the points. You may *not* use the `len()` function!

You may assume that your function will only be called with lists, tuples or strings.

Use the following template:

```
def length(iterable):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( length([1, 2, [3, 4]]) )
print( length(("a", 1, 2, None)) )
print( length("oh dear") )
```

The calls above should produce the following output:

```
3
4
7
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def length(iterable):
    if iterable in ["", [], ()]:
        return 0
    return 1 + length(iterable[1:])
```

```
# examples
print( length([1, 2, [3, 4]]) )
print( length(("a", 1, 2, None)) )
print( length("oh dear") )
```

Implement a function `change` which takes a single positive integer parameter `pennies` and returns a tuple containing 5 integers representing the lowest possible number of dollars, quarters, dimes, nickels and pennies necessary to add up to the number of `pennies` provided. Dollars have a value of 100, a quarter is 25 pennies, a dime is 10 pennies and a nickel is 5 pennies.

For example, 162 pennies can be represented using 1 dollar, 2 quarters, 1 dime and 2 pennies.

You may assume that your function will only be called with non-negative integers.

Use the following template:

```
def change(pennies):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( change(162) )
print( change(3) )
```

The calls above should produce the following output:

```
(1, 2, 1, 0, 2)
(0, 0, 0, 0, 3)
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def change(pennies):
    dollars = pennies // 100
    pennies = pennies % 100
    quarters = pennies // 25
    pennies = pennies % 25
    dimes = pennies // 10
    pennies = pennies % 10
    nickels = pennies // 5
    pennies = pennies % 5
    return (dollars, quarters, dimes, nickels, pennies)

# examples
print( change(162) )
print( change(3) )
```

Implement a function `sum_divisibles` which takes two parameters, a positive integer `divisor` and a non-negative integer `limit`. The function should compute the sum of all values from `0` up to and including `limit` which are divisible by `divisor`.

For example, given a `limit` of `17` and a `divisor` of `4`, the return value should be `40` (which results from `4 + 8 + 12 + 16`).

You may assume that your function will only be called with non-negative integers.

Use the following template:

```
def sum_divisibles(limit, divisor):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( sum_divisibles(5, 2) )
print( sum_divisibles(11, 5) )
```

The calls above should produce the following output:

```
6
15
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def sum_divisibles(limit, divisor):
    res = 0
    for n in range(limit+1):
        if n % divisor == 0:
            res += n
    return res
```

```
# examples
print( sum_divisibles(5, 2) )
print( sum_divisibles(11, 5) )
```

Loops - 5 points, 1 out of 2 tasks

Implement a function `product` which takes two lists of numbers as parameters `xs` and `ys`. For every value *x* in `xs`, the function should multiply *x* with every value in `ys` and print each result, one per line. Note that this function does not return anything.

Important: even though there are various ways to solve this task, you **must** use at least one `for` loop to solve this problem and you may not use `range()`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def product(xs, ys):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( product([5], [10, 11]) )
print( product([2, 3, 4], [1, 10, 20]) )
```

The calls above should produce the following output:

```
50
55
None
2
20
40
3
30
60
4
40
80
None
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def product(xs, ys):
    for x in xs:
        for y in ys:
            print(x*y)

# examples
print( product([5], [10, 11]) )
print( product([2, 3, 4], [1, 10, 20]) )
```

Implement a function `print_range` which takes three numbers as parameters `start`, `stop` and `step`. The function should *print* numbers beginning at `start` in increments of `step` up to (but not including) `stop`, one number per line. You can assume that `start` will never be larger than `stop` and that `step` is always positive. Note that this function does not return anything.

Important: even though there are various ways to solve this task, you **must** use a while loop to solve this problem and you may not use `range()`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def print_range(start, stop, step):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( print_range(0, 5, 1) )
print( print_range(5, 20, 3) )
```

The calls above should produce the following output:

```
0
1
2
3
4
None
5
```

```
8
11
14
17
None
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def print_range(start, stop, step):
    current = start
    while current < stop:
        print(current)
        current += step

# examples
print( print_range(0, 5, 1) )
print( print_range(5, 20, 3) )
```

Functions (1) - 6 points, 1 out of 2 tasks

Implement a function `add` which can take two numbers as parameters `a` and `b`. Both parameters are optional and shall be `0` by default. The function should return the sum of `a` and `b`.

Important: In this task, you must write the function signature yourself. Make sure that your function is called `add`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def ... # Write the function signature and implement the function
```

The following example illustrates how the solution function may be called:

```
print( add(2, 2) )
print( add() )
```

The calls above should produce the following output:

```
4
0
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def add(a=0, b=0):
    return a+b

# examples
print( add(2, 2) )
print( add() )
```

Implement a function `multiply` which takes a number as parameter `n` and, optionally, another number as parameter `factor`, which shall be `1` by default. The function should return `n` multiplied by `factor`.

Important: In this task, you must write the function signature yourself. Make sure that your function is called `multiply`.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def ... # Write the function signature and implement the function
```

The following example illustrates how the solution function may be called:

```
print( multiply(2, 2) )
print( multiply(2) )
```

The calls above should produce the following output:

```
4
2
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def multiply(n, factor=1):
    return n*factor

# examples
print( multiply(2, 2) )
print( multiply(2) )
```

Data structures - 7 points, 1 out of 3 tasks

Implement a function `duplicate_every` which takes two parameters: a list `l` and a positive integer `n`. The function should return a list containing the elements from `l` but with every `n`'th element occuring twice. For example, if `n` is 3, every third element should be repeated once.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def duplicate_every(l, n):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( duplicate_every([1, 3, 4, 5], 2) )
print( duplicate_every([1, 4, 5, 4, 3, 2, 1], 3) )
```

The calls above should produce the following output:

```
[1, 3, 3, 4, 5, 5]
[1, 4, 5, 5, 4, 3, 2, 2, 1]
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def duplicate_every(l, n):
    res = []
    for i, item in enumerate(l):
        if (i+1) % n == 0:
            res.append(item)
        res.append(item)
    return res

# examples
print( duplicate_every([1, 3, 4, 5], 2) )
print( duplicate_every([1, 4, 5, 4, 3, 2, 1], 3) )
```

Implement a function `sort_dict_values` which takes a dictionary as the only parameter `d`. The given dictionary can contain an arbitrary number of key/value pairs, but all of the values will be lists containing values of the same type. The function should return a dictionary with the same key/value pairs, but where each value has been sorted (using Python's built-in `sorted()` or `.sort()` functions).

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def sort_dict_values(d):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( sort_dict_values({"a": [3, 1, 2]}) )
print( sort_dict_values({1: ["z", "az"], 2: [1]}) )
```

The calls above should produce the following output:

```
{'a': [1, 2, 3]}
{1: ['az', 'z'], 2: [1]}
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def sort_dict_values(d):
    res = {}
    for key, value in d.items():
```

```python
        res[key] = sorted(value)
    return res

# examples
print( sort_dict_values({"a": [3, 1, 2]}) )
print( sort_dict_values({1: ["z", "az"], 2: [1]}) )
```

Implement a function `remove_every` which takes two parameters: a list `l`(containing arbitrary values) and a positive integer `n`. The function should return a list containing the elements from `l` but with every `n`'th element removed. For example, if `n` is 3, every third element should be removed.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```python
def remove_every(l, n):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```python
print( remove_every([1, 2, 3, 4, 5], 2) )
print( remove_every([1, 2, 3, 4, 5, 4, 3, 2, 1], 3) )
```

The calls above should produce the following output:

```
[1, 3, 5]
[1, 2, 4, 5, 3, 2]
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```python
#!/usr/bin/env

def remove_every(l, n):
    res = []
    for i, item in enumerate(l):
        if (i+1) % n == 0:
            continue
        res.append(item)
    return res

# examples
print( remove_every([1, 2, 3, 4, 5], 2) )
print( remove_every([1, 2, 3, 4, 5, 4, 3, 2, 1], 3) )
```

Functions (2) - 6 points, 1 out of 3 tasks

Implement a function `add` which takes a number `n` as the only parameter. `add` should return a function that takes exactly one parameter (also a number). The returned function should add `n` to the value passed into it and return the result.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```python
def add(n):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```python
print( add(3)(10) )
print( add(-5)(15) )
```

The calls above should produce the following output:

```
13
10
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```python
#!/usr/bin/env

def add(n):
    def f(x):
        return n + x
    return f

    # or
    #return lambda x: n + x
```

```
# examples
print( add(3)(10) )
print( add(-5)(15) )
```

Implement a function `apply` which takes three parameters: a function `f1`, a function `f2`, and a value `value`. These parameters will always be chosen such that the function `f1` takes one parameter of the same type as `value` and `f2` will take one parameter of whatever type the return value of `f1` will be.

The function `apply` should do the following:

- Call `f1` with `value` as the parameter, which will result in some return value
- Call `f2` with the return value of the previous call as the parameter, which will result in another value
- Return the resulting value

For example, calling `apply(min, chr, [101, 97, 99])` should first call `min` with `[101, 97, 99]` as the parameter, which will result in `97`. Then, `chr` should be called with `97` as the parameter, resulting in `a`, which shall be returned.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def apply(f1, f2, value):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( apply(min, chr, [101, 97, 99]) )
print( apply(max, lambda x: x*2, [2, 50, 3]) )
print( apply(sorted, lambda s: s[0].upper(), "what") )
```

The calls above should produce the following output:

```
a
100
A
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def apply(f1, f2, value):
    return f2(f1(value))

# examples
print( apply(min, chr, [101, 97, 99]) )
print( apply(max, lambda x: x*2, [2, 50, 3]) )
print( apply(sorted, lambda s: s[0].upper(), "what") )
```

Implement a function `apply` which takes two parameters:

- a function `f`, which shall always be a function that takes exactly one parameter
- a list `values`

The function `apply` should iterate through `values` and for each element call the function `f` with the element as the parameter. The return value of `apply` should be a list containing the results of all the calls.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
def apply(f, values):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( apply(max, [[1,2], [4.5, 1, 3]]) )
print( apply(lambda x: x.upper(), ["hello", "world"]) )
```

The calls above should produce the following output:

```
[2, 4.5]
['HELLO', 'WORLD']
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

def apply(f, values):
```

```python
    res = []
    for v in values:
        res.append(f(v))
    return res

    # or simply
    #return [f(v) for v in values]

# examples
print( apply(max, [[1,2], [4.5, 1, 3]]) )
print( apply(lambda x: x.upper(), ["hello", "world"]) )
```

Riddle me this! - 21 points, 1 out of 3 tasks

You have been tasked with programming a simulator for number lotteries.

Implement a function `lottery` according to these implementation instructions:

- `lottery` takes three parameters:
    - an integer `limit` specifying the largest number that can be drawn in this lottery
    - a list `guess` of *n* unique integer numbers between 1 and (including) `limit`; this is the guess provided by the player. *n* will always be larger than 0
    - a number `prize` indicating the maximum amount that can be won in the lottery
- `lottery` shall then perform the following procedure:
    - draw *n* unique random numbers in the range from (and including) 1 to (and including) *limit*. *n* is implied by the length of the guess provided by the player.
    - check how many numbers in `guess` appear in the random draw
    - calculate the payout according to the following rule: For an exact match, the whole prize is paid out. If one number differs, half of the prize is paid out. If two numbers differ, a quarter of the prize is paid out, and so on. If none of the numbers match, the prize money is `0`.
- In the end `lottery` should return three values: the randomly generated numbers in ascending order, the number of matches, and the payout.

Note: Make sure you return the correct number of return values. Partial points are awarded even if not all return values are correct, but only if all three values are present.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```python
import random
# this line of code makes it so that calls to random always produce the same
# successive values so that the examples below always produce the same results
random.seed(7)

def lottery(limit, guess, prize):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```python
print( lottery(52, [4, 8, 15, 16, 23, 42], 1000000) ) # 2 matching
print( lottery(3, [1, 2, 3], 1000000) )               # inevitable perfect match
print( lottery(10000, [1, 2, 3], 1000000) )           # zero matching
```

The calls above should produce the following output:

```
([4, 5, 10, 21, 26, 42], 2, 62500.0)
([1, 2, 3], 3, 1000000)
([951, 8314, 9549], 0, 0)
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```python
#!/usr/bin/env

import random
# this line of code makes it so that calls to random always produce the same
# successive values so that the examples below always produce the same results
random.seed(7)

def lottery(limit, guess, prize):
    match = 0
    numbers = []
    while len(numbers) < len(guess):
        n = random.randint(1, limit)
        if n not in numbers:
            numbers.append(n)
    for number in guess:
        if number in numbers:
            match += 1
    payout = prize
    for n in range(len(numbers) - match):
        payout /= 2
    if match == 0:
        payout = 0
    numbers.sort()
    return numbers, match, payout

    # or if you already know Python:
    #numbers = sorted(random.sample(range(1, limit+1), k=len(guess)))
    #match = len(set(numbers) & set(guess))
    #payout = 0 if not match else prize*(1/2**(len(guess)-match))
    #return numbers, match, payout

# examples
print( lottery(52, [4, 8, 15, 16, 23, 42], 1000000) ) # 2 matching
print( lottery(3, [1, 2, 3], 1000000) )               # inevitable perfect match
print( lottery(10000, [1, 2, 3], 1000000) )           # zero matching
```

You have been tasked with programming a billing system for hotels. A hotel offers a variety of products (rooms and additional services), for which the prices are stored in a dictionary. A hotel booking can comprise an arbitrary positive number of products.

Implement a function `booking` according to these implementation instructions:

- `booking` takes two parameters: the first is a dictionary containing the prices as shown in the example `milton_hotel`; the second is a list of products booked.
- Depending on which products are booked together, various discounts are applied:
  - For each executive room, one parking space is offered for free
  - For each suite room, one parking space and one breakfast are offered for free
  - If at least 3 rooms of any type are booked, all room prices are reduced by 10%
- The function should return three values: the total price of the booking before any discounts, the total discount, and the total price after subtracting the discount.
- Of course, your function should work with arbitrary prices as specified in the dictionary, but you can assume that the keys given in the example will always be present in the dictionary.
- Note that discounts are only applied if the free product has actually been booked. For example, if an executive room is booked without a parking space, then there is no discount on that booking.

Note: Do *not* lookup prices in `milton_hotel`! You need to look them up in the function paramter `pricing`!

Note: Make sure you return the correct number of return values. Partial points are awarded even if not all return values are correct, but only if all three values are present.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```
milton_hotel = {
    "standard":  120.00,
    "executive": 160.00,
    "suite":     320.00,
    "breakfast": 25.00,
    "parking":   30.00,
}

def booking(pricing, products):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( booking(milton_hotel, [ # no discounts
    "executive", "breakfast"
]))
print( booking(milton_hotel, [ # one free parking
    "standard", "executive", "breakfast", "breakfast", "parking"
]))
print( booking(milton_hotel, [ # two free parking (one applied) and one free breakfast plus 10% off all rooms
    "standard", "executive", "suite", "parking",
    "breakfast", "breakfast", "breakfast"
]))
```

The calls above should produce the following output:

```
(185.0, 0.0, 185.0)
(360.0, 30.0, 330.0)
(705.0, 115.0, 590.0)
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```
#!/usr/bin/env

milton_hotel = {
    "standard":  120.00,
    "executive": 160.00,
    "suite":     320.00,
    "breakfast": 25.00,
    "parking":   30.00,
}
```

```python
def booking(pricing, products):
    total = 0
    for product in products:
        total += pricing[product]
    executives = products.count("executive")
    suites = products.count("suite")
    rooms = executives + suites + products.count("standard")
    parkings = products.count("parking")
    breakfasts = products.count("breakfast")
    room_types = ["standard", "executive", "suite"]
    room_discount = 0
    if rooms > 2:
        for product in products:
            if product in room_types:
                room_discount += 0.1 * pricing[product]
    free_breakfasts = min(breakfasts, suites)
    free_parkings = min(parkings, suites + executives)
    discount = (
      free_breakfasts * pricing["breakfast"] +
      free_parkings * pricing["parking"] +
      room_discount)
    return total, discount, total - discount

# examples
print( booking(milton_hotel, [ # no discounts
    "executive", "breakfast"
]))
print( booking(milton_hotel, [ # one free parking
    "standard", "executive", "breakfast", "breakfast", "parking"
]))
print( booking(milton_hotel, [ # two free parking (one applied) and one free breakfast plus 10% off all rooms
    "standard", "executive", "suite", "parking",
    "breakfast", "breakfast", "breakfast"
]))
```

You have been tasked with writing a parser for a simple programming language. The language consists only of three kinds of expressions:

- Integer variable assignments, like `x = 2` or `abc=123`
- Function calls, like `f(x, abc)` or `do_stuff ()`
- Comments, which are anything following the first octothorpe (`#`) on a line

Implement a function `parse` that takes a string and returns three values:

- All variables as a dictionary, where the keys are variable names and the values are the values assigned to each variable. Make sure to convert the string numbers to integers!
- All function calls as a list of tuples, where each tuple has two elements: the first element is the name of the function being called and the second element is a list containing all the parameter names.
- All comments as a list of strings.

Observe the following implementation details:

- There will only be one variable assignment or function call per line
- There can be empty lines in the input string
- If a variable is re-declared, the old assignment is lost
- Function calls may pass zero parameters, in which case the corresponding parameter list in the output should be empty.
- In principle, variable names and function calls can contain any characters except whitespace, commas, `=`, `(` and `)`
- Whitespace around variable names, variable values, function names and function parameters should be stripped.
- For comments, the octothorpe (`#`) and any leading or trailing whitespace should be stripped
- Whitespace can be stripped using Python's `strip()` function. E.g.: `" abc ".strip()` is `"abc"`

Note: Make sure you return the correct number of return values. Partial points are awarded even if not all return values are correct, but only if all three values are present. If you are unable to parse all kinds of expressions, at least return those which you can.

You may assume that your function will only be called with parameters that match the given description.

Use the following template:

```python
def parse(program):
    pass # implement here
```

The following example illustrates how the solution function may be called:

```python
print( parse("""x = 1""") )
print( parse("""  y=2  """) )
print( parse("""fun()""") )
print( parse("""do_stuff (abc)""") )
print( parse("""# comment   """) )
print( parse("""
x = 1
y=   2
y=   3       # y=2 is now lost

fun( x,   y) #a comment after a #, including this last part: fun(x)
fun(a,abc)
"""))
```

The calls above should produce the following output:

```
({'x': 1}, [], [])
```

```
({'y': 2}, [], [])
({}, [('fun', [])], [])
({}, [('do_stuff', ['abc'])], [])
({}, [], ['comment'])
({'x': 1, 'y': 3}, [('fun', ['x', 'y']), ('fun', ['a', 'abc'])], ['y=2 is now lost', 'a comment after a #, including this last part: fun(x)'])
```

Paste your solution **including the function signature** into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit *only* your solution function to minimize the chance of errors.

Sample solution

```python
#!/usr/bin/env

def parse(program):
    variables = {}
    calls = []
    comments = []
    for line in program.splitlines():
        # extract comment
        octothorpe = line.find('#')
        if octothorpe != -1:
            comments.append(line[octothorpe+1:].strip())
            line = line[:octothorpe]
        # detect variable declaration
        assignment = line.find('=')
        if assignment != -1:
            var, value = line.split("=")
            variables[var.strip()] = int(value.strip())
        # detect function call
        fun = line.find('(')
        if fun != -1:
            name, rest = line.split("(")
            name = name.strip()
            rest = rest.strip()
            rest = rest[:-1].strip() # remove ")"
            parameters = rest.split(",")
            clean_parameters = []
            for var in parameters:
                var = var.strip()
                if var != "":
                    clean_parameters.append(var.strip())
            calls.append((name, clean_parameters))

    return variables, calls, comments

# examples
print( parse("""x = 1""") )
print( parse("""  y=2  """) )
print( parse("""fun()""") )
print( parse("""do_stuff (abc)""") )
print( parse("""# comment   """) )
print( parse("""
x = 1
y=   2
y=   3       # y=2 is now lost

fun( x,   y) #a comment after a #, including this last part: fun(x)
fun(a,abc)
"""))
```