Classes - 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.

Instances of a class hold references to attributes of that class.: True
Classes are objects.: True
Not all classes can be instantiated.: True
A class without any methods is abstract.: False

Class attributes are referenced by both the class and its instances.: True
A class is an object.: True
All classes can be instantiated.: False
An abstract class cannot be inherited from.: False

If one instance of a class modifies a class attribute, it will also appear modified to other instances of that class.: True
Classes are not objects.: False
Some classes cannot be instantiated.: True
A class is abstract if it has at least one abstract method.: True

---

Exceptions - 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.

If a function call raises an exception, it does not produce a return value.: True
A 'finally' block is only executed if an exception is raised in the preceding 'try' block.: False
A 'try' block must always be followed by an 'except' block.: False
When using an 'except' block without specifying an exception type, no exceptions are caught.: False

---

If an uncaught exception is raised by a function, the function execution ends immediately.: True
As a general rule, 'try' blocks should be as short as possible.: True
A 'try' block can be followed by multiple 'except' blocks.: True
Catching all exceptions using 'except' without specifying exception types is not recommended.: True

---

An exception in Python is always fatal and will stop the execution of the program, regardless of whether it is handled or not.: False
The 'else' block is only executed if an exception is raised in the preceding 'try' block.: False
An exception can be raised using the 'throw' keyword.: False
When using an 'except' block without specifying an exception type, all exceptions are caught.: True

---

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.

All functions take at least one argument.: False
Functions inherit from object.: True
A function that has no return statements does not return anything, ever.: False
Functions can return functions.: True

A function can take positional or keyword arguments, but not both at the same time.: False
Functions aren't objects.: False
All functions return something (assuming no exceptions occur).: True
Functions can take functions as arguments.: True

Some functions don't take any arguments.: True
Functions are objects.: True
Some functions don't return anything (even if no exceptions occur).: False
Functions can be passed as arguments to other functions.: True

---

Hashing - 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.

When two equal objects have different hashes, that's called a hash collision.: False
When implementing __hash__, one must also implement __eq__.: True
Mutable objects must implement __hash__.: True
Immutable objects can be used as dictionary keys.: True

Two objects that are equal must have the same hash (if any).: True
When implementing __hash__, one must also implement the six rich comparison operators.: False
Calling hash(...) on an object should always return the same value (within the same program execution).: False
Dictionary keys must be mutable.: False

Two objects that have the same hash must be equal.: False
When implementing __eq__, one must also implement __hash__.: False
If two objects have different hashes, then we know they are not equal.: True
Mutable objects cannot be used as dictionary keys.: True

Functions - 6 points, 1 out of 2 tasks

Implement a class `Data` which takes a list of objects as the only constructor argument and refers to it using a public attribute `data`.

`Data` should provide a method `compute` which takes a function as the only parameter. `compute` should call the given function on each element in `data` and return the resulting values in a list (in the same order as the elements in `data`).

The `Data` class should work with arbitrary data types, as long as any functions passed are compatible with whatever `data` is stored.

Use the following template:

```
class Data:
    pass
```

The following example illustrates how the solution may be used:

```
objects = [1.7, 2.3, 3, 4]
d = Data(objects)
print( d.data is objects )
print( d.compute(str) )
import math
print( d.compute(math.floor) )
```

The code above should produce the following output:

```
True
['1.7', '2.3', '3', '4']
[1, 2, 3, 4]
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

class Data:
    def __init__(self, data):
        self.data = data

    def compute(self, fun):
        return [fun(d) for d in self.data]

# examples
objects = [1.7, 2.3, 3, 4]
d = Data(objects)
print( d.data is objects )
print( d.compute(str) )
import math
print( d.compute(math.floor) )
```

Implement a class `Data` which takes a list of objects as the only constructor argument and refers to it using a public attribute `data`.

`Data` should provide a method `compute` which takes two functions as arguments. `compute` should call the given functions with the `Data` object's `data` as the argument and return the two resulting values in a tuple.

The `Data` class should work with arbitrary data types, as long as any functions passed are compatible with whatever `data` is stored.

Use the following template:

```
class Data:
    pass
```

The following example illustrates how the solution may be used:

```
objects = [1.7, 2.3, 3, 4]
d = Data(objects)
print( d.data is objects )
print( d.compute(sum, min) )
print( d.compute(min, max) )
```

The code above should produce the following output:

```
True
(11.0, 1.7)
(1.7, 4)
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

class Data:
    def __init__(self, data):
        self.data = data

    def compute(self, fun1, fun2):
        return (fun1(self.data), fun2(self.data))

# examples
objects = [1.7, 2.3, 3, 4]
d = Data(objects)
print( d.data is objects )
print( d.compute(sum, min) )
print( d.compute(min, max) )
```

Data Structures - 7 points, 1 out of 2 tasks

Implement a function `merge_dicts` which takes a list of dictionaries `dicts` as a positional argument, plus a keyword argument `reverse_priority`, which should be `False` by default.

The function should return a dictionary resulting from merging all the dictionaries in `dicts`. In other words, the resulting dictionary should contain all items of all dictionaries in `dicts`. If multiple dictionaries in `dicts` contain the same key, then the value contained in the last dictionary containing that key (in terms of order within `dicts`) should take precedence, unless `reverse_priority` is `True`, in which case, the first dictionary containing that key should be used.

Note that `merge_dicts` should work no matter how many dictionaries are contained in `dicts`.

Use the following template:

```
def merge_dicts():
    pass
```

The following example illustrates how the solution may be used:

```
d1 = {1: "a", 2: "b", 3: "c"}
d2 = {1: 1, 20: 2, 300: 3}
d3 = {1: "please", 2: "send", 300: "help"}
print(merge_dicts([d1, d2, d3]))
print(merge_dicts([d1, d2, d3], True))
```

The code above should produce the following output:

```
{1: 'please', 2: 'send', 3: 'c', 20: 2, 300: 'help'}
{1: 'a', 2: 'b', 300: 3, 20: 2, 3: 'c'}
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

def merge_dicts(dicts, reverse_priority=False):
    res = {}
    if reverse_priority:
        dicts = reversed(dicts)
    for d in dicts:
        for k, v in d.items():
```

```
            res[k] = v
    return res

# or
def merge_dicts(dicts, reverse_priority=False):
    res = {}
    for d in dicts:
        if reverse_priority:
            res = d | res
        else:
            res = res | d
    return res


# examples
d1 = {1: "a", 2: "b", 3: "c"}
d2 = {1: 1, 20: 2, 300: 3}
d3 = {1: "please", 2: "send", 300: "help"}
print(merge_dicts([d1, d2, d3]))
print(merge_dicts([d1, d2, d3], True))
```

Implement a function `padded_zip` which takes a list of lists `lists` as a positional argument, plus a keyword argument `padding`, which should be `None` by default.

The function should return a tuple of tuples where the i-th tuple contains the i-th element from each of the lists contained in `lists`. If some of the lists in the input are shorter, `padding` should be used in place of further elements. Note that `padded_zip` should work no matter how many lists are contained in `lists`.

The functionality is similar to Python's built-in `zip` function, however `padded_zip` should return a tuple of the same length as the longest input list, and it only strictly needs to work on lists of lists.

Use the following template:

```
def padded_zip():
    pass
```

The following example illustrates how the solution may be used:

```
l1 = [1, 2, 3, 4, 5]
l2 = [1.5, 2.5, 3.5, 4.5]
l3 = ["please", "send", "help"]
print(padded_zip([l1, l2, l3]))
print(padded_zip([l1, l2, l3, l1], "!"))
```

The code above should produce the following output:

```
((1, 1.5, 'please'), (2, 2.5, 'send'), (3, 3.5, 'help'), (4, 4.5, None), (5, None, None))
((1, 1.5, 'please', 1), (2, 2.5, 'send', 2), (3, 3.5, 'help', 3), (4, 4.5, '!', 4), (5, '!', '!', 5))
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

def padded_zip(lists, padding=None):
    if not lists: return ()
    res = []
    for i in range(max(len(l) for l in lists)):
        this = []
        for l in lists:
            try:                        # if i < len(l):
                this.append(l[i])
            except IndexError:          # else:
                this.append(padding)
        res.append(tuple(this))
    return tuple(res)

# or
def padded_zip(lists, padding=None):
    if not lists: return ()
    res = []
    for i in range(max(len(l) for l in lists)):
        res.append(tuple(l[i] if i < len(l) else padding for l in lists))
    return tuple(res)

# or simply
def padded_zip(lists, padding=None):
    from itertools import zip_longest
    return tuple(zip_longest(*lists, fillvalue=padding))
```

```
# examples
l1 = [1, 2, 3, 4, 5]
l2 = [1.5, 2.5, 3.5, 4.5]
l3 = ["please", "send", "help"]
print(padded_zip([l1, l2, l3]))
print(padded_zip([l1, l2, l3, l1], "!"))
```

Recursive Data Structures - 12 points, 1 out of 2 tasks

Implement a class `BinaryTree`, which represents a tree-like, recursive data structure.

A `BinaryTree` is initialized with one positional argument `key`, which can be of arbitrary type, a keyword argument `value`, which is a number (`0` by default), and two more keyword arguments `left` and `right`, each of which is another `BinaryTree` (or `None` by default).

`BinaryTree` should have a method `tally` which takes an argument `needle` of arbitrary type. `tally` should recursively traverse the `BinaryTree` data structure and eventually return a kind-of-sum of all `value`s in the tree, *adding* those `value`s where `needle` matches `key`, and *subtracting* all others. Besides comparing `needle` to the tree's `key`, `tally` will also need to call the `left` and `right` `BinaryTree`'s `tally` methods recursively to add their return values to the result.

In the example below, a tree consisting of eight `BinaryTree` instances is constructed. `tree.tally("C")` and `tree.tally("D")` are called on the root (outermost) `BinaryTree` and return the kind-of-sum according to the rule outlined above.

To receive partial points, make sure that `BinaryTree` can at least be instantiated correctly as per the example. Use *public* attribute names that equal the specified constructor argument names.

Use the following template:

```
class BinaryTree:
    pass
```

The following example illustrates how the solution may be used:

```
tree = BinaryTree("root",
    left = BinaryTree("A",
        left = BinaryTree("B", 20,
            left = BinaryTree("C", 60),
            right = BinaryTree("D", -50)
        ),
        right = BinaryTree("E",
            right = BinaryTree("D", 70)
        )
    ),
    right = BinaryTree("D", 80)
)
print(tree.tally("C"))  # - 0 - 0 - 20 + 60 - -50 - 0 - 70 - 80
print(tree.tally("D"))  # - 0 - 0 - 20 - 60 + -50 - 0 + 70 + 80
```

The code above should produce the following output:

```
-60
20
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

class BinaryTree:
    def __init__(self, key, value=0, left=None, right=None):
        self.key = key
        self.value = value
        self.left = left
        self.right = right

    def tally(self, needle):
        res = self.value if self.key == needle else -self.value
        if self.left != None:
            res += self.left.tally(needle)
        if self.right != None:
            res += self.right.tally(needle)
        return res

# examples
tree = BinaryTree("root",
    left = BinaryTree("A",
```

```
        left = BinaryTree("B", 20,
            left = BinaryTree("C", 60),
            right = BinaryTree("D", -50)
        ),
        right = BinaryTree("E",
            right = BinaryTree("D", 70)
        )
    ),
    right = BinaryTree("D", 80)
)
print(tree.tally("C"))  # - 0 - 0 - 20 + 60 - -50 - 0 - 70 - 80
print(tree.tally("D"))  # - 0 - 0 - 20 - 60 + -50 - 0 + 70 + 80
```

Implement a class `BinaryTree`, which represents a tree-like, recursive data structure.

A `BinaryTree` is initialized with two positional arguments `key` and `data`, which can be of arbitrary type, and two keyword arguments `left` and `right`, each of which is another `BinaryTree` (or `None` by default).

`BinaryTree` should have a method `find` which takes an argument `needle` of arbitrary type. `find` should recursively traverse the `BinaryTree` data structure and return a list of all `data` where `needle` matched `key`. More specifically, on a given `BinaryTree`, `find` should first compare `needle` to the tree's key and if it matches, the tree's data should be the first element of the list to be returned. Then, `find` should call the `left` and `right` `BinaryTree`'s `find` methods recursively, and add their return values to the result to finally return the resulting list.

Naturally, `find` should return an empty list if no matching keys are found.

In the example below, a tree consisting of eight `BinaryTree` instances is constructed. `tree.find("X")` and `tree.find(200)` are called on the root (outermost) `BinaryTree`, which, in both cases, retrieves all `data` where the `key` matches `needle`.

Make sure your implementation returns a list where results are ordered correctly.

To receive partial points, make sure that `BinaryTree` can at least be instantiated correctly as per the example. Use *public* attribute names that equal the specified constructor argument names.

Use the following template:

```
class BinaryTree:
    pass
```

The following example illustrates how the solution may be used:

```
tree = BinaryTree("root", "Tree root",
    left = BinaryTree("X", "Anna",
        left = BinaryTree(123, "Betty",
            left = BinaryTree("C", "Charles"),
            right = BinaryTree(200, "Dora")
        ),
        right = BinaryTree("E", "Emil",
            right = BinaryTree(200, "Denis")
        )
    ),
    right = BinaryTree(200, "Daniel")
)
print(tree.find("X"))
print(tree.find(200))
```

The code above should produce the following output:

```
['Anna']
['Dora', 'Denis', 'Daniel']
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

class BinaryTree:
    def __init__(self, key, data, left=None, right=None):
        self.key = key
        self.data = data
        self.left = left
        self.right = right

    def find(self, needle):
        res = [self.data] if self.key == needle else []
        if self.left != None:
            res.extend(self.left.find(needle))
        if self.right != None:
```

```
            res.extend(self.right.find(needle))
        return res

# examples
tree = BinaryTree("root", "Tree root",
    left = BinaryTree("X", "Anna",
        left = BinaryTree(123, "Betty",
            left = BinaryTree("C", "Charles"),
            right = BinaryTree(200, "Dora")
        ),
        right = BinaryTree("E", "Emil",
            right = BinaryTree(200, "Denis")
        )
    ),
    right = BinaryTree(200, "Daniel")
)
print(tree.find("X"))
print(tree.find(200))
```

Data Model - 8 points, 1 out of 1 task

In the lecture, you learned how Python's data model allows you to supports functionality like `==`, `<=` or `hash(...)`, via methods such as `__eq__`, `__lt__` or `__hash__`. In this task, you will apply this knowledge to implement a class `StringDict`, which largely functions just like a regular dictionary, but coerces all stored values into strings.

The example below shows how after instantiating a `StringDict`, one can set and retrieve values just like with a regular dictionary. However, whenever a value is set, it is immediately converted to a string.

`StringDict` should at least support the following features:

- setting a value via the bracket notation: `d[key] = value`
- retrieving a value via the bracket notation: `d[key]`
- printing: `print(d)`, which should appear the same way as when printing a regular dictionary
- printing in a collection: `print([d])`, which should appear the same way as when printing a regular dictionary in a collection
- getting its size: `len(d)`
- the `in` operator: `123 in d`, which, just like with regular dictionaries, checks if a key is present in the `StringDict`
- equality: `d1 == d2` and `d1 != d2`
- providing an optional, regular dictionary upon instantiation; don't forget to convert the values to strings.

It is up to you how you make this happen. `StringDict` does not *need to* support additional functionality that would be supported by regular dictionaries (such as `del(...)`, `iter(...)`, etc.).

Hint: The requirements above will make use of the following methods of Python's data model: `__init__`, `__len__`, `__eq__`, `__str__`, `__repr__`, `__setitem__`, `__getitem__`, `__contains__`.

Hint: Remeber that `dict` is a class like any other.

Important: Any instance members that aren't methods should be private.

Use the following template:

```
class StringDict:
    pass
```

The following example illustrates how the solution may be used:

```
d1 = StringDict()
print(d1)
d1[1] = 100
d1["abc"] = print
print(d1)
print(isinstance(d1[1], str))
print(1 in d1)
print(100 in d1)
print(len(d1))
d2 = StringDict({"abc": print, 1: "100"})
d3 = StringDict({"abc": max})
print(d1 == d2)
print(d1 == d3)
```

The code above should produce the following output:

```
{}
{1: '100', 'abc': '<built-in function print>'}
True
True
False
```

```
2
True
False
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

class StringDict:
    def __init__(self, elements=None):
        self.__elements = {}
        if elements:
            for k, v in elements.items():
                self.__elements[k] = str(v)

    def __len__(self):
        return len(self.__elements)

    def __eq__(self, other):
        return self.__elements == other.__elements

    def __setitem__(self, key, value):
        self.__elements[key] = str(value)

    def __getitem__(self, key):
        return self.__elements[key]

    def __str__(self):
        return str(self.__elements)

    def __repr__(self):
        return repr(self.__elements)

    def __contains__(self, thing):
        return thing in self.__elements

# or simply:
class StringDict(dict):
    def __init__(self, elements=None):
        if elements:
            for k, v in elements.items():
                elements[k] = str(v)
            super().__init__(elements)
        else:
            super().__init__()


    def __setitem__(self, key, value):
        super().__setitem__(key, str(value))

# examples
d1 = StringDict()
print(d1)
d1[1] = 100
d1["abc"] = print
print(d1)
print(isinstance(d1[1], str))
print(1 in d1)
print(100 in d1)
print(len(d1))
d2 = StringDict({"abc": print, 1: "100"})
d3 = StringDict({"abc": max})
print(d1 == d2)
print(d1 == d3)
```

White-box Testing - 15 points, 1 out of 1 task

You are given a function `process`. Implement a test class which ensures that any alternative implementations would exhibit the same behavior as `process`.

You should assume that `process` is in scope, just like in the template. You do not need to import `process`. Your solution will be graded based on how well it can recognize potential bugs in alternative implementations. Make sure you test edge cases for each of the parameters where appropriate.

Important: Use the provided template and do not change the name of the test class.

Important: You must hand in your test class, including the signature. Handing-in `process` is unnecessary.

Use the following template:

```
def process(string, needle, mode="remove", character=None):
    if len(needle) < 1:
        raise ValueError
    if mode not in ["remove", "replace"]:
        raise NameError
    if mode == "replace" and character is None:
        raise TypeError
    if mode == "replace":
        return string.replace(needle, character)
    if mode == "remove":
        return string.replace(needle, "")

# examples of how process can be used:
print( process("abcd_abcd", "b", "remove") )       # acd_acd
print( process("abcd_abcd", "b", "replace", "*") ) # a*cd_a*cd

from unittest import TestCase

class Tests(TestCase):
    pass # implement here
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

def process(string, needle, mode="remove", character=None):
    if not isinstance(string, str) or \
       not isinstance(needle, str):
       raise TypeError
    if len(needle) < 1:
        raise ValueError
    if mode not in ["remove", "replace"]:
        raise NameError
    if mode == "replace" and character is None:
        raise LookupError
    if mode == "replace":
        return string.replace(needle, character)
    if mode == "remove":
        return string.replace(needle, "")

from unittest import TestCase

class Tests(TestCase):
    def test_input_non_string(self):
        with self.assertRaises(TypeError):
            process(2, "xyz")

    def test_needle_non_string(self):
        with self.assertRaises(TypeError):
            process("xyz", 2)

    def test_needle_empty(self):
        with self.assertRaises(ValueError):
            process("xyz", "")

    def test_mode_invalid(self):
        with self.assertRaises(NameError):
            process("abc", "xyz", "nonsense")

    def test_replace_without_replacement(self):
        with self.assertRaises(LookupError):
            process("abc", "xyz", "replace")

    def test_remove_from_empty(self):
        self.assertEqual(process("", "xy"), "")

    def test_replaces_only_first(self):
        self.assertEqual(process("here's another sentence", "e", "replace", "%"),
        "h%r%'s anoth%r s%nt%nc%")

    def test_removes_only_first(self):
        self.assertEqual(process("here's another sentence", "e"),
        "hr's anothr sntnc")

    def test_ok_with_longer_needle(self):
        self.assertEqual(process("here's another sentence", "oth"),
        "here's aner sentence")

    def test_remove_to_empty_full(self):
        self.assertEqual(process("xxx", "xxx"), "")

    def test_remove1(self):
```

```
        self.assertEqual(process("xyzzyx", "yz"), "xzyx")

    def test_remove2(self):
        self.assertEqual(process("this is a sentence is it not yes it is", "it "),
        "this is a sentence is not yes is")

    def test_replace_from_empty(self):
        self.assertEqual(process("", "xy", "replace", "zz"), "")

    def test_replace_full(self):
        self.assertEqual(process("here's another sentence", "en", "replace", "%%"),
        "here's another s%%t%%ce")
```

Programming - 15 points, 1 out of 1 task

You are given a class `Canvas` which represents a rectangular drawing area. Read the source code and comments provided in the code template to understand how.

Then, implement the `draw` method, which essentially moves a "pencil" across the canvas, drawing a sequence of adjacent pixels one by one.

`draw` takes four parameters:

- `x` a starting x (horizontal) coordinate
- `y` a starting y (vertical) coordinate
- `path` a string containing a sequence of up (`"u"`), down (`"d"`), left (`"l"`), and right (`"r"`) movements represented by the given characters (`""` by default)
- `char` a single character used to paint on the canvas (`"â–ˆ"` by default)

`draw` first places the "pencil" at the provided `x`/`y` coordinates and draws `char` at that location. Then, it goes through the movement instructions one by one to update the pencil coordinates. Thus, if `path` is the empty string, only a single pixel is drawn (at the `x`/`y` coordinates). In the general case, `len(path)+1` pixels are drawn. The top-left-most pixel is at coordinates `0`/`0`.

With regard to possible arguments to `draw`, `draw` should be generous in what it accepts. It should...

- ...not care about casing ("d" vs. "D") in the `path` string
- ...silently ignore `path` characters which are not valid movement instructions, rather than fail
- ...wrap around the edges of the canvas if the pencil is placed or moved out of bounds, rather than fail
- ...allow over-writing pixels multiple times

On the other hand, `draw` should raise an `Exception` if...

- ...`x` or `y` is not an integer
- ...`path` is not a string
- ...`char` is not a string or has any length other than `1`

You must implement `draw`. You may add additional methods to the class if you want; but you don't have to.

Important: Do **not** modify the already implementated methods of `Canvas`!

Important: Do **not** add any additional top-level definitions (except for imports, if you need any). Your solution must be contained entirely inside `Canvas`!

Important: Do **not** modify the signature already provided for `draw`!

Important: Hand in your **entire** class definition, including the signature and the methods provided in the template! In case you need any imports, include them, too.

Use the following template:

```
#!/usr/bin/env

class Canvas:
    def __init__(self, width, height):
        # A canvas consists of *height* number of rows, so for example,
        # ... self.rows[0][0] refers to the top-left pixel
        # ... self.rows[3][5] refers to the 6th pixel on the fourth row
        self.rows = []
        for row in range(height):
            self.rows.append([" "] * width)
        # print(self.rows) given width=5 and height=2 would show:
        # [[' ', ' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ', ' ']]
        # and __str__ would return:
        #  -----
        # |     |
        # |     |
```

```
        #  -----
        # showing a canvas with two rows and 5 pixels in each row.
        # the border added by __str__ is decorative and not part of the canvas

    def __str__(self):
        # returns the canvas surrounded by a border
        return " " + "-" * len(self.rows[0]) + " \n|" + \
               "|\n|".join(''.join(row) for row in self.rows) + \
               "|\n " + "-" * len(self.rows[0]) + " "

    def draw(self, x, y, path="", char="â–ˆ"):
        # implement this method
        pass
```

The following example illustrates how the solution may be used:

```
c = Canvas(10, 6)
c.draw(0, 0, "rdr", "+")
print(c)
# ignore non-movement characters; the following path is equivalent to "rd"
# overwrites three of the + signs drawn before
c.draw(0, 0, "weird!")
# draw U shape in the top-right corner
c.draw(7, 0)
c.draw(9, 0)
c.draw(9, 1, "ll")
print(c)
# start in bottom right corner and wrap around the right border
c.draw(9, 5, "RRUR", ">")
# wrap around the left border
c.draw(2, 3, "LLLLL", "<")
# starting coordinates can also wrap around: (25, 25) is the same as (5, 1)
c.draw(25, 25, "uu", "^")
print(c)
```

The code above should produce the following output:

```
 ----------
|++        |
| ++       |
|          |
|          |
|          |
|          |
 ----------
 ----------
|â–ˆâ–ˆ     â–ˆâ–ˆ |
| â–ˆâ–ˆ+    â–ˆâ–ˆâ–ˆâ–ˆ|
|          |
|          |
|          |
|          |
 ----------
 ----------
|â–ˆâ–ˆ   ^ â–ˆâ–ˆ |
| â–ˆâ–ˆ+  ^ â–ˆâ–ˆâ–ˆâ–ˆ|
|          |
|<<<    <<<|
| >>       |
|>>   ^   >|
 ----------
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

class Canvas:
    def __init__(self, width, height):
        # A canvas consists of *height* number of rows, so for example,
        # ... self.rows[0][0] refers to the top-left pixel
        # ... self.rows[3][5] refers to the 6th pixel on the fourth row
        self.rows = []
        for row in range(height):
            self.rows.append([" "] * width)
        # print(self.rows) given width=5 and height=2 would show:
        # [[' ', ' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ', ' ']]
        # and __str__ would return:
        #  -----
        # |     |
        # |     |
        #  -----
        # showing a canvas with two rows and 5 pixels in each row.
        # the border added by __str__ is decorative and not part of the canvas
```

```
    def __str__(self):
        # returns the canvas surrounded by a border
        return " " + "-" * len(self.rows[0]) + " \n|" + \
               "|\n|".join(''.join(row) for row in self.rows) + \
               "|\n " + "-" * len(self.rows[0]) + " "

    def pixel(self, x, y, char):
        x = x % len(self.rows[0])
        y = y % len(self.rows)
        self.rows[y][x] = char

    def draw(self, x, y, path="", char="â–ˆ"):
        if not isinstance(x, int) or \
            not isinstance(y, int) or \
            not isinstance(path, str) or \
            not isinstance(char, str) or \
            len(char) != 1:
            raise Exception
        path = path.lower()
        self.pixel(x, y, char)
        for move in path:
            move = move
            if move == "u": y -= 1
            if move == "d": y += 1
            if move == "l": x -= 1
            if move == "r": x += 1
            self.pixel(x, y, char)

# examples
c = Canvas(10, 6)
c.draw(0, 0, "rdr", "+")
print(c)
# ignore non-movement characters; the following path is equivalent to "rd"
# overwrites three of the + signs drawn before
c.draw(0, 0, "weird!")
# draw U shape in the top-right corner
c.draw(7, 0)
c.draw(9, 0)
c.draw(9, 1, "ll")
print(c)
# start in bottom right corner and wrap around the right border
c.draw(9, 5, "RRUR", ">")
# wrap around the left border
c.draw(2, 3, "LLLLL", "<")
# starting coordinates can also wrap around: (25, 25) is the same as (5, 1)
c.draw(25, 25, "uu", "^")
print(c)
```

Inheritance - 19 points, 1 out of 2 tasks

Important: read the problem description, implementation notes and provided examples carefully for a comprehensive specification. All three parts are relevant for understanding the requirements! Also, make sure you do not run out of time to submit your solution!

Problem description

You are working for a restaurant chain and are asked to implement the salary accounting software for their chain of restaurants and staff.

The chain runs several restaurants, each of which has a name.

The chain employs different kinds of staff, but regardless of type, any staff works for a specific restaurant and has a unique employee number (`101` for the first employee, `102` for the second, and so on). Furthermore, each staff holds a record of all shifts worked in the past. When a shift is worked, the number of hours is added to the shift record. Each staff also provides a way to determine the total salary earned by the staff, which is calculated depending on the type of staff.

A cook receives the same salary, no matter how many hours are worked. A server receives a base salary plus an hourly salary for each hour worked. For dish washers, the salary is computed the same way as for servers, but with a 10% deduction.

Additional implementation instructions:

- Fill in the missing class implementations in the provided template. Do **not** add any other top-level definitions.
- Your classes *must* be compatible with the provided examples. Make sure their members are named appropriately.
- You must minimize code duplication! Call super-methods where possible to re-use functionality implemented in super-classes.

- Do **not** include any example calls in your submission. Submit only top-level class definitions and imports.

Use the following template:

```
from abc import ABC, abstractmethod

class Restaurant:
    pass

class Staff:
    pass

class Server:
    pass

class Dishwasher:
    pass

class Cook:
    pass
```

The following example illustrates how the solution may be used:

```
restaurant = Restaurant("Mc Ronalds")

cook = Cook(restaurant, 3200)
cook.work(10)
cook.work(50)
print(cook.number)
print(cook.shifts)
print(type(cook.shifts))
print(cook.salary())
print(cook)

server = Server(restaurant, 100, 9.50)
server.work(10)
server.work(50)
print(server.number)
print(server.shifts)
print(server.salary()) # (100 + 60*9.50)

washer = Dishwasher(restaurant, 100, 9.50)
washer.work(10)
washer.work(50)
print(washer.number)
print(washer.shifts)
print(washer.salary()) # (100 + 60*9.50) * 0.9
```

The code above should produce the following output:

```
101
[10, 50]
<class 'list'>
3200
Staff working for Mc Ronalds with salary 3200
102
[10, 50]
670.0
103
[10, 50]
603.0
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env python3

from abc import ABC, abstractmethod

class Restaurant:
    def __init__(self, name):
        self.name = name

class Staff(ABC):
    counter = 101
    def __init__(self, restaurant):
        self.restaurant = restaurant
        self.shifts = []
        self.number = Staff.counter
        Staff.counter += 1

    def work(self, hours):
        self.shifts.append(hours)

    @abstractmethod
```

```
    def salary(self):
        pass

    def __str__(self):
        return f"Staff working for {self.restaurant.name} with salary {self.salary()}"

class Server(Staff):
    def __init__(self, restaurant, base_salary, hourly_salary):
        super().__init__(restaurant)
        self.base_salary = base_salary
        self.hourly_salary = hourly_salary

    def salary(self):
        return self.base_salary + sum(self.shifts) * self.hourly_salary

class Dishwasher(Server):
    def salary(self):
        return super().salary() * 0.9

class Cook(Staff):
    def __init__(self, restaurant, base_salary):
        super().__init__(restaurant)
        self.base_salary = base_salary

    def salary(self):
        return self.base_salary

# examples
restaurant = Restaurant("Mc Ronalds")

cook = Cook(restaurant, 3200)
cook.work(10)
cook.work(50)
print(cook.number)
print(cook.shifts)
print(type(cook.shifts))
print(cook.salary())
print(cook)

server = Server(restaurant, 100, 9.50)
server.work(10)
server.work(50)
print(server.number)
print(server.shifts)
print(server.salary()) # (100 + 60*9.50)

washer = Dishwasher(restaurant, 100, 9.50)
washer.work(10)
washer.work(50)
print(washer.number)
print(washer.shifts)
print(washer.salary()) # (100 + 60*9.50) * 0.9
```

Important: read the problem description, implementation notes and provided examples carefully for a comprehensive specification. All three parts are relevant for understanding the requirements! Also, make sure you do not run out of time to submit your solution!

Problem description

You are working for a transportation company and are asked to implement the management software for their fleet of vehicles and drivers.

The company employs several drivers. Each driver has a name and a unique employee number (`1001` for the first employee, `1002` for the second, and so on).

Different kinds of transports are on offer, but regardless of type, a transport has a designated driver and holds a record of all trips made. When a trip is made, the number of kilometers travelled is added to the trip record. Each transport also provides a way to determine the total price, which is calculated depending on the type of transport.

A taxi has a trip fee, plus a small fee for each kilometer travelled. A discount taxi is the same, but its total calculated price is reduced by 20%. A shuttle is rented for a flat fee, no matter the number of trips or distance travelled.

Additional implementation instructions:

- Fill in the missing class implementations in the provided template. Do **not** add any other top-level definitions.
- Your classes *must* be compatible with the provided examples. Make sure their members are named appropriately.
- You must minimize code duplication! Call super-methods where possible to re-use functionality implemented in super-classes.

- Do **not** include any example calls in your submission. Submit only top-level class definitions and imports.

Use the following template:

```
from abc import ABC, abstractmethod

class Driver:
    pass

class Transport:
    pass

class Shuttle:
    pass

class Taxi:
    pass

class DiscountTaxi:
    pass
```

The following example illustrates how the solution may be used:

```
driver = Driver("Travis Bickle")
driver = Driver("Korben Dallas")
print(driver.number)

taxi = Taxi(driver, 2.00, 0.10)
taxi.trip(10)
taxi.trip(50)
print(taxi.trips)
print(type(taxi.trips))
print(taxi.total_cost()) # (2.00+10*0.10 + 2.00+50*0.10)
print(taxi)

dtaxi = DiscountTaxi(driver, 2.00, 0.10)
dtaxi.trip(10)
dtaxi.trip(50)
print(dtaxi.trips)
print(dtaxi.total_cost()) # (2.00+10*0.10 + 2.00+50*0.10) * 0.8

shuttle = Shuttle(driver, 250)
shuttle.trip(10)
shuttle.trip(50)
print(shuttle.trips)
print(shuttle.total_cost())
```

The code above should produce the following output:

```
1002
[10, 50]
<class 'list'>
10.0
Transport revenue: 10.0 (driver: Korben Dallas)
[10, 50]
8.0
[10, 50]
250
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env python3

from abc import ABC, abstractmethod

class Driver:
    counter = 1001
    def __init__(self, name):
        self.name = name
        self.number = Driver.counter
        Driver.counter += 1

class Transport(ABC):
    def __init__(self, driver):
        self.driver = driver
        self.trips = []

    def trip(self, km):
        self.trips.append(km)

    @abstractmethod
    def total_cost(self):
        pass
```

```
    def __str__(self):
        return f"Transport revenue: {self.total_cost()} (driver: {self.driver.name})"

class Taxi(Transport):
    def __init__(self, driver, trip_fee, km_fee):
        super().__init__(driver)
        self.trip_fee = trip_fee
        self.km_fee = km_fee

    def total_cost(self):
        return sum(self.trips) * self.km_fee + len(self.trips) * self.trip_fee

class DiscountTaxi(Taxi):
    def total_cost(self):
        return super().total_cost() * 0.8

class Shuttle(Transport):
    def __init__(self, driver, day_fee):
        super().__init__(driver)
        self.day_fee = day_fee

    def total_cost(self):
        return self.day_fee

# examples
driver = Driver("Travis Bickle")
driver = Driver("Korben Dallas")
print(driver.number)

taxi = Taxi(driver, 2.00, 0.10)
taxi.trip(10)
taxi.trip(50)
print(taxi.trips)
print(type(taxi.trips))
print(taxi.total_cost()) # (2.00+10*0.10 + 2.00+50*0.10)
print(taxi)

dtaxi = DiscountTaxi(driver, 2.00, 0.10)
dtaxi.trip(10)
dtaxi.trip(50)
print(dtaxi.trips)
print(dtaxi.total_cost()) # (2.00+10*0.10 + 2.00+50*0.10) * 0.8

shuttle = Shuttle(driver, 250)
shuttle.trip(10)
shuttle.trip(50)
print(shuttle.trips)
print(shuttle.total_cost())
```
