Worksheet #11B Tuples, Sets, and Dictionaries Answer Key =)

TRUE/FALSE

1. (11 points: 1 point each) The following are true/false questions. **Write either T or F in the boxes at the bottom of page 1.** If there's any counterexample, it's false.

   (a) If a list in Python contains multiple identical items and you try to convert it to a set, the program will crash.

   (b) If you try to access a key that does not exist in a dictionary, Python will return `None`.

   (c) Using curly braces {} creates an empty set.

   (d) The dictionary comprehension `{str(x):0 for x in range(1, 6)}` is equivalent to the dictionary `{"1":0, "2":0, "3":0, "4":0, "5":0}`.

   (e) The `"in"` operator can be used to check for the presence of an element in a tuple, but it doesn't work on sets because sets have no order.

   (f) The symmetric difference of two sets in Python can also be expressed as the union of the two sets, minus their intersection.

   (g) The `.pop()` method for dictionaries in Python removes the last added key-value pair of the dictionary and returns it.

   (h) When iterating over a dictionary using a `for` loop, the loop variable refers to the keys of the dictionary.

   (i) You can loop over sets, but you cannot loop over tuples.

   (j) Assuming `s1` and `s2` are both sets, the expression `s1.difference(s2)` can return a different result from `s2.difference(s1)`.

   (k) Both lists and tuples in Python support adding elements after creation, but unlike lists, elements in a tuple cannot be changed once added.

a	b	c	d	e	f	g	h	i	j	k
F	F	F	T	F	T	F	T	F	T	F

Notes:
(A) nope! The duplicates just won't be stored in the list.
(B) nope, this is a KeyError.
(C) nope, this makes an empty dictionary. To make an empty set, use set().

(G) Nope, .pop on dictionaries requires specifying a key. .pop will remove the item with that key from the dictionary and return the value.
(J) Yes! Because s1.difference(s2) gets you the items in s1, that are not in s2. It removes the items in s1 that are in s2. Likewise, s2.difference(s1) filters out the items in s2 that are also in s1.
(K) nope! Tuples are immutable, so you can't add to a tuple without creating a new tuple (unlike lists).

MULTIPLE CHOICE

2	3	4	5	6	7	8	9	10	11	12
F	B	B	B	E	E	A	E	A	D	B

2. Which of the following is a valid way to create a tuple containing the integer 1?

A. `harry = 1,`
B. `liam = (1,)`
C. `niall = (1)`
D. `louis = {1,}`
E. All of the above.
F. A and B

Correct answer explanation:
(F) Yes! Both A and B will create a tuple. To create a tuple in Python, all that is needed is a comma and in some cases, parentheses, to get Python to interpret correctly. For example, if you're trying to append a tuple to a list:

myLst.append(1, 1)

This will cause an error, because Python interprets each 1 as a separate parameter. But if we do

myLst.append((1, 1))

Then Python will interpret this as a tuple, and won't crash.

But in most cases, all we need is a comma grouping our items together (or at the end to signify a tuple). And! We can't have brackets reserved for other data types – we can't have [] or {}. (A) and (B) both have a comma at the end, and avoid using [] and {}, so Python will interpret them as tuples.

Wrong answer explanations:
(C) Python will assign the value 1 to 'niall', so 'niall' will be an int.
(D) {1,} simply creates a set containing 1 – not a tuple!

3. Which approach will correctly delete key/value pairs from a dictionary (`myDict`) where the value is less than some threshold `x`?

A.
```
for value in myDict:
    if not value >= x:
        del value
```

B.
```
myKeys = list(myDict.keys())
for key in myKeys:
    if myDict[key] < x:
        del myDict[key]
```

C.
```
for i in range(len(myDict)):
    if myDict[i] < x:
        myDict.pop(i)
```

D.
```
removeKeys = [val for key, val in myDict.items() if val < x]
for key in removeKeys:
    del myDict[key]
```

Correct answer explanation:
(B) We create a list of our dictionary keys by doing list(myDict.keys()). Then, we loop over our list. We do 'for key in myKeys', and provide the key to our dictionary in a conditional: 'if myDict[key] < x:'. This is accessing the value and comparing it to x. If it is less than, then we do del myDict[key], which properly specifies a key/value pair to delete.

Wrong answer explanations:
(A) When we iterate over our dictionary by doing 'for varName in myDict', varName will always refer to the keys in the dictionary. Even if we name it 'value', which I did here to throw you kids off, it will refer to the keys in the dictionary. So, when we do 'if not value >= x', we'd really just be comparing the keys to x, and not the values to x. And 'del value' would be trying to delete by using strictly the key instead of accessing the value, which is incorrect.

(C) by doing 'for i in range(len(myDict))', we create a loop that lasts for as many keys that we have in the dictionary. But it's not guaranteed that our keys are numbers matching i. Our keys could be strings, for instance. So, doing 'if myDict[i] < x' is not guaranteed to work.

(D) We use list comprehension: [val for key, val in myDict.items() if val < x]

But note that, if we reorder this:

```
[ for key, val in myDicts.items()
        if val < x
                val
]
```

Our list will contain the values in our dictionary, not the keys. So when we do

for key in removeKeys:

The loop variable key refers to the items in our removeKeys list, which are the values from our dictionary. So, 'key' doesn't refer to dictionary keys.

So when we try doing 'del myDict[key]', we're trying to access values in our dictionary, by providing a value. This probably won't work. LOL

4. If `s1` and `s2` are both sets, and `s1.issuperset(s2)` is `False`, which of the following expressions will always evaluate to `True`?

   A. `s1.difference(s2) == s1`
   B. `s1.symmetric_difference(s2) != set()`
   C. `s1.intersection(s2) == set()`
   D. `len(s2) >= len(s1)`

Correct answer explanation:
(B) If s1.issuperset(s2) is False, then we know that s2 must contain at least one item that is not in s1. So, because there is an item in one set, but not the other, s1.symmetric_difference(s2) cannot be equal to the empty set (which is what set() is).

Wrong answer explanations:
(A) .difference means, removing from the first set, the items that are present in the second set. So, s1.difference(s2) == s1 would imply that s1 and s2 have no items in common. But they can share items, even if s1.issuperset(s2) is False. Read next answer explanation for more info =)
(C) We only know that s2 contains at least one item that s1 does not. This does not mean that the two sets share /no/ items. Only that s2 has an item absent from s1. Ex:

s1 = {1, 2, 3}
s2 = {3, 4}

s1.issuperset(s2) will be False, because s1 does not contain 4. But s1.intersection(s2) will be {3}, because both sets have 3 in them.

(D) superset doesn't necessarily have to do with length. In the example for (C), s1.issuperset(s2) is False, but it still has more elements than s2. So, it's not guaranteed that s2 would have greater length.

5. What will happen if you try to convert a dictionary into a list using `list()`?

   A. An error is raised.
   B. It will create a list of the dictionary's keys.
   C. It will create a list of the dictionary's values.
   D. It will create a list of the dictionary's key-value pairs as tuples.

Correct answer explanation:
(B) Yes! If my dictionary is called myDict, then list(myDict) will return a list of the keys present in myDict. Similar to how, when we loop over a dictionary, our loop variable will refer to the keys in the dictionary (not the values).

Wrong answer explanations:
(A) False! We can convert dictionaries to lists.
(C) False! It will create a list of the dictionary's keys, not values.
(D) False! To get a list of the dictionary's key-value pairs, we would need to do myDictionaryName.items(). Well, actually, I think .items(), .keys() and .values() return something slightlyyyy different from a list, but they behave very similarly. For the scope of this class: don't worry about it. LOL. Unless you really want to know, in which case make an Ed thread! =)

6. What can the `tuple()` constructor be used for?

   A. Creating an empty tuple.
   B. Converting a single number into a one-item tuple.
   C. Converting other types such as strings or lists to tuples.
   D. All of the above.
   E. A and C.

Correct answer explanation:
(E) Yes! (A) and (C) are both correct. If we do x = tuple(), then x will be an empty tuple. And if we have x = [1, 2], and we do x = tuple(x), then x will be (1, 2). If x is "howdy", then tuple(x) gets us ("h", "o", "w", "d", "y").

Wrong answer explanations:
(B) The tuple() parameter surprisingly cannot be used on a single number. If we try doing tuple(1), Python will raise an error. So, tuple() only takes objects that are 'iterable' – that is to say, if we couldn't loop over the object, then tuple() would not accept it. We can loop over strings and lists, but we can't loop over a single number.
(D) Wrong because (B) is wrong.

7. Assuming `set1` and `set2` are both sets, how can you obtain a set containing only the elements present in one (not both)?

   A. `set1.difference(set2).union(set2.difference(set1))`
   B. `set1.symmetric_difference(set2)`
   C. `set1.union(set2).difference(set1.intersection(set2))`
   D. `set1.difference(set2).union(set2 - set1)`
   E. All of the above.
   F. A and C.

Correct answer choice:
(E) A, B, C, and D all work! This is because the symmetric difference of two sets is defined as the set of elements that are in either set, but not in the intersection.

So, clearly, (B) should work, just using symmetric_difference().

(A) works because it obtains the items exclusively in set1, by doing set1.difference(set2) and adds (union) them with the items exclusively in set2 (set2.difference(set1)).

(D) is similar to (A), except that for the second difference, we use the minus sign. But this still works. It's similar to lists. You can use .append(), or add two lists using +. Here, with sets, instead of using .difference(), we're just using -.

(C) works because it's the definition of symmetric difference. We union the sets to get all items, but we call .difference() to remove those in the intersection.

8. What is possible output of the following code?

```
pleakley = {"lilo", "stitch"}
jumba = pleakley
jumba.add("nani")
bubbles = set(pleakley)
bubbles.remove("stitch")
print(pleakley, jumba, bubbles)
```

A. {"nani", "stitch", "lilo"} {"nani", "stitch", "lilo"} {"nani", "lilo"}
B. {"stitch", "lilo"} {"nani", "stitch", "lilo"} {"lilo"}
C. {"stitch", "lilo"} {"nani", "stitch", "lilo"} {"nani", "lilo"}
D. {"nani", "stitch", "lilo"} {"nani", "stitch", "lilo"} {"lilo"}
E. None of the above.

Correct answer choice:

(A) We start with pleakley = {"lilo", "stitch"}. When we set jumba = pleakley, this doesn't necessarily duplicate pleakley. jumba and pleakley will refer to the exact same object in memory, because Python variables are essentially labels or names that refer to objects in memory. So when we assign jumba = pleakley, both jumba and pleakley will refer to the same object.

Thus, after jumba.add("nani"), jumba and pleakley refer to a set containing {"lilo", "stitch", "nani"}.

We set bubbles = set(pleakley) – this will create a distinct object in memory, because we are directly invoking the set constructor to create a new set, from the items in pleakley. So, bubbles will be a new set containing {"lilo", "stitch", "nani"}.

We then do bubbles.remove("stitch"), which means bubbles is now {"lilo", "nani"}.

So, when we print(pleakley, jumba, bubbles), we should display two sets consisting of "lilo", "stitch" and "nani", and one set consisting of only "lilo" and "nani". (A) is the only answer that matches this.

9. Which of the following correctly demonstrates tuple unpacking in Python?

A. `dreamland, eatsAlot = ("kirby", "dedede", "metaKnight")`
B. `(arcanine, chansey) = "jigglypuff", "jynx", "clefairy"`
C. `blathers, ableSisters = ("isabelle", "tomNook"), ("rover", "kkSlider")`
D. `courage, (wisdom, power) = ("link", ("zelda", "ganon"))`
E. C and D.
F. B, C and D.

Correct answer explanation:
(E) OOOHHHH! This is a tricky question. I was evil for this.

OK. Let's analyze (C) first. C is

blathers, ableSisters = ("isabelle", "tomNook"), ("rover", "kkSlider").

On the right side of this code, we're creating two tuples – one tuple is ("isabelle", "tomNook") and the other is ("rover", "kkSlider"). And we're assigning them each to a variable on the left side. The first tuple is stored as blathers, and the second is stored as ableSisters. Because we have two variables and two objects, this is a valid expression.

(D) is tricky. On the right side of this code, we have a tuple ("link", ("zelda", "ganon")). For clarity, the first item in this tuple is string "link", and the second item in this tuple is another tuple, consisting of "zelda" and "ganon".

So, if we wanted to unpack this whole tuple on the right hand side, we need to store "link" in its own variable, and the items in the inner tuple in two other variables.

This is what courage, (wisdom, power) is doing. "link" is being assigned to courage, whereas (wisdom, power) refers to the tuple ("zelda", "ganon"). We're creating variables for each item present on the right hand side.

What's interesting – we could have done

courage, wisdomPower = ("link", ("zelda", "ganon"))

wisdomPower would refer to the tuple ("zelda", "ganon"). But, in this answer choice, since we do

courage, (wisdom, power) = (“link”, (“zelda”, “ganon”))

We’re unpacking the right tuple, to get “link” into courage, but we’re also unpacking that nested tuple to get “zelda” and “ganon” into wisdom and power.

Wrong answer explanations:
(A) On the right side, we have a tuple consisting of three items – “kirby”, “dedede”, and “metaKnight”. But on the left side, we’re only trying to create two separate variables. Because our number of variables differs from the length of the object we’re trying to unpack, this will be an error.

(B) Same as (A) , only that we have parentheses around the left side and not the right side.

10. Which of the following correctly represents the comparison rules for tuples, sets, and dictionaries in Python?

    A. Tuples are compared element-wise using < and >. Sets can be tested to see if one is a subset or superset of another. Dictionaries can only be compared for equality.

    B. Tuples are compared based on their length. Sets support element-wise comparisons using > and <. Dictionaries cannot be compared.

    C. Tuples can only be compared for equality. Sets are compared based on their length. Dictionaries can only be compared for equality.

    D. Tuples are compared element-wise using < and >. Sets can be tested to see if one is a subset or superset of another. Dictionaries cannot be compared.

Correct answer explanation:
(A) This is mostly knowing functionality for tuples, sets, and dictionaries. Sets don’t have order, but they can be compared based on supersets or subsets. Tuples have orders, so we can do element-wise comparisons. Dictionaries don’t have methods like superset or subset, and they also don’t support < or > comparisons (how would that work?), so we can only compare dictionaries for equality.

Wrong answer explanations:
(B) element-wise comparisons uses positioning. Think strings. Comparing two strings means you’re comparing the characters at each index. But sets don’t have indexes, so (B) is false. Also, dictionaries can be compared for equality.
(C) Tuples can be compared element-wise, like a string. Comparing the items at each index of the tuples. Ex: (5, 10) is greater than (5, 9)

(D) Dictionaries can be compared for equality.

11. What happens when you try to add two dictionaries using the + operator?

    A. The dictionaries are merged, and if there are overlapping keys, the values from the second dictionary overwrite those from the first.
    B. The dictionaries are merged, and if there are overlapping keys, their values are summed.
    C. It creates a list containing both dictionaries as elements.
    D. It raises an error.

Correct answer explanation:
(D) Yes! This is correct. We can't add two dictionaries using +. But A is VERY close. If we wanted to combine two dictionaries, I think we would need to use .update(). Which we didn't cover in this class. But

myDict1.update(myDict2) essentially just adds the key/value pairs from myDict2, to myDict1. If myDict2 happens to share a key with myDict1, then its value for that key will override the value in myDict1. Example:

```
x = {1: "one", 2: "two"}
y = {1: "five", 2: "two", 3: "three"}
x.update(y)
x will be {1: 'five', 2: 'two', 3: 'three'}
```

But this is pretty niche. Not sure when we would need to combine two dictionaries. . .

12. Which of the following statements about tuples, sets, and dictionaries in Python is false?

    A. Slicing a tuple will not raise an error.
    B. Dictionary keys can be associated with multiple values.
    C. The keys in a dictionary must be of immutable types, such as strings or tuples.
    D. For any set `s1`, `s1.issuperset(set())` will evaluate to `True`.
    E. The `"in"` operator applied to dictionaries searches keys and not values.

Correct answer explanation:

(B) False! It wouldn't make any sense for keys to have multiple values. If we do

```
myDict = {}
myDict[1] = 5
myDict[1] = 10
print(myDict[1])
```

This will simply print 10. There's no way for us to assign multiple values to a key, because when we try to do so, we override the last value.

You might question – what if we did this:

```
myDict[1] = [5, 10]
```

But this simply sets our value for 1, to be a list. A *singular* list. So, our key 1 still only has one value associated with it.

Wrong answer explanations:
(A) True! Because tuples are ordered, and have indexes, we can slice a tuple, even though they're immutable. If that confuses you, think about how strings are also immutable, but we can slice them (because they're ordered)

(C) This is true. We can't have lists as keys in our dictionary, for example. Doing something like

```
myDict = {}
myList = [1]
myDict[myList] = 0 # set myList as a key, with value 0
```

Our code will crash.

(D) This is true! s1.issuperset(set()) is simply asking, is s1 a superset of the empty set. Or, in other words, it is asking, if the empty set is a subset of s1. And this must be true. To be a subset of another set, each one of your items have to be present in the other set. So, the empty set is a subset of every set because all of its elements are in any other set -- because the empty set has no items! Teehee! =)

TRACING

13. (3 points)

```
def avatar(krew):
    krew += ("tenzinAir", "bolinEarth", "makoFire")
    return krew

otp = ("korra", "asami")
avatar(otp)
for i in range(len(otp)):
    if i % 2:
        print(otp[i][:5], end = "")
    else:
        print(otp[i][:4], end = "")
```

korrasami

An Avatar question! It's about time I referenced these characters.

OK, we have function avatar() which takes in a parameter (krew) and adds to it a tuple ("tenzinAir", "bolinEarth", "makoFire"). It then returns this new version of krew.

We start with a tuple called otp ("korra", "asami"). We call avatar(otp), but note we are *not* updating otp with this. We simply send otp as a parameter, but we do *not* reassign otp. That would require doing

otp = avatar(otp)

But we're only doing

avatar(otp)

So avatar will add to its parameter, but the change will not persist, because we're not reassigning otp when this is called.

So after we run that function, otp will not have changed.

We then loop over the length of otp – this tuple has length 2. So range(len(otp)) will get us back [0, 1]. The first time through the loop, i will be 0, so 0 % 2 == 0.

This means ‘if i % 2’ is equal to ‘if 0’, which is False. So we do the else. We print otp[i][:4] – in this case, otp[0][:4], or just “korr”. Our end is “” meaning we do not move onto the next line or append any space.

The second time through the loop, i will be 1, so 1 % 1 == 1. This means ‘if i % 2’ is equal to ‘if 1’, which is True. So we do the ‘if’. We print otp[i][:5] – in this case, otp[1][:5], or just “asami”.

So, our final result displayed is “korrasami”.

14. (3 points)

```
gaang = ("aang", "katara", "sokka")
gaang += ("toph", "zuko")
print(gaang, end = " ")
gaang -= ("zuko", "aang")
print(gaang, end = " ")
```

('aang', 'katara', 'sokka', 'toph', 'zuko') Error

Fun!! Another Avatar question! You’d be surprised, but I actually never watched the show. I included the references just for the kiddos who have seen it. It’s popular!

OK, we start with gaang, tuple (“aang”, “katara”, “sokka”). Then we overwrite gaang by adding to it (“toph”, “zuko”). So now gaang is (“aang”, “katara”, “sokka”, “toph”, “zuko”).

We print gaang, and that is what will display. After, we try subtracting from it (“zuko”, “aang”). But!! We can’t use - on tuples. Remember that tuples are immutable? If we want to remove items in a tuple, we need to create a whole new tuple.

“But we were able to add to gaang!”

Yes – we were! True. But -= is not supported for tuples. Subtraction isn’t defined for tuples (like how it isn’t defined for strings).

If this still confuses you, let’s replace tuples with strings (which are also immutable):

myString = “silly”
myString += “Goose”
myString -= “Goose”

This will, similarly, crash.

15. (3 points)

```
marioKart = {"donkeyKong": "goodie", "wario": "baddie", \
             "luigi": "goodie", "bowser": "baddie"}
marioParty = {char: align for align, char in marioKart.items()}
print(marioParty)
```

{'goodie': 'luigi', 'baddie': 'bowser'}

Heehee. Mario Kart is so toxic.

OK. We create a dictionary where characters are our keys and their alignment are our values. We name this marioKart. And we then create a new dictionary, called marioParty, by swapping the keys and values in our dictionary. We do

{char: align for align, char in marioKart.items() }
.items() on dictionary returns a list object. And each item of this list, is a tuple consisting of a key and its value from our dictionary. So, marioKart.items() returns something like

[(“donkeyKong”, “goodie”), (“wario”, “baddie”), (“luigi”, “goodie”), (“bowser”, “baddie”)]

So, {char: align for align, char in marioKart.items() } . Let’s rewrite this as

{ for align, char in marioKart.items()
        char: align
}

So in ‘for align, char in marioKart.items()’ we’re doing tuple unpacking, referring to the first item in the tuple as ‘align’ and the second item as ‘char’. So for the first item in our list, (“donkeyKong”, “goodie”), this will mean align is “donkeyKong” and char is “goodie”.

So, we place in our new dictionary the key-value pair char:align, which would be “goodie”:”donkeyKong”.

For (“wario”, “baddie”) in our list, we add to the new dictionary “baddie”: “wario”.

But for (“luigi”, “goodie”), we add to the dictionary “goodie”: “luigi”. So, we’re overwriting the last value for “goodie”, which was “donkeyKong”.

We do the same for (“bowser”, “baddie”), which changes the value (“wario”) for the key “baddie” to be “bowser”.

Thus, our resulting dictionary is {“goodie”: “luigi”, “baddie”: “bowser”}

16. (3 points)

```
teenTitans = {"robin", "starfire", "raven", "beastboy", "cyborg"}
girlsRule = {"starfire", "raven"}
boysDrool = {"robin", "beastboy", "cyborg"}

azarath = teenTitans.difference(boysDrool)
silkie = teenTitans.symmetric_difference(boysDrool)

if azarath == silkie:
    print(teenTitans.intersection(girlsRule))
else:
    print(boysDrool.union(girlsRule))
```

```
{'raven', 'starfire'}
```

This is a fun question!

OK, we have sets teenTitans (all 5 characters), girlsRule (which has “starfire” and “raven”) and boysDrool (which has “robin”, “beastboy”, and “cyborg”).

The difference method for sets, what that does: set1.difference(set2), you can think of ‘starting’ with the items of set1, and ‘removing’ from that group, the items that are also present in set2.

So, azarath = teenTitans.difference(boysDrool). We start with all 5 characters, and we remove those present in boysDrool, which are “robin”, “beastboy”, and “cyborg”. So our remaining items are “starfire” and “raven”. Thus, azarath will be {“starfire”, “raven”}.

The symmetric difference, however. . . set1.symmetric_difference(set2). This will get us the items present in strictly *one* of the sets, not both. So, we can think of the symmetric_difference as the union of the two sets, minus their intersection (the items that are in both). We essentially pool together all possible elements, and remove the ones that happen to be in both sets.

We're trying to do teenTitans.symmetric_difference(boysDrool). In this case, teenTitans already contains all items from boysDrool. So, the union between these two sets is simply teenTitans. We remove the intersection (which is "robin", "beastboy", "cyborg"). Again, the result is {"starfire", "raven"}, and we store that in silkie.

Now we test whether azarath is equal to silkie. Because they contain the same items, they will be equal to each other. So, the 'if' triggers, and we print the intersection between teenTitans and girlsRule, which is simply {"starfire", "raven"}.

17. (3 points)

```
monsterHigh = {"frankie", "draculaura", "clawdeen", "cleo"}
ghoulSpirit = {"frankie", "lagoona", "ghoulia", "abbey", "spectra", "clawdeen"}
ghoulPower = monsterHigh.symmetric_difference(ghoulSpirit)
sixInchHeels = monsterHigh + ghoulSpirit.difference(monsterHigh)
print(ghoulPower == sixInchHeels)
```

Error

Love Monster High!

We start with two sets – monsterHigh and ghoulSpirit.

We take the symmetric difference between monsterHigh and ghoulSpirit. Recall, symmetric_difference gets us the items present in only *one* of the sets. I really do like to think of symmetric_difference as the union of the two sets minus their intersection.

In this case, the union is {"frankie", "draculaura", "clawdeen", "cleo", "lagoona", "ghoulia", "abbey", "spectra"}. The intersection is {"frankie", "clawdeen"}. So, this means the symmetric difference will be {"draculaura", "cleo", "lagoona", "ghoulia", "abbey", "spectra"} – this is stored in ghoulPower.

So good so far.

We then try to find the sum between monsterHigh and ghoulSpirit.difference(monsterHigh). '

ghoulSpirit.difference(monsterHigh): this will be the items in ghoulSpirit that are *not* present in monsterHigh – the result of this will be {"draculaura", "cleo"}.

We try adding monsterHigh to this, – but!!!! We're trying to use +. We can't use + with sets. Oops. Really. If you want to pool the items of two sets together, you need to do .union(). You can't do set1 + set2. Python gets mad and raises an error.

So, yes, I had you read the above text just for there to be a flippant error in the code. Sorry!!! But I really want you kiddos to know what symmetric difference does (and knowing you can't use + on two sets helps, too), so I included this question. Gotcha!!!! =)

18. (3 points)

```
blastFromPast = ("nancyDrew", ["peppaPig", "goosebumps"], "junieBjones")
blastFromPast[1][0] = "magicSchoolBus"
print(blastFromPast)
```

('nancyDrew', ['magicSchoolBus', 'goosebumps'], 'junieBjones')

Interesting!! I know this question probably got a few of you to shake in your boots.

OK, we have blastFromPast, which is a tuple:

("nancyDrew", ["peppaPig", "goosebumps"], "junieBjones")

And we index into this tuple, to get the item at index 1 (["peppaPig", "goosebumps"]), and we index into this item to change "peppaPig" to "magicSchoolBus".

This might seem strange – you kiddos are probably asking, I thought we couldn't change the items of a tuple!

Well, you are correct! Tuples in Python are immutable, which means once they are created, their elements cannot be changed. But! In this case, the tuple contains a list (["peppaPig", "goosebumps"]), and lists are mutable. So, even though the tuple itself cannot be modified, we can change the contents of the list it contains.

So, "peppaPig" is successfully changed to "magicSchoolBus". When we print blastFromPast after, it will be

("nancyDrew", ["magicSchoolBus", "goosebumps"], "junieBjones")

19. (3 points)

```
simba = (3, 7)
timon, pumbaa = simba
timon += 1; pumbaa += 1
simba += (2, 5)
print(timon, pumbaa, simba)
```

4 8 (3, 7, 2, 5)

OK. We start off with simba, which is a tuple (3, 7). We unpack the items of this tuple, so we store 3 in timon and 7 in pumbaa. We then increment both timon and pumbaa by 1 – so timon becomes 4 and pumbaa becomes 8.

We then do a += on simba, (2, 5). But note that simba += (2, 5) will *not* increment index 0 of simba by 2 and index 1 by 5. Nope! It will simply create a new tuple, adding on 2 and 5 to the end. So at this point, simba is (3, 7, 2, 5).

We then print timon, pumbaa, and simba. This will be

4 8 (3, 7, 2, 5)

20. (3 points)

```
spotifyPlaylist = {"partyInUSA": "miley", "hitMeBaby": "britney",
                   "want4Christmas": "mariah", "theClimb": "miley",
                   "obsessed": "mariah", "circus": "britney",
                   "singleLadies": "beyonce"}
print({spotifyPlaylist.pop(song) for song in list(spotifyPlaylist.keys())})
```

{'beyonce', 'britney', 'miley', 'mariah'}

LOL

This is so funny.

OK. We create a dictionary where our keys are the best songs ever, such as partyInUSA, and each key's value is the artist.

Then, we create a set, using set comprehension.

{spotifyPlaylist.pop(song) for song in list(spotifyPlaylist.keys())}

This is a mouthful. But reordering comprehensions always helps us understand it a bit more. We can reorder it like so:

```
{ for song in list(spotifyPlaylist.keys()):
        spotifyPlaylist.pop(song)
 }
```

So, we create a list of our keys, by doing list(spotifyPlaylist.keys()). This would be [“partyInUSA”, “hitMeBaby”, “want4Christmas”, “theClimb”, “obsessed”, “circus”, “singleLadies”].

And for each of these items, which we refer to as ‘song’, we do spotifyPlaylist.pop(song). When we call .pop() on a dictionary, we specify a key, and .pop removes that key from the dictionary and returns us the value associated with it. So, when we do

spotifyPlaylist.pop(“partyInUSA”)

We get back “miley”. So, “miley” will be an item in our set.

So, we go through our list, calling .pop() on each one, and getting back the artist name. And remember that sets only allow for unique items, so although we pop both partyInUSA and theClimb, and we get back “miley” for both, we will still only have one “miley” in our set.

So, our final set will be some ordering of our values in our dictionary – that is, {“miley”, “britney”, “mariah”, “beyonce”}.

21. (3 points)

```
overwatch = {"soldier76": {"hp":200, "ult":300, "dmg":100},
             "mercy":     {"hp":200, "ult":250, "dmg":60},
             "dva":       {"hp":600, "ult":400, "dmg":80} }
patch =     {"soldier76": {"hp":50, "ult":-20, "dmg":10},
             "mercy":     {"hp":25, "ult":10, "dmg":10},
             "dva":       {"hp":0, "ult":20, "dmg":5} }

for hero, changes in patch.items():
    for stat, adjust in changes.items():
        overwatch[hero][stat] = overwatch[hero][stat] + adjust
# i didnt realize how much this is to write...
# i'll probably tweak this question, just treat
# it as a fun exercise for now =)
print(overwatch)
```

I don't know what I was thinking!!! I'm sorry!

{'soldier76': {'hp': 250, 'ult': 280, 'dmg': 110}, 'mercy': {'hp': 225, 'ult': 260, 'dmg': 70}, 'dva': {'hp': 600, 'ult': 420, 'dmg': 85}}

Unless you want to be a tech wiz, you can easily skip this question. It's MESSY. Consider this question a RELIC. But I do think it's neat to have /some/ weird questions, for you kiddos who truly want a challenge and something unexpected.

Alright, so we have nested dictionaries, overwatch and patch. We're doing

for hero, changes in patch.items()
    for stat, adjust in changes.items()
        overwatch[hero][stat] = overwatch[hero][stat] + adjust

patch.items() returns a list object containing key-value tuples. So, patch.items would be

[('soldier76', {'hp': 50, 'ult': -20, 'dmg': 10}),
 ('mercy', {'hp': 25, 'ult': 10, 'dmg': 10}),
 ('dva', {'hp': 0, 'ult': 20, 'dmg': 5})]

Our keys are the character name, and their values are another dictionary.

'for hero, changes in patch.items()' loops over the tuples in this list, and unpacks them. So, hero will refer to the key (name), and changes will refer to the key (dictionary).

Our inner loop, ‘for stat, adjust in changes.items()’ similarly gets a list of key-value pairs in changes, and unpacks them.

As an example: the first iteration of the outer loop, our item is ('soldier76', {'hp': 50, 'ult': -20, 'dmg': 10}). So, hero is ‘soldier: 76’, changes is {'hp': 50, 'ult': -20, 'dmg': 10}.

Then, for the inner loop, changes.items() will be [(‘hp’:50), (‘ult’:-20), (‘dmg’:10)].

So the inner loop iterates over this list, unpacking the tuples, with ‘stat’ referring to the first item of the current tuple, and ‘adjust’ referring to the second item. We make the changes

overwatch[‘soldier: 76’][‘hp’] = overwatch[‘soldier: 76’][‘hp’] + 50
overwatch[‘soldier: 76’][‘ult’] = overwatch[‘soldier: 76’][‘ult’] - 20
overwatch[‘soldier: 76’][‘dmg’] = overwatch[‘soldier: 76’][‘ult’] + 10

So, after this, in our overwatch dictionary, the value for ‘soldier: 76’ should be {'hp': 250, 'ult': 280, 'dmg': 110}.

And we’re doing this sequence of steps for each character (so, also for mercy and dva). Essentially, we’re applying the adjustments specified in the patch dictionary to the corresponding stats of each hero in the overwatch dictionary. We're updating the stats by adding or subtracting the adjustment values provided in the patch dictionary to the current values of the corresponding stats for each hero in the overwatch dictionary.

I don’t think this question is at all indicative of your performance on the exam. But I do think wrapping your head around the gist of this question, helps flex a muscle of sorts. So don’t take this question too seriously. Just try to understand what’s going on =)

22. (3 points)

```
mister = [3, 5, 2, 8, 1]
taterTot = set()
for pumpkin in mister:
    if pumpkin % 3 == 0:
        taterTot.add(pumpkin)
    elif pumpkin % 2 == 0:
        taterTot.add(pumpkin // 2)
    else:
        taterTot = taterTot.difference({pumpkin // 2})
print(taterTot)
```

{1, 3, 4}

Mister, Tater Tot, and Pumpkin were all my cats =)

OK. We have our list mister containing [3, 5, 2, 8, 1]. taterTot is an empty set. We then start a loop

for pumpkin in mister:

So, this iterates over our list mister, referring to the current item as pumpkin.

We then have a few conditionals, testing pumpkin % 3 and pumpkin % 2.

If pumpkin % 3 is 0, we add pumpkin to the set.
Else if pumpkin % 2 is 0, we add pumpkin // 2 to the set.
Else, we subtract pumpkin // 2 from the set.

The first iteration, pumpkin will be 3. We test the ‘if’. 3 % 3 == 0, so we add 3 to our set. Our set is currently {3}.

Second iteration, pumpkin will be 4. 4 % 3 is not 0, so we test the ‘elif’. 4 % 2 == 0, so we add pumpkin // 2, which is 2, to our set. Our set is currently {3, 2}.

Third, pumpkin will be 5. 5 % 3 is not 0. So our ‘if’ fails. We test 5 % 2, which is also not 0, so our ‘elif’ fails. So, the ‘else’ triggers. We remove 5 // 2, which is 2, from our set. Our set is currently {3}.

Fourth, pumpkin will be 8. 8 % 3 is not 0, so the ‘if’ fails. We then test 8 % 2 for the ‘elif’. This is 0, so we add 8 // 2, or 4, to our set. Our set is currently {3, 4}.

On the fifth and last iteration, pumpkin is 1. 1 % 3 is not 0, so the ‘if’ fails. 1 % 2 is also not 0, so the ‘elif’ fails too. So, we remove 1 // 2, or 0, from our set. 0 wasn’t in our set anyways, so our set stays the same.

Thus, when we print taterTot at the end, it will be {3, 4}.