Last Name: ________________ First: ________________ Netid: ________________

# CS 1110 Prelim 1 October 11th, 2018

This 90-minute exam has 6 questions worth a total of 100 points. Scan the whole test before starting. Budget your time wisely. Use the back of the pages if you need more space. You may tear the pages apart; we have a stapler at the front of the room.

**It is a violation of the Academic Integrity Code to look at any exam other than your own, to look at any other reference material, or to otherwise give or receive unauthorized help.**

You will be expected to write Python code on this exam. We recommend that you draw vertical lines to make your indentation clear, as follows:

```
def foo():
|   if something:
|   |   do something
|   |   do more things
|   do something last
```

You should not use loops or recursion on this exam. Beyond that, you may use any Python feature that you have learned in class (if-statements, try-except, lists), **unless directed otherwise**.

| Question | Points | Score |
|---|---|---|
| 1 | 2 | |
| 2 | 16 | |
| 3 | 22 | |
| 4 | 20 | |
| 5 | 20 | |
| 6 | 20 | |
| Total: | 100 | |

**The Important First Question:**

1. [2 points] Write your last name, first name, and netid, at the top of *each* page.

Last Name: ____________ First: ____________ Netid: ____________

## Reference Sheet

Throughout this exam you will be asked questions about strings and lists. You are expected to understand how slicing works. In addition, the following functions and methods may be useful.

### String Functions and Methods

| Expression or Method | Description |
|---|---|
| `len(s)` | **Returns**: number of characters in `s`; it can be 0. |
| `a in s` | **Returns**: True if the substring `a` is in `s`; False otherwise. |
| `s.count(s1)` | **Returns**: the number of times `s1` occurs in `s` |
| `s.find(s1)` | **Returns**: index of the first character of the FIRST occurrence of `s1` in `s` (-1 if `s1` does not occur in s). |
| `s.find(s1,n)` | **Returns**: index of the first character of the first occurrence of `s1` in `s` STARTING at position n. (-1 if `s1` does not occur in s from this position). |
| `s.isalpha()` | **Returns**: True if `s` is *not empty* and its elements are all letters; it returns False otherwise. |
| `s.isdigit()` | **Returns**: True if `s` is *not empty* and its elements are all numbers; it returns False otherwise. |
| `s.isalnum()` | **Returns**: True if `s` is *not empty* and its elements are all letters or numbers; it returns False otherwise. |
| `s.islower()` | **Returns**: True if `s` is *has at least one letter* and all letters are lower case; it returns False otherwise (e.g. `'a123'` is True but `'123'` is False). |
| `s.isupper()` | **Returns**: True if `s` is *has at least one letter* and all letters are uppper case; it returns False otherwise (e.g. `'A123'` is True but `'123'` is False). |

### List Functions and Methods

| Expression or Method | Description |
|---|---|
| `len(x)` | **Returns**: number of elements in list `x`; it can be 0. |
| `y in x` | **Returns**: True if `y` is in list `x`; False otherwise. |
| `x.count(y)` | **Returns**: the number of times `y` occurs in `x` |
| `x.index(y)` | **Returns**: index of the FIRST occurrence of `y` in `x` (an error occurs if `y` does not occur in `x`). |
| `x.index(y,n)` | **Returns**: index of the first occurrence of `y` in `x` STARTING at position n (an error occurs if `y` does not occur in `x`). |
| `x.append(y)` | Adds `y` to the end of list `x`. |
| `x.insert(i,y)` | Inserts `y` at position `i` in list `x`, shifting later elements to the right. |
| `x.remove(y)` | Removes the first item from the list whose value is `y`. (an error occurs if `y` does not occur in `x`). |

The last three list methods are all procedures. They return the value `None`.

2. [16 points total] **Short Answer Questions**.

(a) [5 points] What is the definition of a type in Python? List at least four examples of types built into Python.

A type is a collection of values together with the operations on them. The four basic types are `int`, `float`, `bool`, and `str`. However, we would also allow `list` or `tuple`.

(b) [4 points] What is the definition of a *type cast*? What is the difference between a *widening* cast and a *narrowing* cast? Give an example of each.

A *type cast* is the conversion of a value from one type to another. A widening cast converts the type from one with less information to more information (e.g. int to float). An example is `float(2)` or `3/2.0`. A narrowing cast is the reverse – from more information to less. An example is `int(2.3)`.

(c) [4 points] Explain the purpose of *preconditions* in a function specification. Why are they necessary in Python?

Preconditions are a promise that our function will work properly if the arguments all satisfy the restrictions listed. They are necessary because we cannot possibly guarantee that our function will work on any arbitrary argument. For example, we cannot guarantee that a math function like cos will work on a string, or on a list. So we only guarantee the function on those arguments that satisfy the precondition.

(d) [3 points] Consider the code below. Is the code correct or will it produce an error? If it is correct, what value is put in the variable `y`? If it is not correct, explain the error.

```
import math

def foo(math):
    return math.cos(0)

y = foo(3)
```

It produces an error. The call to `math.cos` will fail because `foo` has a parameter named `math` and Python will use that instead of the module `math`. Hence `math` is the number 3, and it does not contain a function called `cos`.

3. [22 points] **Call Frames.**

Consider the following function definitions.

```
def back_to_front(p):
    """Returns: copy of p with ends swapped
    Precondition: p a list, len(p) >= 2"""
    y = p[-1]
    x = add_front(p[1:-1],y)
    x.append(p[0])
    return x
```

```
def add_front(p,y):
    """Appends y to front of p
    Precondition: p a list, y anything."""
    p.insert(0,y)
    return p
```

This function returns a copy of a list with the first and last elements swapped.

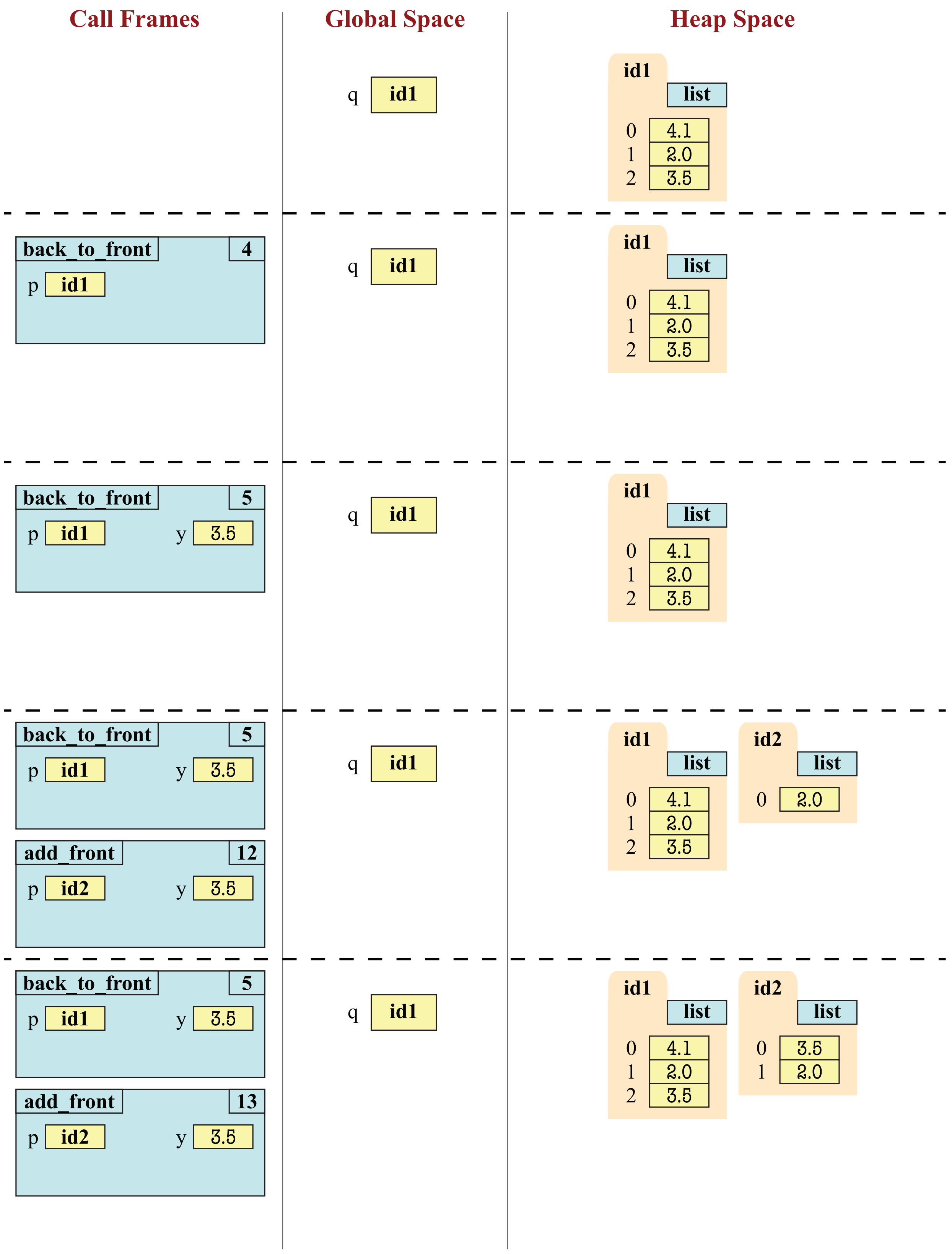
Assume that `q = [4.1,2.0, 3.5]` is a global variable referencing a list in heap space, as shown on the next page. On the next two pages, diagram the evolution of the call

```
p = back_to_front(q)
```

Diagram the state of the *entire call stack* for the function `back_to_front` when it starts, for each line executed, and when the frame is erased. If any other functions are called, you should do this for them as well (at the appropriate time). This will require a total of **nine** diagrams, not including the (pre-call) diagram shown.

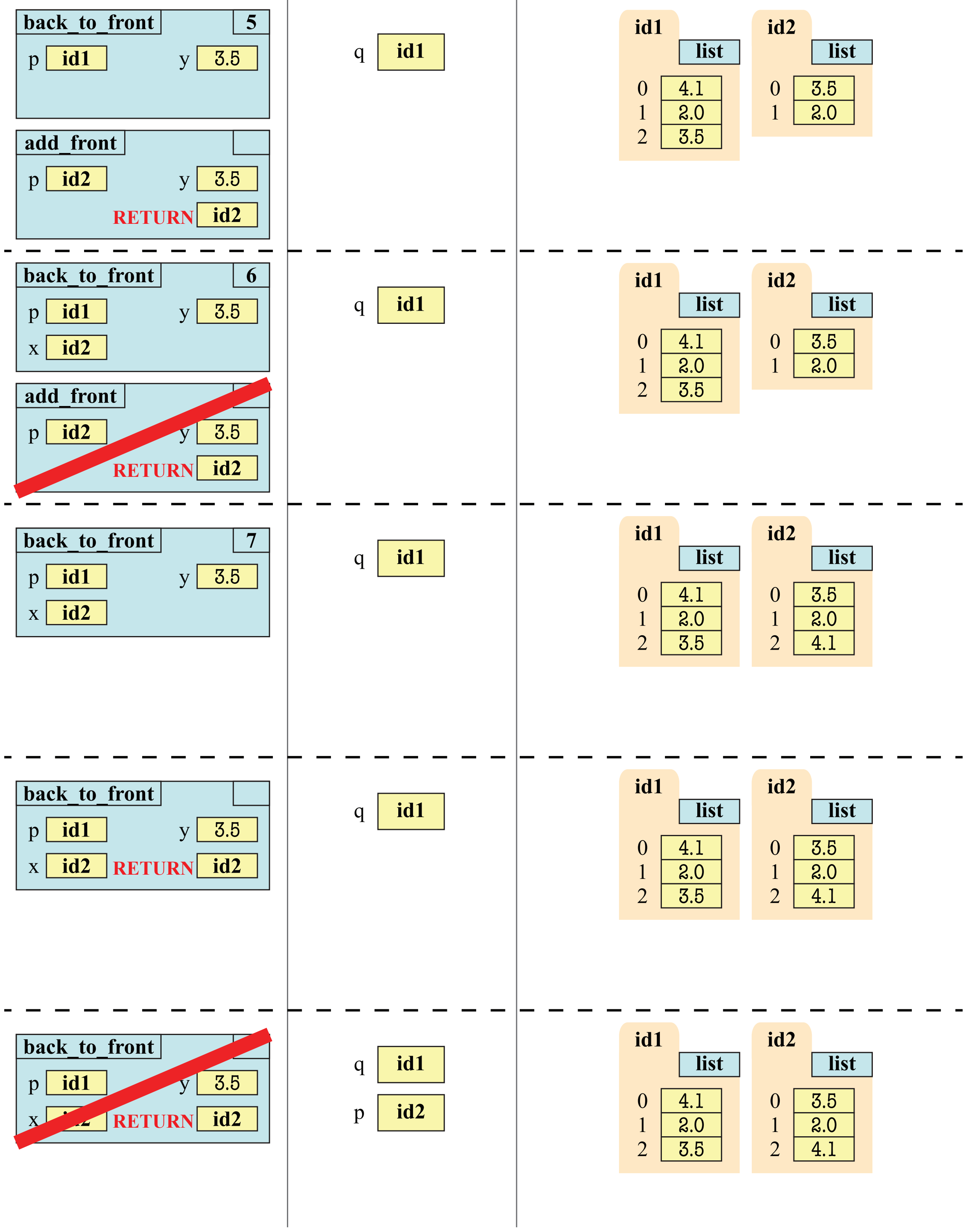
You should draw also the state of global space and heap space at each step. You can ignore the folders for the function definitions. Only draw folders for lists or objects. You are also allowed to write "unchanged" if no changes were made to either global or heap space.

Last Name: ______________________ First: ______________________ Netid: ______________________



Last Name: ______________ First: ______________ Netid: ______________

**Call Frames** **Global Space** **Heap Space**



Last Name: ________________ First: ________________ Netid: ________________

4. [20 points] **String Slicing.**

   If you have ever downloaded DLC for a game, or redeemed a coupon online, you know that they are often defined as groups letters and numbers separated by dashes. The simplest variation has just one dash and groups its letters in numbers in blocks of four, like this: `K97J-FTRE`.

   Implement the function below, which takes an arbitrary string and determines whether it looks like a coupon code. You will need to use several of the functions and methods on the reference sheet. **Pay close attention to the specifications of these methods and functions. You may not use loops.**

```
def is_code(value):
    """Returns: True if value is a coupon code, otherwise False

    A code has the form XXXX-XXXX where XXXX is four characters
    that are each either an upper case letter or a number.

    Example: is_code('K97J-FTRE') is True
             is_code('K97J-9876') is True
             is_code('K97J') is False
             is_code('K97J-FTRE-9876') is False
             is_code('k97J-fTRe') is False
             is_code('K97J8-FTREK') is False

    Precondition: value is a string"""
    # Verify just one dash
    if value.count('-') != 1:
        return False

    # Split into two blocks
    pos = value.find('-')
    block1 = value[:pos]
    block2 = value[pos+1:]

    # Verify the block size
    if len(block1) != 4 or len(block2) != 4:
        return False

    # All numbers or alphanumeric upper case
    good1 = block1.isdigit() or (block1.isupper() and block1.isalnum())
    good2 = block2.isdigit() or (block2.isupper() and block2.isalnum())
    return good1 and good2
```

5. [20 points total] **Testing and Debugging**.

   (a) [10 points] Consider the functions below. The function `valid_date` takes a string of the form 'month/day/year' and determines whether it is a valid date. It is leap-year aware so that `valid_date('2/29/2004')` is True, while `valid_date('2/29/2003')` is False. Note that a year is a leap-year if it is divisible by 4, but centuries are only leap years if they are divisible by 400 (so 2000 was a leap year, but 2100 is not).

   To simplify the code, the precondition of the function `valid_date` specifies that the string `date` must have two slashes. The month part before the first slash is 1 or 2 numbers, as is the day part between the two slashes. The year part after the last slash is 4 numbers. So `'2/29/2004'` satisfies the precondition, but `'a/2/20045'` violates it.

   There are ***at least* three bugs** in the code below. These bugs are across all functions and are not limited to a single function. To help find the bugs, we have added several print statements throughout the code. The result of running the code with these print statements shown on the next page. Using this information as a guide, identify and fix the three bugs on the next page. You should explain your fixes.

   **Hint**: Pay close attention to the specifications. You may not be able to express your fixes as a single line of code. If you have to make changes to multiple lines to solve a single problem, just describe your changes instead of writing the complete code.

```python
def leap_year(year):
    """Returns: True a leap year; else False

    Precondition: year is a positive int"""
    if year % 4 != 0:
        print('Not leap year')      # TRACE
        return False
    if year % 100 == 0 and year % 400 != 0:
        print('Not leap century')   # TRACE
        return False
    print('Leap year')              # TRACE
    return True


def num_days(month,year):
    """Returns: number of days in month

    Example: days_in_month(2,2003) is 28, but
    days_in_month(2,2004) is 29.

    Precondition: month is an int 1..12
    year is a positive int"""
    # List of days for each month
    days=[31,28,31,30,31,30,31,31,30,31,30,31]
    if leap_year(year):
        days[1] = 29    # Change Feb
    result = days[month]
    print('Month '+str(month))         # WATCH
    print(' has '+str(result)+' days') # WATCH
    return result
```

```python
def valid_date(date):
    """Returns: True if date is an actual date

    Example: valid_date('2/29/2004') is True
    but valid_date('2/29/2003') is False

    Precondition: date is a string month/day/year
    where month and day are 1 or 2 numbers each
    and year is four numbers"""
    # Split up string
    pos1 = date.find('/')
    print('First / at '+str(pos1))      # WATCH
    pos2 = date.find('/',pos1+3)
    print('Second / at '+str(pos2))     # WATCH

    # Turn month, day, and year into ints
    month = int(date[:pos1])
    print('Month is '+str(month))       # WATCH
    day   = int(date[pos1+1:pos2])
    print('Day is '+str(day))           # WATCH
    year = int(date[pos2+1:])
    print('Year is '+str(year))         # WATCH

    if day < 1 or day > num_days(month,year):
        print('Day out of range')       # TRACE
        return False
    if month < 1 or month > 12:
        print('Month out of range')     # TRACE
        return False
    return True
```

**Tests**:

```
>>> valid_date('2/29/2003')
First / at 1
Second / at 4
Month is 2
Day is 29
Year is 2003
Not leap year
Month 2
 has 31 days
True
```

**First Bug:**

The bug for the first test is in `num_days`. List indices start at 0, not 1. Therefore, we need to change the list access at Line 27 to:

```
    result = days[month-1]
```

```
>>> valid_date('2/2/2000')
First / at 1
Second / at -1
Month is 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "dates.py", line 51, in valid_date
    day   = int(date[pos1+1:pos2])
ValueError: invalid literal for int() with
   base 10: '2/200'
```

**Second Bug:**

The bug for the second test is in `valid_date`. While it crashes at line 51, the problem is that the value for `pos2` is wrong (No \ was found). We need to change the method call at Line 45 to:

```
    pos2 = date.find('/',pos1+1)
```

```
>>> valid_date('13/10/2017')
First / at 2
Second / at 5
Month is 13
Day is 10
Year is 2017
Not leap year
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "dates.py", line 56, in valid_date
    if day < 1 or day > num_days(month,year):
  File "dates.py", line 27, in num_days
    result = days[month]
IndexError: list index out of range
```

**Third Bug:**

The bug for the third test is again in `valid_date`. While it crashes in `num_days`, a month value of 13 violates the precondition in this function. To solve this problem, we need to check the month *before* we check the day. This can be fixed by swapping the order of the two if-statements.

There is a fourth bug we missed. The precondition of valid_date allows a year of `'0000'`. However, num_days requires that the year be positive, and you assert this in Part C. You cannot change specifications to fix bugs, so the fix needs an addition if-statement in valid_date.

(b) [7 points] On the previous page you saw three different tests for `valid_date`. Below, write **seven more test cases** for this function. By a test case, we just mean an input and an expected output; you do not need to write an `assert_equals` statement. For each test case *explain why it is significantly different from the others*. The test cases need to be different from each other and **different from the three test cases on the previous page**.

| Input | Output | Reason |
|---|---|---|
| `'2/29/2004'` | `True` | February 29th in leap year |
| `'1/12/2003'` | `True` | One-digit month, two-digit day |
| `'12/1/2003'` | `True` | Two-digit month, one-digit day |
| `'12/12/2003'` | `True` | Two-digit month and day |
| `'0/12/2003'` | `False` | Zeroed month |
| `'12/0/2003'` | `False` | Zeroed day |
| `'6/31/2003'` | `False` | Day 31 in a 30 day month (e.g. June) |

There are many different possible answers to this question. Above are some of test cases we were thinking of. If you had (at least) seven test cases that were close to the ones below, you got full credit. Otherwise, we checked if your test cases were *different enough*, and awarded you 1 point for each test. Keep in mind that the examples on the previous page are: February 29th outside leap year, one-digit month and day, and month greater than 12.

There are a lot of leap year variations tha you can test as well. But leap year tests are not interesting or different if the date is not February 29th.

(c) [3 points] Below is the header and specification for the function `num_days`. Using assert statements, enforce the precondition of this function.

```
def num_days(month,year):
    """Returns: number of days in month

    Precondition: month is an int 1..12
    year is a positive int"""

    assert type(month) == int
    assert 0 < month and month < 13
    assert type(year) == int
    assert year > 0
```

Last Name: ____________________ First: ____________________ Netid: ____________________

6. [20 points total] **Objects and Functions**.

You saw the `Time` class in lab. However, to properly keep track of time, we also need the timezone. You are probably familiar with the main US timezones: Eastern, Central, Mountain, and Pacific. However, those names are not very helpful when we want to write code. So instead, timezones are expressed as an offset of UTC (Universal Coordinated Time). Eastern Standard Time is UTC-4, or four hours behind UTC. New Zealand Standard Time is UTC+12 or 12 hours ahead of UTC.

To change from one time zone to another, we usually do it in a two step process. We subtract the offset of the old timezone to shift to UTC. Then we add the offset of the new timezone to shift it again. For example, suppose we have 09:45 in Eastern (UTC-4) and we want to know the time in New Zealand. We add 4 hours to move to UTC, giving us 13:45. As New Zealand is UTC+12, we add 12 hours again to get 01:45 the next day.

Not all timezones are offset by a full hour. Indian Standard Time is UTC+5:30. Therefore, if want to add a timezone offset to the `Time` class, we should express it in minutes. The `Time` class now has the following attributes.

| **Attribute** | **Meaning** | **Invariant** |
|---|---|---|
| `hour` | the hour of the day | `int` value between 0 and 23 (inclusive) |
| `minute` | the minute of the hour | `int` value between 0 and 59 (inclusive) |
| `zone` | the offset from UTC in minutes | `int` value (no limitations) |

(a) [8 points] Implement the function below according to the specification
**Hint**: This problem is a lot easier if you convert the time to minutes as you did in the lab.

```
def is_before(time1,time2):
    """Returns: True if time1 happens before time2, False otherwise.
    It returns False if time1 and time2 are the same time.

    Example: If time1 is 10:30 in zone -240 (Eastern) and time2 is 15:20
    in zone +60 (Central Europe), then is_before(time1,times) is False.

    Preconditions: time1 and time2 are Time objects"""

    # Compute the minutes in UTC
    min1 = time1.hour*60+time1.minute-time1.zone
    min2 = time2.hour*60+time2.minute-time2.zone

    # Compare minutes
    return min1 < min2
```

(b) [12 points] Implement the function below according to the specification
**Hint**: The tricky part is satisfying the invariants. You may find `//` and `%` to be useful.

```
def shift_tz(time,tz):
    """MODIFIES time to use the timezone tz instead.

    In addition to setting the timezone attribute in time, this function
    modifies the time and minutes to be the correct time in the new timezone.

    Example: Suppose time is 16:40 in zone -240 (Eastern) and tz is +210
    (Iran Standard Time).  Then shift_tz(time,tz) modies time to be 00:10
    with timezone +210.

    Precondition: time is a Time object, tz is an int"""

    # Find the timezone difference
    diff = tz-time.zone

    # Add the difference to the time
    minutes = (time.minute+diff) % 60
    carry   = (time.minute+diff) // 60
    hours   = (time.hour+carry) % 24

    # Reset the object
    time.hour = hours
    time.minute = minutes
    time.zone = tz
```