CMPT 120 Standard Final Exam

Sample 2
Multiple Choice Questions

Duration	1 hour
Aids allowed	Pencil (or pen) and eraser. No notes, no papers, no books, no computers, no calculators, no cheat sheets, etc.
Scoring	For each question fill in **the one best answer** on the answer sheet. Each correct answer scores 1 point. Incorrect answers, multiple answers, illegible answers, or unanswered questions score 0 points.
During the exam	Raise your hand if you would like to speak with a proctor. Questions about exam/course content will **not** be answered during this exam.

1) What does this print?

```
print('3' * '4' + '2')
```

A. 122
B. 342
C. 3332
D. 44442
E. nothing: the statement has an error

Explanation: The expression `'3' * '4'` is an error because you cannot multiply two strings in Python. It would be okay to write `3 * '4'` (which is `'444'`) or `'3' * 4` (which is `'3333'`).

2) Consider this program:

```
a = -3
b = a - 3
a = b + a
b = a + b
print(a)  # line 1
print(b)  # line 2
```

i) line 1 prints -9
ii) line 2 prints -9

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is true and ii) is false
D. i) is false and ii) is true

Explanation: We can trace this program line by line, keeping track of each variable's value:

	a	b
`a = -3`	**-3**	**undefined**
`b = a - 3`	**-3**	-3 - 3 = **-6**
`a = b + a`	-3 + -6 = **-9**	**-6**
`b = a + b`	**-9**	-9 + -6 = **-15**

This shows that `a` is -9 and `b` is -15, and so i) is true and ii) is false.

3) Consider this code fragment:

```
a = 4
b = 1
???
print(a) # 1
print(b) # 4
```

Suppose `???` is replaced by one of the fragments below. Which one makes the code print 1 for `a` and 4 for `b`?

A.
```
a = a - b
b = b + a
a = a - b
```

B.
```
t = a
a = b
b = t
```

C.
```
t = b
a = b
b = t
```

D. all of A, B, and C

Explanation: The code in `???` swaps the values of the variables. B is the only correct way to do this among the examples given, which you can check by tracing the code.

B works by using variable, `t` (for "temporary") to save the value of `a`. Then the value of `b` is copied into `a`, which over-writes `a`'s value. Then `t` is copied into `b`, giving it the value `a` originally had.

4) Consider this statement:

```
print(2 + (4 ??? 3))
```

How many of these 4 operators can replace `???` so that the statement prints 3?

```
+    -    *    %
```

A. 0
B. 1
C. 2
D. 3
E. 4

Explanation: We can see that 2 (C) is the correct answer by evaluating each of the expressions:

```
print(2 + (4 + 3))  # 9
print(2 + (4 - 3))  # 3
print(2 + (4 * 3))  # 14
print(2 + (4 % 3))  # 3
```

Recall that `%` is the **mod** (or **remainder**) operator, i.e. `n mod a` returns the remainder when `n` is divided by `a`. For example, `4 % 3` is 1, because 3 goes into 4 with a remainder of 1.

5) Consider these statements:

i) Python strings **can** be modified
ii) Python lists **cannot** be modified

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is true and ii) is false
D. i) is false and ii) is true

Explanation: i) is false since Python strings are **immutable**, i.e. not changeable. Once you create a string, you cannot change it's size or modify any of its characters. ii) is false because Python lists are **mutable**, i.e. after you create a list you can change its value or change its size.

6) What does this print?

```
lst = [2, 0, -1, 1]
print(lst[1 + lst[1]])
```

A. 2
B. 0
C. -1
D. 1
E. nothing: there is an indexing error

Explanation: We can evaluate the print statement step-by-step like this:

```
print(lst[1 + lst[1]])  # original statement
print(lst[1 + 0])       # lst[1] is 0
print(lst[1])           # 1
print(0)                # lst[1] is 0
```

7) What does this print?

```
scores = [2, 1, 3]
T = scores
scores[2] = 0
print(T)
```

A. `[0, 1, 3]`
B. `[2, 0, 3]`
C. `[2, 1, 0]`
D. `[2, 1, 3]`

Explanation: Since `scores` is a list, after running the assignment statement `T = scores` the list `T` *refers* to the same list as `scores`. It does *not* make a copy. If you did want to store a copy you could use the statement `T = scores[:]`.

8) Consider this code:

```
s = 'thimble'
```

Which statement prints `himb` ?

A. `print(s[1:4])`
B. `print(s[1:5])`
C. `print(s[2:4])`
D. `print(s[2:5])`

Explanation: Expressions of the form `s[a:b]` are slices. The slice `s[a:b]` returns a copy of all the values from `s[a]` to `s[b-1]`. Note that `s[b]` is *not* included in the slice.

Since `s[0]` is `'t'` (the first character of `s`), we get:

```
print(s[1:4])  # 'him'
print(s[1:5])  # 'himb'
print(s[2:4])  # 'im'
print(s[2:5])  # 'imb'
```

9) What does this print?

```
d = {}
d[5] = 3
d[5] = 2
print(d[5])
```

A. 2
B. 3
C. it prints some value other than 2 or 3
D. this code crashes with an error when run

Explanation: We can trace the code to see what it prints:

	Value of `d`
`d = {}`	{} (the empty dictionary)
`d[5] = 3`	**{5 : 3}**
`d[5] = 2`	**{5 : 2}**
`print(d[5])`	**2**

In a Python dictionary, if you write `d[5] = 2` then the `5:2` is guaranteed to be in `d`. If 5 was already a key in `d` then it's corresponding value will get over-written by 2.

10) How many of these three programs print 6?

```
# program P
d = {'x':1, 'y':2, 'z':3}
total = 0
for x in range(len(d)):
    total += d[x]
print(total)
```

```
# program Q
d = {'a':1, 'b':2, 'c':3}
total = 0
for x in d:
    total += d[x]
print(total)
```

```
# program R
d = {'t':1, 's':2, 'u':3}
total = 0
for x in d:
    total += x
print(total)
```

A. 0
B. 1
C. 2
D. 3

Explanation: Program Q is the only program that prints 6, as you can verify by tracing it. Program P crashes with an error because the first value for `x` in the for-loop is 0, and `d[0]` does not exist, which causes an error.

Program R crashes with an error because `total` contains the `int` 0, and the first value of `x` is `'t'`. In Python, it's an error to add `'t'` (a string) to 0 (an `int`), so that line crashes.

11) What string values for `a` and `b` make this code print just the string `done`, and nothing else?

```
a = ???
b = ???
if not (len(a) >= len(b)):
    print('yes')
if len(a) == len(b):
    print('no')
print('done')
```

A. `a = 'cat'`
   `b = 'dog'`
B. `a = 'parrot'`
   `b = 'dog'`
C. `a = 'cat'`
   `b = 'parrot'`
D. There **are** strings values of `a` and `b` that make the code print just `done`, but none of the above options A, B, or C do that.
E. There are **no** possible string values for `a` and `b` that make the code print just `done`.

Explanation: To print just `'done'`, the conditions of the two if-statements must both evaluate to false. The means `len(a) >= len(b)` must be true, and `len(a) != len(b)` must also be true. Any strings `a` and `b` for which both `len(a) >= len(b)` and `len(a) != len(b)` are true will work. So B is the only choice that makes those two expressions true.

12) Which code fragment prints `good` *just* when the lengths of strings `a`, `b`, and `c` are *all* different, and `bad` in every other case?

A.
```
if len(a) != len(b) and len(b) != len(c):
    print('good')
else:
    print('bad')
```

B.
```
if not (len(a) == len(b) and len(b) == len(c)):
    print('good')
else:
    print('bad')
```

C.
```
if not (len(a) == len(b) or len(a) == len(c) or len(b) == len(c)):
    print('good')
else:
    print('bad')
```

D. All of A, B, or C
E. None of A, B, or C

Explanation: If `a`, `b`, and `c` are strings, then they are all different if `a !=b`, `a != c`, and `b != c`. All three of those inequalities are needed to ensure that they are all different.

Another way to think about this is that if any two of the strings are equal, then they are not all different. And so they are all different when it's not the case that one or more paisr of them are equal.

So C is the correct answer. Note that `not (len(a) == len(b) or len(a) == len(c) or len(b) == len(c))` is logically equivalent to the expression `len(a) != len(b) and len(a) != len(c) and len(b) != len(c))`.

This equivalence is an application of **De Morgan's Law**, which says that for any logical variables `p` and `q`, `not (p and q)` is logically equivalent to `(not p) or (not q)`.

13) What values of `a` and `b` make this code print 2?

```
if a < 0 or b < 0:
    print(a)
elif a < b < 0:
    print(b)
else:
    print(a + b)
```

A. `a` is 2, `b` is 2
B. `a` is 2, `b` is -1
C. `a` is -1, `b` is 2
D. `a` is -1, `b` is -1
E. none of the above values of `a` and `b` that make the code print 2

Explanation: Perhaps the easiest way to answer this question is to run the code for each of the values given in the options. If you do that you'll see that B is the correct answer.

We could also try to solve it by reasoning about the code. The code is an if-elif-else statement with two conditions:

- Condition 1: `a < 0 or b < 0`
- Condition 2: `a < b < 0`

If Condition 2 is true then b *cannot* be 2. So if this code prints 2, it *cannot* be due to `print(b)`.

However, it *is possible* for the program to print 2 if Condition 1 is true. If `a` is 2, then `a < 0 or b < 0` (Condition 1) is true when `b` is *any* negative number. So we can check if any of the options fit this, and indeed in option B `a` is 2 and `b` is -1, and so that will cause the code to print 2.

14) What function call returns the same value as `f('4')` ?

```
def f(c):
    result = 0
    if c in '0123456789':
        if c in '01':
            result += int(c)
        if c in '02468':
            result += int(c) - 1
        else:
            result += int(c)
    else:
        result = -1

    return result
```

A. `f('2')`
B. `f('3')`
C. `f('5')`
D. `f('6')`
E. none of the above return the same value as `f('4')`

Explanation: Lets trace the function call `f('4')`. When it's called, `c` is set to `'4'` and `result` is set to 0. The condition `c in '0123456789'` is true because `'4'` is a substring of `'0123456789'`. Then `c in '01'` is false, and `c in '02468'` is true, so the statement `result += int(c) - 1` is called. The expression `int('4') - 1` evaluates to `4 - 1`, which is 3. Then that 3 is added to `result`, and so the final value of `result` is 3, which is the value that's returned.

15) If variables `a` and `b` are both strings, what are the possible values of this expression?

```
(a == b) or (a != b)
```

A. it always evaluates to `True`
B. it always evaluates to `False`
C. depending upon the values of `a` and `b`, sometimes it evaluates to `True`, and sometimes it evaluates to `False`

Explanation: Any two strings are either the same, or not the same. If `a` and `b` are strings, that means either `a == b` is true, or `a != b` is true. Thus the combined expression `(a == b) or (a != b)` must *always* be true.

16) What does this print?

```
x = 2
result = 1
for i in range(5):
    if i > x:
       result += i
print(result)
```

A. 7
B. 8
C. 12
D. 13
E. it prints an `int` other than 7, 8, 12, or 13

Explanation: Running this code shows that the answer is 8. To trace it by hand, note that the for-loop assigns `i` the values 0, 1, 2, 3, 4. The if-statement causes only the values greater than 2 to be added to `result`. So 3 and 4 get added to `result`, and since its initial value is 1 the final printed value is 1 + 3 + 4 = 8.

17) What does this print?

```
s = 'orange'
result = ''
for i in s:
    if i < 'k':
        result += i
print(result)
```

A. the empty string: the final value of `result` is the empty string
B. `age`
C. `orn`
D. nothing: the program crashes when `i < 'k'` is evaluated
E. a string other than `age`, `orn`, or the empty string

Explanation: Running this code shows that the answer is `age`. To trace it by hand, note that the for-loop assigns `i` the values `'o'`, `'r'`, `'a'`, `'n'`, `'g'`, `'e'`. In the loop, the if-condition `i < 'k'` is true just when `i` is a string that is alphabetically before `'k'`, and so only the letters of `'orange'` that come before `'k'` alphabetically get appended to `result`. Thus the final printed answer is `age`.

18) What does this print?

```
lst = [4, 0, 9, 1]
result = 0
for i in range(len(lst)):
    result += lst[i] + i
print(result)
```

A. 6
B. 14
C. 20
D. an `int` other than 6, 14, or 20

Explanation: Running this code shows that the answer is 20. To trace it by hand, note that since `lst` has four elements the for-loop makes `i` take on the values 0, 1, 2, 3. Then the value of both `i` and `lst[i]` are added to `result`. So what gets printed is the value of (4 + 0) + (0 + 1) + (9 + 2) + (1 + 3), namely 20.

19) What does this print?

```
result = 'start'
for s in ['up', 'moose', 'elephant', '!']:
    if len(s) < len(result):
        result = s
print(result)
```

A. `start`
B. `up`
C. `moose`
D. `elephant`
E. `!`

Explanation: Running this code on the computer will show that the answer is `!`. To trace it by hand, note that `s` is assigned `'up'`, `'moose'`, `'elephant'`, `'!'`. The if-condition `len(s) < len(result)` is true just when the number of characters in `s` is less than the number of characters in `result`. Whever that condition is true it sets `result` to be that value of `s`. By carefully tracing through each step, you find that `'!'` is the correct answer.

20) What does this print?

```
result = 0
for i in range(2, 5):
    for j in range(4):
        result += 1
print(result)
```

A. 4
B. 11
C. 12
D. 13
E. 16

Explanation: This code has a nest-looped, i.e. a loop within loop. For the outer loop `i` takes on the values 2, 3, 4, and for each of those values `j` is assigned 0, 1, 2, 3. That means the statement `result += 1` is executed 3 * 4 = 12 times.

21) What does this print?

```
result = 0
i = 6
while i < 10:
    result += i
    i += 2
print(result)
```

A. 14
B. 20
C. 30
D. an `int` other than 14, 20, or 30
E. it doesn't print an `int`

Explanation: The core of this code is the while loop. The variable `i` starts at 6, and every time the loop body is executed, it is incremented by 2. The values `i` has *in* the loop body are 6, 8 (when `i` is 10, the loop body is skipped), and each of these values get added to `i` by `result += i`. So the final printed value is 6 + 8 = 14.

22) What does this print?

```
i = 4
result = -1
while i >= 0:
    if (i + 1) % 2 == 1:
        result = i
    i += -1
print(result)
```

A. 0
B. 1
C. 2
D. 4
E. 5

Explanation: The core of this code is the while-loop. The variable `i` is initially 4, and each time the loop body executes it is *incremented* by -1 (which is the same as *decrementing* it by 1). The values of `i` in the loop are 4, 3, 2, 1, 0 (when `i` is -1 the loop body is skipped). For each of those values, the condition `(i + 1) % 2 == 1` is checked, and it's true just when `i + 1` is odd, or, equivalently, `i` is even. That means that `result` is assigned the value of `i` whenever `i` is even. Since 0 is the final even value of `i` in the loop, that's the value that is printed.

23) What does this print?

```
s = 'apple'
i = 1
result = '!'
while i < len(s):
    if s[i - 1] == s[i]:
        result += s[i]
    i += 1
print(result)
```

A. !
B. !p
C. !pp
D. !pl
E. nothing: the program crashes when run

Explanation: The core of this code is the while-loop. Every time through the loop `i` is incremented by 1, and the loop stops when `i` is equal to, or greater than, then length of `s`. The values of `i` in the loop are 1, 2, 3, 4. These values of `i` are exactly the index values of `s`, and so we can think of the loop as looping through all the index values of `s`. The condition `s[i - 1] == s[i]` is true just when two adjacent characters in `s` are the same. Since `'pp'` are the only adjacent letters in `s that` are the same, `'p'` gets appended to the end of `result`, and the final printed value is `!p`.

24) What does this print?

```
s = 'mysterious'
i = 0
flag = False
while not flag:
    if s[i] in 'aeiou':
        flag = True
        i += 1
    else:
        i += 2
print(s[i])
```

A. `e`
B. `i`
C. `o`
D. `r`
E. nothing: the `print` statement is never called

Explanation: The core of this code is the while-loop, and the idea is it is searching for a vowel in `s`. Each time through the loop body, if the character `s[i]` is a vowel then `flag` is set to true (indicating a vowel was found) and `i` is incremented by 1. If `s[i]` isn't a vowel, then all that's done is `i` is incremented by 2; this has the effect that when a non-vowel is encountered, the next letter is skipped over. So overall, the final value of `i` is 0 + 2 + 2 + 1 = 5, and final printed value is `s[5]`, which is `r`.

25) What does this print?

```
n = 64
while n > 1:
    n = n / 2
print(n)
```

A. 0.0
B. 0.5
C. 1.0
D. 2.0
E. nothing: the `print` statement is never called

Explanation: Each time through the loop the value of `n` is cut in half. So the values of `n` in are 64, 32, 16, 8, 4, 2, 1. The final printed value of `n` is 1.

This is **Code Listing 1**, referred to in the next few questions:

```
def print_n(s, n):          # line 1
    for i in range(n):      # line 2
        print(s)            # line 3

def f(n):                   # line 4
    if n % 2 == 0:          # line 5
        return n // 2       # line 6
    else:                   # line 7
        return 3 * n + 1    # line 8

def main():
    a = 3                   # line 9
    b = int(f(a + 1))       # line 10
    print_n('Kermit', b)    # line 11
```

Code Listing 1

26) In Code Listing 1, when `main()` is called, how many times is `Kermit` printed?

A. 0
B. 1
C. 2
D. 3
E. more than 3 times

Explanation: When `main` is called, since `a` is 3 the call `f(a + 1)` is equal to `f(4)`. When `f(4)` is called, the first if-statement body is called (because 4 is even), and so `f(4)` returns 2. `int(2)` is 2, so `b` is set to 2. This calls `print_n('Kermit', 2)`, which prints `Kermit` exactly 2 times.

27) In Code Listing 1, `main` has two local variables.

A. True
B. False

Explanation: `a` and `b` are the two local variables in `main`.

28) In Code Listing 1, how many times is `Kermit` printed if line 9 is changed to `a = 4` ?

A. 0
B. 1
C. 2
D. 3
E. more than 3

Explanation: If `a` is 4, then `f(a + 1)` is `f(5)`, which returns 3*5 + 1 = 16. Since `int(16)` is 16, `b` gets set to 16. This prints `Kermit` 16 times.

29) In Code Listing 1, `f(1) + f(4)` evaluates to 7.

A. True
B. False

Explanation: `f(1)` evaluates to 3 * 1 + 1 = 4, and `f(4)` evaluates to 2, and `f(1) + f(2)` is 6. Thus the correct answer is B, false.

30) In Code Listing 1, if line 2 was changed to `for i in range(2, n + 2)`, then the program would print the same thing as if the change was not made.

A. True
B. False

Explanation: The initial line 2 is `for in range(n)`, which executes its loop body `n` times. If it's changed to `for i in range(2, n + 2)`, then the loop body is still executed `n` times, i.e. `i` takes on the `n` values 2, 3, 4, …, `n` + 1, `n` + 2.

In general, `for i in range(a, b)` executes its loop body `b - a` times.

31) In Code Listing 1, if function `main()` was moved to be defined before function `print_n()`, the program would print the same thing as if the change was not made.

A. True
B. False

Explanation: The order in which Python functions does not matter in this case. Even though in the source code `print_n` is called before seeing it's definition, when the program runs `print_n` will have been defined.

32) In Code Listing 1, if lines 9, 10, and 11 had their indent removed so that they each start in the same column as the `d` in `def` on line 4, then calling `main()` would print `Kermit` twice.

A. True
B. False

Explanation: In Python, consistent indentation of at least 1 space is needed to indicate what lines are in the body of a function (or other structure). Thus, removing the indentation will effectively make the body of main empty, which is an error (a function always needs at least one statement).

33) Consider this code:

```
def reset(n):
    n = 0

def test1(x):
    x = 1
    reset(x)
    print(x)

def test2():
    n = 1
    reset(n)
    print(n)
```

i) Calling `test1(0)` prints 0.
ii) Calling `test2()` prints 0.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is true and ii) is false
D. i) is false and ii) is true

Explanation: When an `int` is passed in a function call, the function gets a *copy* of the `int`, and does *not* have access to the original value of the passed-in variable. This is called **pass by value**. When `test1(0)` is called, `x` is set to 1, and `reset(x)` does *not* change the value of `x` in `test1` (`reset` only changes the copied value that is in its body); thus 1 is printed. Similarly, `test2()` prints 1.

34) Suppose we want a function that takes a string `s` as input and returns a new string as follows:

- If `s` *ends* with a newline character, then the returned string is the same as `s` except that the one newline at the end has been removed.
- If `s` *does not end* with a newline character, then the returned string is the same as `s`.

Here are two possible implementations of this function:

```
def chop1(s):
    if s == '':
        return s
    elif s[-1] == '\n':
        return s[:len(s) - 1]
    else:
        return s
```

```
def chop2(s):
    n = len(s)
    if n == 0:
        return s
    elif s[n] == '\n':
        return s[:n-1]
    else:
        return s
```

A. both are **correct** implementations
B. both are **incorrect** implementations
C. `chop1` is a **correct** implementation, and `chop2` is an **incorrect** implementation
D. `chop1` is an **incorrect** implementation, and `chop2` is a **correct** implementation

Explanation: `chop1` is correct, and `chop2` is incorrect. The problem with `chop2` is the line `elif s[n] == '\n'`: since `n` is the length of `s`, `s[n]` is *not* a valid character (`s[n-1]` is its last character). `chop1` works correctly: the expression `s[-1]` is a standard Python way of accessing the last character of a string.

35) Suppose this code correctly opens the non-empty text file named `errors.txt`:

```
f = open('errors.txt')
```

How can you print the **first** line of `errors.txt`?

A. `print(f[0])`
B. `print(f.read())`
C. `print(f.readline())`
D. all of the above print the first line

Explanation: If `f` is a file object (i.e. a value returned by a call to `open`), then `f.readline()` returns the next line of the file. So the first time it is called, it returns the first line of the file. `f[0]` is an error, and `f.read()` returns that file contents as one big string.

36) Suppose this code correctly opens the text file named `animals.txt`:

```
f = open('animals.txt')
```

Which statement prints the total number of characters in `animals.txt` ?

A. `print(sum(f))`
B. `print(len(f))`
C. `print(f.size())`
D. `print(len(f.read()))`

Explanation: `f.read()` returns the contents of the file as a string, and so `len(f.read())` is the number of characters in the file. It essentially converts the text file into a string and counts the number of characters in the string.

37) Suppose this line of code correctly opens the text file named `data.txt`:

```
f = open('data.txt')
```

`f` is open:

A. just for reading
B. just for writing
C. for both reading and writing
D. neither reading nor writing

Explanation: `open('data.txt')` opens the file just for reading. So does `open('data.txt', 'r')`. The statement `open('data.txt', 'w')` would open it for writing.

38) Which function returns the index location of the `int x` in a list `lst`? Assume `x` occurs exactly once in `lst`.

A.
```
def search1(x, lst):
    for i in range(len(lst) - 1):
        if lst[i] == x:
            return i
    return -1
```

B.
```
def search2(x, lst):
    i = 0
    while i < len(lst):
        if lst[i] == x:
            return i
        i += 1
        return -1
```

C.
```
def search3(x, lst):
    for i in lst:
        if i == x:
            return i
    return -1
```

D.
```
def search4(x, lst):
    i = 0
    while i < len(lst):
        if lst[i] == x:
            return i
        i += 1
    return -1
```

E. none of the above

Explanation: D is the correct answer. It uses a while loop to check the value of the list against `x`. The other options are incorrect because:

- `search1` does *not* check the last element of `lst`
- `search2` has the `return` statement indented incorrectly (it's in the while loop body, but should be indented to line of up with the `w` of `while`)
- `search3` returns the value in the list, not its index. Calling the loop variable `i` is mis-leading because `i` usually means an index, but here it's the value in the list.

39) What does this print?

```
def f(lst, target):
    for i in range(len(lst)):
        if lst[i] + 5 == target:
            return i
    return -1

data = [10, 3, 6, 5, 2, 7]
print(f(data, 6))
```

A. 6
B. 5
C. 2
D. -1
E. an `int` other than 6, 5, 2, or -1

Explanation: The core of this code is the for-loop in `f`. The loop assigns `i` the values 0, 1, 2, …, n-1 (where n is the length of `lst`). If 5 plus the value at `lst` location `i` is equal to the `target`, then `i` is immediately returned. Since the `target` is 6, the code returns `i` when it finds a 1 somewhere in `lst`. Since there is no 1 in `lst`, -1 is returned.

40) Here are two possible implementations of a function that is meant to return the sum of a list of numbers:

```
def addem1(lst):
    for n in lst:
        result += n
    return result
```

```
def addem2(lst):
    result = 0
    i = len(lst) - 1
    while i >= 0:
        result += lst[i]
    return result
```

A. both are **correct** implementations
B. both are **incorrect** implementations
C. `addem1` is a **correct** implementation, and `addem2` is an **incorrect** implementation
D. `addem1` is an **incorrect** implementation, and `addem2` is a **correct** implementation

Explanation: `addem1` is incorrect because the `result` variable is not initialized; the first line should be `result = 0`. `addem2` is incorrect because the index variable `i` never changes after it is initialized. The statement `i -= 1` or `i = i - 1` should be the second line in the body of the while loop.

41) What does this print?

```
lst = [4, 5, 1, 3, 2]
m = lst[0]
for x in lst:
    if x < m:
        m += x
print(m)
```

A. 0
B. 1
C. 2
D. 10
E. an `int` other than 0, 1, 2, or 10

Explanation: The core of this code is the for-loop, and it assigns `x` the values 4, 5, 1, 3, 2 (the values of `lst`). By carefully tracing the code step-by-step, you can determine that it prints 10. The 5 in `lst` is *not* added, but the other values are. So the final value of m is 4 + 1 + 3 + 2 = 10.

42) What value of `x` makes this program print 3?

```
lst = [2, x, 1, 1, 3, 1, 3]
print(lst.count(lst[1]))  # prints 3
```

A. 0
B. 1
C. 2
D. 3
E. an `int` other than 0, 1, 2, or 3

Explanation: `lst.count(a)` returns the number of times `a` occurs in `lst`. If `x` is set to 3, then there will be three 3s in `lst`, which is the correct answer.

43) What is the biggest number that this code prints?

```
for x in [0, 1, 2, 3, 4]:
    A = [x, 2, 1, 1, 2, 2]
    A[4] = x
    B = [A.count(1), A.count(2)]
    print(B.count(1) + B.count(2))
```

A. 0
B. 1
C. 2
D. 3
E. an `int` bigger than 3

Explanation: `lst.count(a)` returns the number of occurrences of `a` in `lst`. To answer this you should carefully trace through the code line by line, being careful to do exactly what each step says.

44) If you run binary search on this list, what is the first value the search checks?

```
[4, 5, 7, 9, 10, 11, 15, 16, 20]
```

A. 4
B. 9
C. 10
D. 11
E. an `int` other than 4, 9, 10, or 11

Explanation: Binary search always starts searching in the middle of a list, and so the first value checked here is 10.

45) Consider these statements:

i) Linear search requires that the data it is searching be in sorted order.
ii) Binary search requires that the data it is searching be in sorted order.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is true and ii) is false
D. i) is false and ii) is true

Explanation: Linear search does *not* require that the data it is searching be in any particular order. But binary search does require that the list be in sorted order, from smallest to biggest.

46) In the worst case, about how many comparisons does **selection sort** do to sort a list of n ints?

A. n
B. $2n$
C. n^2
D. n^3
E. 2^n

Explanation: Selection sort is a quadratic algorithm, which means it has a worst-case running time of approximately n^2.

47) What is the minimum number of comparisons (i.e. calls to `<`) needed to check if a list of 50 different numbers is in ascending sorted order?

A. 48
B. 49
C. 50
D. 51

Explanation: To check if 50 numbers are in sorted order, imagine putting the numbers in a line and inserting `<` between them to make a big boolean expression. If this expression is true, then the numbers are in sorted order. Since there are 50 numbers, 49 `<` are needed.

48) What does this print?

```
x = 0
for i in range(1, 100):
    if i % 2 == 0:
        x += 1
print(x)
```

A. 49
B. 50
C. 51
E. an `int` other than 49, 50, or 51

Explanation: The core of this code is the for-loop which assigns `i` the values 1, 2, 3, …, 99. The expression `i % 2 == 0` is true just when `i` is even, in which case it increments `x`. Thus this code counts the number of *even* numbers from 1 to 99. From 1 to 99 there are exactly 49 even numbers.

49) What does this print?

```
x = 0
for i in range(10):
    for j in range(15):
        x += 1
print(x)
```

A. 23
B. 25
C. 126
D. 150
E. an `int` other than 23, 25, 126, or 150

Explanation: The core of this code is a loop within a loop (a nested loop). `i` is assigned the values 0, 1, 2, …, 9, and for each one of those values `j` is assigned the values 0, 1, 2, …, 14. So for each pair of `i` and `j` it increments `x`, and since there are 10 * 15 = 150 pairs, 150 is the printed answer.

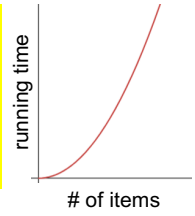
50) What does this print?

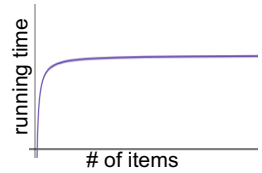
```
x = 0
for i in range(5):
    for j in range(5):
        if i != j:
            x += 1
print(x)
```

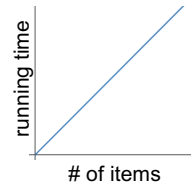
A. 5
B. 12
C. 20
D. 25
E. an `int` other than 5, 12, 20, or 25

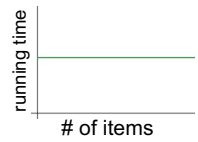
Explanation: The core of this code is a loop within a loop (a nested loop). `i` is assigned the values 0, 1, 2, 3, 4, and for each of those values the variable `j` is assigned 0, 1, 2, 3, 4. The inner if-statement `if i != j` runs exactly 5 * 5 = 25 times, and `x` is incremented whenever `i` and `j` are different. Since `i` and `j` are the *same* at 5 different times, then they are *different* at 25 – 5 = 20 times. Thus the final printed value of `x` is 20.

51) Which graph best describes the worst-case running-time of the **selection sort** algorithm?

A. 

B. 

C. 

D. 

Explanation: Selection sort is a quadratic algorithm, which means its running time is a quadratic curve, i.e. a parabola. So A is the correct answer since it is the only parabola among the options.

52) What is a recursive function? A function that:

A. does not called any other functions
B. has no loops
C. calls itself
D. calls itself, and does not call any other functions

Explanation: The essential feature of a recursive function is that it calls itself (either directly or indirectly). A recursive function can call functions other than itself. While recursion is often used to replace loops, having no loops is *not* a requirement for being recursive. It is certainly possible for a recursive function to contain a loop.

53) Consider these statements:

i) Any recursive function can be re-written as an equivalent function (or functions) that doesn't use recursion.
ii) Any function that uses loops can be re-written as an equivalent function (or functions) that uses recursion instead of loops.

A. i) and ii) are both true
B. i) and ii) are both false
C. i) is true and ii) is false
D. i) is false and ii) is true

Explanation: Both statements are true. It is a fundamental fact of recursion that it can simulate any loop, and that any loop can be implemented using recursion. Indeed, there are programming languages without loops, such as Haskell or Prolog.

54) What does this print?

```
def g(n):
    if n <= 0:
        return 0
    else:
        return g(n - 2) + n

print(g(g(3)))
```

A. 4
B. 5
C. 6
D. an `int` other than 4, 5, or 6
E. nothing: it never returns

Explanation: To calculate `g(g(3))`, first calculate `g(3)`. Since `g` is a recursive function it can be calculated like this:

g(3) = g(1) + 3
= (g(-1) + 1) + 3
= (0 + 1) + 3
= 4

Thus `g(g(3))` returns the same value as `g(4)`. From the definition of `g`:

g(4) = g(2) + 4
= (g(0) + 2) + 4
= (0 + 2) + 4
= 6

This shows `g(g(3))` is 6, which is what the program prints.

55) What does this print?

```
def h(n):
    if n == 0:
        return 0
    else:
        return h(n - 1)

print(h(99))
```

A. 0
B. 1
C. 98
D. 99
E. none of the above

Explanation: In this recursive function, `h(n)` is replaced with `h(n-1)` in the recursive case. For example `h(3)` has the same value as `h(2)`, which has the same value as `h(1)`, which has the same value as `h(0)`. Since `h(0)` = 0, then `h(3)` = 0. The same reasoning applies to `h(99)`: `h(99)` = `h(98)` = `h(97)` = … = `h(2)` = `h(1)` = 0. Thus `h(99)` is 0. In fact, `h(n)` is 0 for all `int`s greater than, or equal to, 0.

56) What is pseudocode?

A. the generic name of the language that Python is automatically converted to just before it runs on a real computer
B. the generic name for any programming language, such as Python, that contains English words in it
C. a description of an algorithm/program designed for human reading
D. source code with one or more bugs in it

Explanation: Pseudocode is English-like code that is designed to for human reading. It's often used to communicate algorithms to other people, or sometimes to help design a program, e.g. you might sketch out the how a function works by writing in pseudocode.

Option A describes **bytecode**.

57) Which application is a **BAD** fit for Python?

A. data science, e.g. processing and displaying data
B. machine learning scripting, e.g. processing data and running learning algorithms
C. high-performance real-time systems, such as airplane control software
D. back-end web development

Explanation: Option C is correct because, compared to languages like C/C++, Java, Rust, etc., Python runs relatively slowly, e.g. 10-100 times slower depending on the situation. Thus, it is not a good language to use for high-performance real time systems where it can important that a calculation be finished in some small amount of time. The other options are popular applications of Python.

58) What does this print?

```
lst = [4, 1, 3, 2, 5]
lst = lst[1:4] + lst[:2]
lst.sort()
lst.reverse()
print(lst[1] - lst[3])
```

A. -2
B. -1
C. 1
D. 2
E. an `int` other than -2, -1, 1, or 2

Explanation: `lst[a:b]` is **slice notation**, which returns a copy of the list starting at `lst[a]`, and going up to, and including `lst[b-1]` (`lst[b]` is *not* part of the slice). The expression `lst[:b]` returns a list that is a copy of the elements from `lst[0]` to `lst[b-1]`. So we can trace the code like this:

```
lst = [4, 1, 3, 2, 5]
lst = lst[1:4] + lst[:2]  # lst is [1, 3, 2] + [4, 1] = [1, 3, 2, 4, 1]
lst.sort()                # lst is [1, 1, 2, 3, 4]
lst.reverse()             # lst is [4, 3, 2, 1, 1]
print(lst[1] - lst[3])    # lst[1] - lst[3] = 3 - 1 = 2
```

Thus the final printed answer is 2.

59) What doe this print?

```
s = 'abcde'
x = s[2:5] + s[1:4] + s[:3]
print('dab' in x)
print('deb' in x)
```

A.
```
False
False
```
B.
```
False
True
```
C.
```
True
False
```
D.
```
True
True
```

Explanation: For a string `s`, the expression is **slice notation** `s[a:b]` that evaluates to a new string consisting of the characters `s[a]` to `s[b-1]` (`s[b]` is *not* included). `s[:b]` is the same as `s[0:b]`. So we get this:

- `s[2:5]` is `'cde'`
- `s[1:4]` is `'bcd'`
- `s[:3]` is `'abc'`

Adding these three strings together gives `x` the value `'cde'` + `'bcd'` + `'abc'` = `'cdebcdabc'`. We can see that both `'dab'` and `'deb'` are substrings. So D is the correct answer.

60) What does this print?

```
a, b, c, = 1, 'two', [3, 4]
c, a, b = b, c, a
print(2*a, 2*b, 2*c)
```

A. `[6, 8] 2 twotwo`
B. `[3, 4, 3, 4] 4 twotwo`
C. `[3, 4] 1 two`
D. the code prints something, but none of the above
E. nothing: the code has an error

Explanation: A statement like `x, y, z = 1, 2, 3` is the same as the three separate statements `x = 1`, `y = 2`, and `z = 3`. The statement `c, a, b = b, c, a` is the same as `a, b, c = 'two', [3, 4], 1`. Thus `a` gets the value `'two'`, `b` gets `[3, 4]`, and `c` gets 1. Then `print(2*a, 2*b, 2*c)` prints `[3, 4, 3, 4] 4 twotwo`.