CMPT 120 Standard Final Exam

Sample 3
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('1' * 2 + '3')
```

A. 5
B. 113
C. 123
D. 223
E. nothing: the statement has an error

Explanation: `'1' * 2 + '3'` simplifies to `'11' + '3'`, which is the same as `'113'`. So 113 is printed.

2) Which program prints the same thing as this one?

```
a = 2
b = 2
a = b - a
b = b - a
print(a)
print(b)
```

A.	B.	C.	D.

`a = 2`
`b = 2`
`a = a + b`
`b = a - b`
`print(a)`
`print(b)`
	`a = 2`
	`b = 2`
	`a = b - a`
	`b = b + a`
	`print(a)`
	`print(b)`
		`a = 2`
		`b = 2`
		`b = b + a`
		`b = b - a`
		`print(a)`
		`print(b)`
			`a = 2`
			`b = 2`
			`b = a - b`
			`a = a + b`
			`print(a)`
			`print(b)`

Explanation: By tracing the program given in the question you can see that it prints 1 and then 2. By tracing the other code examples, C is the only other one that prints 1 and then 2.

3) Assuming `a` is initialized 2 and `b` is initialized to 6, which code fragment prints 6 on one line, and then 2 on the next line?

A.	B.	C.	D.

`tmp = a`
`b = a`
`a = tmp`
 	
`print(a)`
`print(b)`
	`tmp = b`
	`a = b`
	`b = tmp`
	
	`print(a)`
	`print(b)`
		`tmp = b`
		`b = a`
		`b = tmp`
		
		`print(a)`
		`print(b)`
			`tmp = a`
			`a = b`
			`b = tmp`
			
			`print(a)`
			`print(b)`

Explanation: This question is asking which code fragment *swaps* the values of `a` and `b`. D is the one correct answer because it first saves the value of `b` in `tmp`, then overwrites `b` with value of `a`, and then finally assign the saved value of `a` to `b`.

4) Consider this statement:

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

How many of these 4 arithmetic operators could replace ??? so that it 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) You *can* change the length of a Python string
ii) You *can* change the length of a Python list

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

Explanation: Python strings are *immutable*, i.e. they cannot be changed in any way. However, lists are *mutable*, i.e. they can be changed by adding/removing/changing elements. Thus i) is false and ii) is true.

6) What does this print?

```
lst = [2, 0, 1, 3]
print(lst[lst[2]] + lst[lst[3]])
```

A. 1 [✓ C. 3] E. nothing: there is an error
B. 2 D. some `int` other than 1, 2, or 3

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

```
print(lst[lst[2]] + lst[lst[3]])  # original statement
print(lst[1]      + lst[3]     )
print(0           + 3          )
print(3)
```

7) What does this print?

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

[✓ A. `[2, 0, 3]`] B. `[0, 1, 3]` C. `[2, 1, 3]` D. nothing: the code has an error

Explanation: Since `scores` is a list, after running the assignment statement `T = scores` the variable `T` *refers* to the same list as `scores`. It does *not* make a copy. So `T[1] = 0` changes the list that `scores` refers to.

If you did want to make a copy you could use the statement `T = scores[:]`

8) What does this print?

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(len(lst[2:5]))
```

A. 5 B. 4 [✓ C. 3] D. 2

Explanation: `lst[2:5]` is a *slice expression* equal to `[3, 4, 5]`, which is a list of length 5 – 2 = 3. In general, if `L` is a list, then `L[a:b]` is a slice expression consisting of a new list with all the elements from `L[a]` to `L[b-1]` (`L[b]` is *not* part of the slice). Assuming `0 <= a <= b`, the length of `L[a:b]` is `b - a`.

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. it crashes with an error when run

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

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

For a Python dictionary, if you write `d[5] = 2` then `5:2` is guaranteed to be in `d`. If 5 is not already in the dictionary, then `5:2` is added to it. If 5 is already a key in `d` then it's corresponding value will get over-written to be 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** string values of `a` and `b` that make the code print just `done`, but none of 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) Suppose `x`, `y`, and `z` are `int` variables (but we don't know which `int`s exactly). Consider these statements:

i) if `[x, y, z] == [y, y, y]` evaluates to `True`, then `x`, `y`, and `z` all equal the same value

ii) if `x`, `y`, and `z` all equal the same value, then `[x, y, z] == [y, y, y]` evaluates to `True`

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 i) and ii) are true. To see this, note that `[x, y, z] == [y, y, y]` can be written as the equivalent expression `(x == y) and (y == y) and (z == y)`.

For i), if `[x, y, z] == [y, y, y]` is true then `(x == y) and (y == y) and (z == y)` must be true. If `(x == y) and (y == y) and (z == y)` is true then all three variables are equal to `y`, meaning they all have the same value. So i) is a true statement.

For ii), if `x`, `y`, and `z` are all equal, then `(x == y) and (y == y) and (z == y)` must also be true, which means `[x, y, z] == [y, y, y]` is true. So ii) is a true statement.

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

Explanation: Tracing the code for each option shows that B is the correct answer.

You 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('0')` ?

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

    return total
```

A. `f('1')` B. `f('2')` C. `f('3')` D. none of A, B, or C

Explanation: By tracing each call to `f`, you can see that `f('0')` returns -1, `f('1')` returns 2, `f('2')` returns 1, and `f('3')` returns 3. So none of the options return the same value as `f('0')`.

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

```
(a < b) or (b < a)
```

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: `a < b` is true just when `a` comes alphabetically before `b`, and similarly `b < a` is true just when `b` comes alphabetically before `a`. So the combined expression `(a < b) or (b < a)` is true just when one of the strings comes alphabetically before the other. If `a` and `b` are equal, then `a < b` is false, and `b < a` is false, meaning `(a < b) or (b < a)` is false. Thus C is the correct answer since the expression can be true or false, depending on `a` and `b`.

16) What does this print?

```
cutoff = 10
result = 0
for i in range(20):
    if i < cutoff:
       result += 1
print(result)
```

A. 9   B. 10   C. 11   D. 19   E. 20

Explanation: Tracing this code shows that the answer is 10. To trace it by hand, note that the for-loop assigns `i` the values 0, 1, 2, …, 19, and the if-statement counts values less than 10. So when i is one of the ten values 0, 1, 2, …, 8, 9 then `result += 1` is executed, and thus the final printed value is 10.

17) What does this print?

```
s = 'soccer'
result = 0
for i in range(len(s)):
    if s[i] <= s[i + 1]:
        result += 1
print(result)
```

A. 0   B. 1   C. 2   D. 3   E. nothing: the program crashes due to an error

Explanation: The code crashes due to an error when it runs. The problem is that when `i` is 5 (the last time through the loop), then `s[5 + 1]` is `s[6]`, which does not exist (the last value of `s` is `s[5]`) and so causes an error when evaluated.

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. 15 D. 20

Explanation: Running this code shows that the answer is 20. To trace it by hand, note that since `lst` has four elements and so 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`. The final printed value is (4 + 0) + (0 + 1) + (9 + 2) + (1 + 3) = 20.

19) What value of `lst` makes this print `!` ?

```
result = 'start'
for s in lst:
    if len(s) < len(result):
        result = s
print(result)
```

A. `['!', 'up', 'moose', 'elephant']`

B. `['up', '!', 'moose', 'elephant']`

C. `['up', 'moose', '!', 'elephant']`

D. `['up', 'moose', 'elephant', '!']`

E. all of A, B, C, and D

Explanation: Running this code on the computer will show that the answer is `!` for each given list. To trace it by hand, note that `s` is assigned each element of the list in order from left to right. If the current value of `s` is shorter than the shortest string seen so far, then `s` becomes the new shortest. So the code prints the shortest of all the strings in `lst` (and including `'start'`).

20) What does this print?

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

A. 12 B. 13 C. 16 D. 20

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

21) Which program prints the biggest number?

```
# program 1
result = 0
i = 0
while i < 5:
    i += 1
    result += i
print(result)
```

```
# program 2
result = 0
i = 0
while i < 5:
    result += i + 1
    i += 1
print(result)
```

A. program 1 prints the biggest number
B. program 2 prints the biggest number
C. they print the same number

Explanation: By tracing the code you can see that they both print the same number.

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: Tracing this code is the best way to see what it prints. You could also reason about it like this. The variable `i` is initially 4, and each time through the loop 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. 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?

```
a = 1
b = 20
while a * a <= b:
    a += 1
print(a)
```

A. 20   B. 19   C. 6   D. 5   E. nothing: the `print` statement is never called

Explanation: The core of this code is the while-loop, and the idea is that the loop body is executed if the square of `a` is less than, or equal to, `b`. It stops for the first value of `a` such that `a * a > b`. So the final printed answer is 5.

25) What does this print?

```
result = 0
i = 0
while i < 3:
    j = 0
    while j < 4:
        result += 1
        j += 1
    i += 1
print(result)
```

A. 6   B. 8   C. 12   D. 20   E. nothing: the `print` statement is never called

Explanation: In the outer while-loop, `i` takes on the values 0, 1, 2. For each of those values, `j` takes on the values 0, 1, 2, 3. So in total, `result` is incremented 3 * 4 = 12 times.

26) This program is meant to print the sum of the numbers in `L`, but it may have a bug. When the program runs, what line causes a *run-time error*?

```
L = [5, 1, 3]
i = 0                    # line 1
total = 0
while i <= len(L):       # line 2
   total = L[i] + total  # line 3
   i = i + 1             # line 4

print(total)
```

A. line 1   B. line 2   C. line 3   D. line 4   E. there is no error: the program correctly prints the sum of any list L of numbers

Explanation: The run-time error occurs on line 3: `total = L[i] + total`. The problem with is it allows the case `i == len(L)`, which inside the loop causes `L[len(L)]` causes an out of bounds error. To fix the error, the `<=` in line 2 should be changed to `<`.

27) What does this print?

```
a = 1
b = 2

def f(a):
    a = a + a
    a += a
    print(a)

f(b)
```

A. 8 B. 4 C. 2 D. nothing: there is an error in the program

Explanation: Since `b` is 2, `f(b)` is the same as `f(2)`. When `f(2)` is called, the local a variable in `f` is initialized to 2. Then the first line of `f` doubles `a` to 4, and the next line doubles it again, giving 8. So 8 is printed.

28) What does this print?

```
def f(n - 1):
    return n + 1

print(f(0) + f(1))
```

A. -1 B. 0 C. 1 D. 2 E. nothing: the program has an error

Explanation: The function header for `f` is incorrect: `n - 1` is allowed inside a function header like this. It should be `n`. So the program has an error and so prints nothing.

29) Consider this program:

```
x = 1
y = 4
x = x - y
y = y + x
???
print(x)
print(y)
```

What can replace `???` to make it print:

```
4
1
```

A. `y = y - x` B. `x = x - y` C. `x = y - x` D. `y = x + y`

Explanation: The program swaps the values of `x` and `y`. By tracing, the code up before `???` sets y to 1 and `x` to -3, and so `???` must assign `x`. Thus A and D can be removed as possible answers. By testing the two remaining options, the answer is C.

30) What does this print?

```
def f(n):
    result = n - 1
    for i in range(2, n + 1):
        result += i
    return result

print(f(3))
```

A. 5 B. 6 C. 7 D. 8

Explanation: By carefully tracing the code, you can see that it prints 0. When `f(3)` is called, `n` is initialized to 3 inside of `f`. Then `result` is 2, and the for-loop header is "`for i in range(2, 4):`", which means the values 2 and 3 are added to `result`. So the final value of `result` is 2 + 2 + 3 = 7, which is what the program prints.

31) What does this print?

```
def greet(s):
    say_hi(s)

def say_hi(s):
    print("Hi " + s + "!")

greet("Alice")
```

A. `Hi Alice!`

B. it runs without error and prints something other than `Hi Alice!`

C. the program has an error and so prints nothing

Explanation: The correct answer is option A. The main concern is that `say_hi` is called inside `greet` before it's definition. But that's okay, since when the program runs it will have seen the definition for `say_hi`.

32) If C is a temperature in Celsius, then this formula converts it to temperature F in Fahrenheit:

$$F = \left(\frac{9}{5}\right) C + 32$$

This function correctly converts Celius to Fahrenheit:

```
def to_fahrenheit(C):
    return (9/5)C + 32
```

A. True

B. False

Explanation: The expression `(9/5)C + 32` is incorrect: it is missing a *, and should be `(9/5) * C + 32`.

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)]
    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: Both `chop1` and `chop2` are incorrect. The problem with `chop1` is that the expression `s[:len(s)]` returns the entire string `s`, and so does *not* remove the newline. 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).

35) Suppose the non-empty text file `errors.txt` is opened correctly by the program below. What gets printed?

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

A. the first character of the file

B. the first line of the file

C. the entire file

D. nothing: the program has an error

Explanation: If `f` is a file object (i.e. a value returned by a call to `open`), then `f.read()` returns the file contents as one big string.

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

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

What does the program print?

A. the number of lines in the file
B. the number of characters in the file
C. the size of the file in bytes
D. nothing: the code has an error

Explanation: When `f` is a file object, `len(f)` is not a well-defined operation and so it causes an error when it runs.

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

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

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

38) Which function always 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):
    for i in lst:
        if i == x:
            return i
    return -1
```

C.
```
def search3(x, lst):
    i = 0
    while i < len(lst):
        if lst[i] == x:
            return lst[i]
        i += 1
    return -1
```

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

Explanation: D (`search4`) 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` returns the *value* in the list, not its *index*. Calling the loop variable `i` is mis-leading because `i` usually means an index, and here it's the value in the list.
- `search3` returns the *value* at location of `i` of the list, not the *index* `i` as required.

39) What does this print?

```
def f(lst):
    result = 0
    for i in range(1, len(lst)):
        if lst[i-1] < lst[i]:
            result = lst[i]
    return result

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

A. 2 B. 3 C. 5 D. 6 E. 7

Explanation: The core of this code is the for-loop in `f`. The loop looks at each pair of adjacent values "`a`, `b`", and if `b` is bigger than `a` then `b` is assigned to `result`. By inspecting the list, we can see that "3, 7" is the last pair of adjacent numbers where the left is less than the right, and so the 7 is the returned value.

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

```
def addem1(lst):
    result = 0
    for i in range(len(lst)):
        result += lst[i]
    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: `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]
acc = 0
for x in lst:
    if x > lst[0]:
        acc += x
print(acc)
```

A. 4 B. 5 C. 9 D. 10

Explanation: The core of this code is the for-loop, and it assigns `x` the values 4, 5, 1, 3, 2. `x` is only added to `acc` when it is bigger than 4 (the value of `lst[0]`), so the final value of `acc` is 5 (the only value bigger than 4).

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 last number that this code prints?

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

A. 1 B. 2 C. 3 D. 4 E. 5

Explanation: By inspecting the code, you can see that the value printed in the loop depends only on the current value of `x`. So the last value that's printed is when `x` is -1, in which case `A` is `[-1, 0, 2, 3]`, `B` is `[0, 1]`, and so `B[0] + B[1]` is 1, which is the last printed value.

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

```
[4, 5, 6, 10, 11, 12, 13, 16, 21]
```

A. 4 B. 10 C. 11 D. 12 E. an `int` other than 4, 10, 11, or 12

Explanation: Binary search always starts searching in the middle of a list, and so the first value checked here is 11.

45) Suppose `L` is a list of the numbers from 1, 2, …, 99, 100 in some random order, and linear search is used to determine if a given target number `x` is in `L`.

i) The fewest number of comparisons linear search might do to is 1 comparison.
ii) The greatest number comparisons linear search might do to is 99 comparisons.

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

Explanation: If `x` is the first element in `L`, then 1 comparison is done; so i) is true. If `x` is not in `L` (or is the last element of `L`), then 100 comparisons are done, and so ii) is false.

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) Marge has a bag filled with 5 marbles numbered 1, 2, 3, 4, 5. She repeats these steps until the bag is empty:

- She removes the *lowest-numbered* marble from the bag.
- She places it on the right end of the marbles already on the table.

When there are *n* marbles in the bag it takes Marge exactly *n* seconds to remove the smallest and place it on the table.

How many seconds in total would it take Marge to remove and place all 5 marbles?

A. 5 B. 10 C. 12 D. 15 E. 20

Explanation: The time to remove each marble is:

- 5s to remove the first marble (since there are 5 marbles in the bag)
- 4s to remove the second marble
- 3s to remove the third marble
- 2s to remove the fourth marble
- 1s to remove the fifth marble (since there is 1 marble in the bag)

So the total time to remove all marbles is 5 + 4 + 3 + 2 + 1 = 15 seconds.

48) What does this print?

```
x = 0
for i in range(10, 20):
    if i % 2 == 1:
        x += 1
print(x)
```

A. 4 B. 5 C. 6 E. an `int` other than 4, 5, or 6

Explanation: The core of this code is the for-loop which assigns `i` the values 10, 12, …, 18, 19. The expression `i % 2 == 1` is true just when `i` is odd, in which case it increments `x`. Thus this code counts the number of *odd* numbers from 10 to 19. From 10 to 19 there are exactly 4 odd numbers.

49) What does this print?

```
x = 0
for i in range(5):
    for j in range(5):
        x += 2
print(x)
```

A. 16 B. 25 C. 32 [✓ D. 50]

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 one of those values `j` is assigned the values 0, 1, 2, 3, 4. So for each pair of `i` and `j` it increments `x` by 2, and since that is done 5 * 5 = 25 times, and so 50 is the printed answer.

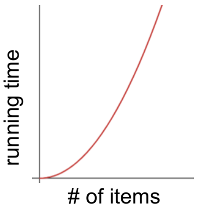
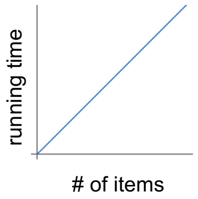
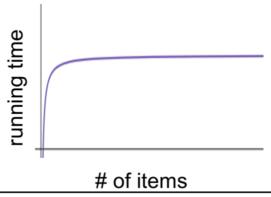
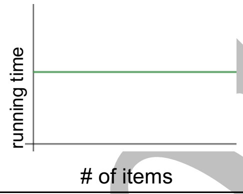
50) What does this print?

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

A. 9 [✓ B. 10] C. 18 D. 20 E. 90

Explanation: The core of this code is a loop within a loop (a nested loop). `i` is assigned the values 0, 1, …, 8, 9, and for each of those values the variable `j` is assigned 0, 1, …, 8, 9. The inner if-statement `if i == j` runs exactly 10 * 10 = 100 times, and `x` is incremented whenever `i` and `j` are the same. For each value of `i` there is one value of `j` that is the same, and so `i` and `j` are the *same* at 10 different times. Thus the final printed value of `x` is 10.

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

A.  	C. 
B.  	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) A function is recursive if it:

A. has no loops
B. does not call any other functions
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 - 1) + 2

print(g(3) + g(4))
```

A. 6 B. 8 C. 14 E. nothing: g never returns

Explanation: To calculate `g(g(3) + g(4))`, first calculate `g(4)`:

```
g(4) = g(3) + 2
     = (g(2) + 2) + 2
     = ((g(1) + 2) + 2) + 2
     = (((g(0) + 2) + 2) + 2) + 2
     = (((0 + 2) + 2) + 2) + 2
     = 8
```

Thus `g(4) = 8`. Since `g(4) = g(3) + 2`, we get `8 = g(3) + 2`, so `g(3) = 8 – 2 = 6`. Thus `g(3) + g(4) = 6 + 8 = 14`.

55) What does this print?

```
def h(n):
    if n <= 2:
        return n
    else:
        return h(n) - 1

print(h(4))
```

A. -1 B. 0 C. 2 D. nothing: `h(4)` does not return a value

Explanation: You can trace the function call to see that it does not return a value:

```
h(4) = h(4) – 1
     = (h(4) - 1) - 1
     = ((h(4) - 1) - 1) - 1
     = (((h(4) - 1) - 1) - 1) - 1
     = …
```

The expression that `h(4)` evaluates to gets bigger after each recursive call, so it never returns a value.

56) What is pseudocode?

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

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 C describes **bytecode**.

57) Consider these statements:

i) Python is a good language for implementing a high-performance real-time system, such as an operating system of 3D graphics systems
ii) Python is a good language for running (but not necessarily implementing) machine learning algorithms

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: Python strings are *immutable*, i.e. they cannot be changed in any way. However, lists are *mutable*, i.e. they can be changed by adding/removing/changing elements. Thus i) is false and ii) is true.

58) What does this print?

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

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

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

Thus the final printed answer is 2. Note that `lst.reverse()` makes no difference to the final result, and could be deleted.

59) What does this print?

```
s = 'abcde'
for i in range(3):
    s = s[1:4] + s[:2]
print(s)
```

A. `abcde` B. `dabcd` C. `eabcd` D. `bcdab`

Explanation: We can trace the code like this:

i	s	s[1:4]	s[:2]
0	abcde	bcd	ab
1	bcdab	cda	bc
2	cdabc	dab	cd
(after loop)	dabcd		

So the final value of `s` is `'dabcd'`.

60) What does this print?

```
a, b, c = 1, 'two', [3, 4]
c, a, b = a, c, b
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 program 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`. So the statement `c, a, b = a, c, b` is the same as `c, a, b = 1, [3, 4], 'two'`, i.e. `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] twotwo 2`, which is not one of the choices. So D is the correct answer.