Name: _____SOLUTION_____

Use this quiz to help you prepare for the Paper-and-Pencil portion of Exam 1. Print it and write your answers directly on the printed copy, or read the electronic copy and write your answers on your own document – your choice. ***Answer all questions.*** Make additional notes as desired. ***Not sure of an answer?*** Ask your instructor to explain in class and revise as needed then. ***Key problems* include: 7, 15, 16, 20, 21, 23, 24, 27, 32, and 33.**

Throughout, where you are asked to "circle your choice", you can circle or underline it (whichever you prefer).

Throughout, assume that there are no global variables (if you happen to know what they are).

1. Consider the ***secret*** function defined to the right. What are the values of:

```
def secret(x):
    y = (x + 1) ** 2
    return y
```

   a. **secret(2)** _____ **9** _____

   b. **secret(secret(2))** ___ **100** _____

2. Consider the ***mystery*** function defined to the right. What are the values of:

```
def mystery(x, y):
    result = x + (3 * y)
    return result
```

   a. **mystery(5, 2)** ____**11** _____

   b. **mystery(2, 5)** ____**17**_____

   c. **x = 2**
      y = 5
      mystery(x, y) ____**17**_____

   d. **x = 2**
      y = 5
      mystery(y, x) ___**11**_____

3. Consider the ***secret*** and ***mystery*** functions defined above. What are the values of:

   a. **x = 2**
      y = 5
      secret(3) + mystery(x, y) _____**33**_____

   b. **secret(mystery(2, 1))** _____**36**_____

   c. **x = 2**
      y = 1
      mystery(secret(x), secret(y)) _____**21**_____

4. Consider the code snippet to the right. Explain briefly why there is a ***red X*** beside ***Line 3***.

```
1  def main():
2      foo(10)
3      print(n)
4
5
6  def foo(m):
7      n = m + 20
8      return n
```

The name (i.e., variable) *n* in *main* is undefined. That is, it has no value when Line 3 executes.

The presence of a name (variable) *n* in *foo* is irrelevant. Names defined inside one function are *independent* of names defined in other functions.

5. Consider the code snippet to the right. Explain briefly why there is a ***red X*** beside ***Line 7***.

```
1  def main():
2      a = 3
3      foo()
4
5
6  def foo():
7      b = a + 7
8      print(7)
```

The name (variable) *a* in *foo* is undefined. That is, it has no value when Line 7 executes.

The presence of a variable *a* in *main* is irrelevant. Names defined inside one function are *independent* of names defined in other functions.

6. Consider the code snippets defined below. They are contrived examples with poor style but will run without errors. For each, what does it print when *main* runs?

(Each is an independent problem. Pay close attention to the order in which the statements are executed.)

```
def main():
    x = 5
    foo(x)
    print(x)

def foo(x):
    print(x)
    return x ** 3
```

Prints: 5

5

```
def main():
    x = 5
    y = foo(x)
    print(y)

def foo(x):
    x = 10
    print(x)
    return x ** 3
```

10

1000

```
def main():
    x = 5
    x = foo(x)
    print(x)

def foo(x):
    print(x)
    return x ** 3
```

5

125

```
def main():
    a = 2
    b = 3
    c = 44
    d = 55

    foo1(a, b)
    #### Location 1

    foo2(c, d)
    #### Location 2

    r = 25
    s = 35
    r = foo3(r, s)
    #### Location 3


def foo1(a, b):
    #### Location 4
    a = 88
    b = 99
    #### Location 5


def foo2(x, y):
    #### Location 6
    a = 400
    b = 500
    y = 600
    #### Location 7


def foo3(s, r):
    #### Location 8
    return r + s


main()
#### Location 9
```

7. **Consider** the code to the left. It is a contrived example with poor style but will run without errors. In this problem, you will trace the execution of the code. ***As each location is encountered during the run:***

1. ***CIRCLE* each variable** that is ***defined*** at that location.
2. ***WRITE* the *VALUE*** of each variable that you ***circled*** directly ***BELOW*** the circle.

For example, the run defines the functions and then calls ***main***, as usual. The first of the nine locations **to be encountered** is **Location 4**. At Location 4, the only variables defined are ***a*** and ***b***, with values ***2*** and ***3*** at that point of the program's run. So, on the row for Location 4, I have circled ***a*** and ***b*** and written their values at Location 4 directly below them.

Note that you fill out the table **in the order that the locations are encountered, *NOT from top to bottom. ASK FOR HELP IF YOU DO NOT UNDERSTAND WHAT THIS PROBLEM ASKS YOU TO DO.***

	a	b	c	d	r	s	x	y
Location 1	2	3	44	55				
Location 2	2	3	44	55				
Location 3	2	3	44	55	60	35		
Location 4	2	3	c	d	r	s	x	y
Location 5	88	99	c	d	r	s	x	y
Location 6	a	b	c	d	r	s	44	55
Location 7	400	500	c	d	r	s	44	600
Location 8	a	b	c	d	35	25	x	y
Location 9	a	b	c	d	r	s	x	y

None of the above are defined at Location 9

8. What is the value of each of the following expressions?

```
17 // 4        =  4
```
Hint: This is a WHOLE number (i.e., integer).

```
17 % 4         =  1
```
Hint: This is the REMAINDER from `17 // 4`.

```
3 / 4          =  0.75
```

```
7 % 2          =  1
```
Aside: If `x % 2 == 0`, then `x` is EVEN.
If `x % 2 == 1`, then `x` is ODD.

```
7 ** 2         =  49
```

```
'fun' + 'ny'        =  'funny'
```

```
'hot' * 5           =  'hothothothothot'
```

```
'fun' + 3
```
This is not a legal expression. It breaks when it runs.

```
10 ^ 2
```
This does NOT evaluate to 100. The `^` (caret) symbol does NOT mean exponentiation (raising to a power) in Python. It has an entirely different meaning that is not important to our current work.[1] We won't ask you what `^` means on the test, but it is important to know that `^` is NOT exponentiation.

9. List ***two*** reasons why functions are useful and important.

Reason 1: **They help *organize* the code, which makes it easier to get the code correct when writing it and to maintain that code's correctness as changes are made later in the lifetime of the software.**

Reason 2: **They *allow for code re-use*, by allowing the function to be called multiple times with different values for the parameters.**

[1] But just in case you are curious, here is what it does mean: bitwise exclusive-OR. Since `10` is `0110` in binary and `2` is `0010` in binary and `0110` bitwise exclusive-OR'ed with `0010` is `0100`, which is `8` in decimal, `10 ^ 2` evaluates to `8`.

10. Consider the two functions shown to the right.

```
def foo1(a):
    z = 100
    if a > 10:
        z = 98
    else:
        z = 99
    return z
```

```
def foo2(a):
    z = 100
    if a > 10:
        z = 98
    if a <= 10:
        z = 99
    return z
```

a. Are ***foo1*** and ***foo2*** logically equivalent? That is, is it the case that, for all integers ***x***,

foo1(x) == foo2(x)

(***Yes***) ***No*** (circle your choice)

If they are NOT logically equivalent, specify an ***x*** for which it is NOT true that **foo1(x) == foo2(x)** _____

b. Which runs faster, (***foo1***) or ***foo2*** ? (circle your choice)

Explain.

foo1 does one comparison (of ***a*** versus **10**) and one assignment.

Function ***foo2*** also does only one assignment, but it always does TWO comparisons.

11. Consider the two functions shown to the right.

```
def foo1(a):
    z = 100
    if a > 10:
        z = 98
    else:
        z = 99
    return z
```

```
def foo2(a):
    z = 100
    if a > 10:
        z = 98
    if a < 10:
        z = 99
    return z
```

a. Are ***foo1*** and ***foo2*** logically equivalent? That is, is it the case that, for all integers ***x***,

foo1(x) == foo2(x)?

Yes (***No***) (circle your choice)

If they are NOT logically equivalent, specify an ***x*** for which it is NOT true that **foo1(x) == foo2(x)** _____**10**_____

foo1(10) returns **99**, while **foo2(10)** returns **100**.

b. Which runs faster, (***foo1***) or ***foo2*** ? (circle your choice)

Explain. Same reasoning as in the previous problem.

12. Assume that you have a variable (name) **x** that is a positive integer. Write a snippet of code that prints **30** if **x** is odd.

```
if (x % 2) == 1:
    print(30)
```

13. Continuing the previous problem, write a snippet of code that prints **30** if **x** is odd and prints **22** if **x** is even.

The solution in this box is CORRECT.

```
if (x % 2) == 1:
    print(30)
else:
    print(22)
```

The solution in this box is WRONG. It achieves the same effect as that of the correct answer (to the left), but it does so in a way that is less efficient and an abuse of conditionals.

```
if (x % 2) == 1:
    print(30)
if (x % 2) == 0:
    print(22)
```

14. Continuing the previous problem, write a snippet of code that prints **30** if **x** is odd and prints **'hello'** if **x** is greater than or equal to **100**. Note that when **x** is (for example) **151** this snippet would print both **30** and **'hello'**.

```
if (x % 2) == 1:
    print(30)
if x >= 100:
    print('hello')
```

In the previous problem, the conditions (odd or even) were exclusive of each other, so ELSE is called for. In this problem, both conditions may fire, so we need separate IFs for them.

15. Write a snippet of code that would construct an **`rg.Point`** object at **`(300, 444)`** and give the **`rg.Point`** object a name. You can choose any reasonable name that you like.

```
point = rg.Point(300, 444)
```

Names like `p` or `p1` or `point1` all are reasonable. Names like `A` or `b` are NOT reasonable. Names like `x` or `circle` are TERRIBLE choices.

16. Assume that you have a variable (name) **`p2`** that is an **`rg.Point`** object. Write a snippet of code that would triple the y-coordinate of **`p2`**.

```
p2.y = p2.y * 3
```

17. Assume that you have a variable (name) **`p2`** that is an **`rg.Point`** object and a variable named **`x`** that is a floating point number. Write a snippet of code that would increase the x-coordinate of **`p2`** by **`x`**. (Note: using **`x`** as a variable name here is a poor choice, but solve the problem as written anyhow.)

```
p2.x = p2.x + x
```

18. Assume that you have a variable (name) **`circle9`** that is an **`rg.Circle`** object. Write a snippet of code that would make the radius of **`circle9`** decrease by **`30`**.

```
circle9.radius = circle9.radius - 30
```

19. Continuing the previous problem, write a snippet of code that would make the radius of **`circle9`** decrease by **`30`** unless doing so would make that radius less than **`1`**, in which case the code should make the radius be **`1`**.

```
if circle9.radius >= 31:
    circle9.radius = circle9.radius - 30
else:
    circle9.radius = 1
```

20. Assume that you have a variable (name) **rectangle** that is an **rg.Rectangle** object. Write a snippet of code that would use its **get_upper_left_corner** method to print the rectangle's upper-left corner.

```
print(rectangle.get_upper_left_corner())
```

21. Assume that you have a name (variable) **fido** that refers to a **Dog** object. Assume further that **Dog** objects have a **bark** method that takes as an argument the number of times to bark. Write a statement that would make **fido** bark 5 times.

```
fido.bark(5)
```

22. Write a snippet of code that would print the following numbers (but each on its own line):

5 8 11 14 17 20 23 26 29 32 35

Note: No credit for just 11 ***print*** statements, that is, for `print(5) print(8) print(11) ... print(35).`

```
for k in range(11):
    print((3 * k) + 5)
```

23. Assume that you have a name (variable) **win3** that is an **rg.RoseWindow** object, and also names (variables) **r** and **s** that are positive, even integers, with **s > r**.

Write a snippet of code that would construct **rg.Circle** objects, with each centered at **(300, 200)**, but with radii that are:

r r + 2 r + 4 r + 6 ... s

For example, if **r** is **8** and **s** is **14**, your code must construct **4** **rg.Circle** objects, with radii **8**, **10**, **12**, and **14**, respectively.

Your code must also attach each **rg.Circle** that it constructs to **win3**.

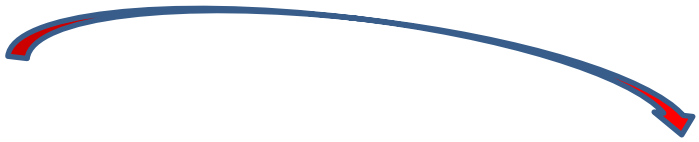
Write code for the generic case for **r** and **s**. That is, use the names (variables) **r** and **s** in your code. You should use a **range** statement in your solution. You may NOT use the multiple-argument form of **range** in this problem. That is, the **range** expression in your solution must have only a ***single*** argument.

```
center = rg.Point(300, 300)
for k in range( ((s – r) // 2) + 1):
    circle = rg.Circle(center, r + (2 * k))
    circle.attach_to(win3)
```

24. For each of the 3 code snippets below, what does it print?
(Write each answer directly below its code snippet.)

Hint: Solve problems like this by make a ***table with the variables, showing the places where their values change***. Here is an example of a table appropriate for the **3rd (rightmost)** problem. It was made by tracing the code by hand, starting from line 1 of the table (which came from the statement **`b = 0`**) and continuing downward from there as the by-hand trace continues.

k	b
	0
0	
1	1
2	
3	
4	2



```
for j in range(4):
    print((j * 2) + 1)
```

```
1
3
5
7
```

```
a = 10
for k in range(8):
    if k % 2 == 0:
        a = a + k
        print(k, a)
print(a)
```

```
0  10
2  12
4  16
6  22
22
```

```
b = 0
for k in range(5):
    if (k + 4) % 3 == 2:
        b = b + 1
        print('b is:', b)
    print(k, b)
print(b)
```

```
0  0
b  is: 1
1  1
2  1
3  1
b  is: 2
4  2
2
```

25. What gets printed when ***main*** is called in the program shown to the right?

    Write the output in the provided box.

```
def main():
    a = 2
    b = 3

    foo1()
    print(a, b)

    foo2(a, b)
    print(a, b)

    foo3(a, b)
    print(a, b)

def foo1():
    a = 88
    b = 99

def foo2(a, b):
    a = 400
    b = 500

def foo3(x, y):
    x = 44
    y = 55
```

Output

2 3

2 3

2 3

26. Consider the snippet of code shown to the right. Circle whichever of the following is correct, assuming that **foo_11** and **foo_12** were written as they appear in the snippet intentionally.

```
43 def blah():
44     x = foo_11(3, 7)
45     print(x)
46
47     y = foo_12(3, 7)
48     print(y)
49
50
51 def foo_11(a, b):
52     print(a * b)
53
54
55 def foo_12(a, b):
56     return (a * b)
```

- Line 44 is sensible but Line 47 is not.
- Line 47 is sensible but Line 44 is not.
- Both Line 44 and Line 47 are sensible.
- Neither Line 44 nor Line 47 is sensible.

The foo_11 function does not ***return*** a value, so it is silly for line 44 to capture the returned value (which is necessarily None).

27. Consider the code snippet below. It is a contrived example with poor style, but it will run without errors. What does it print when it runs? (Consider using ***post-it notes*** as was done in one of the videos you watched on function calls.)

```
def main():
    one()
    two()
    three()


def one():
    print('One!')
    return 1 + two()


def two():
    print('Two!')
    return 1
    print('Done!')


def three():
    print('Three!', two(), one())


main()
```

Write your answer in the box to the right of the code.

Output:

```
One!
Two!
Two!
Two!
One!
Two!
Three! 1 2
```

Here is an explanation of what happens in the above:

1. The definitions are all read, then ***main*** is called at the bottom of the code.
2. ***main*** calls ***one***.
3. ***one*** prints `One!` and then calls ***two***.
4. ***two*** prints `Two!` and then returns **1**. (The `print('Done!')` statement is never reached, since a ***return*** statement really ***leaves the function***, returning to its caller.)
5. Control returns to ***one***, where the returned **1** is added to the **1** in `return 1 + two()`, yielding **2**, so **2** is returned from the ***one*** function, back to ***main***.
6. ***main*** ignores the returned value from ***one*** and calls ***two***.
7. ***two*** prints `Two!` and returns **1**. (Again, the `print('Done')` is never reached.)
8. ***main*** ignores the returned value from ***two*** and calls ***three***.
9. ***three*** calls ***two***. two prints `Two!` and returns **1** back to ***three***.
10. ***three*** calls ***one***. ***one*** prints `One!`, calls ***two*** which prints `Two!` and returns **1** to ***one***. Then ***one*** adds the returned **1** to **1** and returns **2** to ***three***.
11. ***three*** has now computed the values of the 3 arguments to its ***print*** statement and prints them: `Three! 1 2`.

28. True or False: As a ***user*** of a function (that is, as someone who will ***call*** the function), you *don't need to know how the function is* ***implemented***; you just need to know the ***specification*** of the function. **True** **False** (circle your choice)

29. Does the function definition shown to the right meet its specification? If not, why not?

```
def get_number(x):
    """
    Returns x squared plus x cubed, for the given x.
    For example, if x is 5, returns (5 ** 2) + (5 ** 3),
    which is 150.
    """
    answer = (x ** 2) + (x ** 3)
    print(answer)
```

No – it does NOT meet its specification.

Its specification says to RETURN the answer, *not* PRINT it.

30. Does the function definition shown to the right meet its specification? If not, why not?

```
def get_number(x):
    """
    Returns x squared plus x cubed, for the given x.
    For example, if x is 5, returns (5 ** 2) + (5 ** 3),
    which is 150.
    """
    answer = (x ** 2) + (x ** 3)
    print(answer)
    return answer
```

No – it does NOT meet its specification.

Its specification does NOT say to PRINT anything, so doing so violates the specification. Printing is a SIDE-EFFECT – a function must have no side-effects beyond what the specification specifies.

31. Does the function definition shown to the right meet its specification? If not, why not?

```
def test_get_number(x):
    """ Tests the   get_number   function. """
    answer1 = get_number(5)
    answer2 = get_number(1)
    answer3 = get_number(2)
```

No – it does NOT meet its specification.

Its specification says to TEST the function. The code CALLS the function (good!), but does nothing with the returned value. As such, it does not TEST whether the returned value is correct.

(This explanation continues on the next page)

Testing the returned value requires either printing it (so that the human user can check whether or not the returned value is correct) or otherwise checking the returned value (e.g., by comparing the returned value to the correct answer and printing an appropriate message as a result).

Furthermore, we will also require that you print the EXPECTED value to be returned, so that you can demonstrate that you really did have something to check the answer against.

IMPORTANT: Finally, if you simply *RUN* your function and *THEN* provide the "expected value" as the value that your function produces, that is NOT A TEST and you will get NO CREDIT for doing so.

You MUST have tests that are either *GIVEN* to you by us (possibly as an example in the specification, possibly in the testing code) or *COMPUTED BY HAND* by you.

32. Consider a function whose name is ***print_string*** that takes two arguments as in this example:

```
print_string('Robots rule!', 4)
```

The function should print the given string the given number of times. So, the above function call should produce this output:

```
Robots rule!
Robots rule!
Robots rule!
Robots rule!
```

Write (in the space to the right) a complete implementation, ***including the header (def) line,*** of the above ***print_string*** function.

Answer:

```
def print_string(s, n):
    for k in range(n):
        print(s)
```

A better answer might choose better names for `s` and `n` (e.g. `string_to_print` and `times_to_print`), but the answer above is acceptable in this context.

33. Assume that you have a function ***is_perfect*** whose specification is as shown to the right.

```
def is_perfect(m):
    """
    What comes in: an integer m.
    What goes out:  Returns True if the argument m
      is "perfect".  Returns False otherwise.
    """
```

Consider a function whose name is ***add_them*** that takes two integer arguments ***m*** and ***n*** (with $m \leq n$) and returns the sum of the integers from ***m*** to ***n***, inclusive, that are NOT perfect. Write (in the space below) a complete implementation, ***including the header (def) line***, of the ***add_them*** function. Note that you do not need to know what makes a number "perfect" to solve this problem.

Answer:

```
def add_them(m, n):
    total = 0
    for k in range(n - m + 1):
        if not is_perfect(k + m):
            total = total + (k + m)
    return total
```

The IF statement in the above could also be written as:

```
    if is_perfect(k + m) != True:
```

or as:

```
    if is_perfect(k + m) == False:
```