Conditionals & Recursions

Chapter 4-5

Bonus joke: what do you call a cross between an elephant and a rhinoceros?

`elif` I know.

Today's Outline

- Modulus operator (%)
- Boolean expressions & Logical operators
- Conditional execution using "if" and "while" statements
- Recursion and Infinite recursion

- All of the above is about writing logical steps that Python can understand to perform a task.

Modulus Operator (%)

- The modulus operator, %, divides two numbers and returns the remainder.

```
>>> quotient = 7 / 3
>>> print (quotient)
2
>>> remainder = 7 % 3
>>> print (remainder)
1
```

Boolean Expressions

- What’s a Boolean expression?
  - In Python `True` and `False` (notice the caps!) are boolean values: an expression that is either `True` or `False`.
- Relational operators are can create a boolean expression.

Boolean Expressions

- With **Relational Operators** I can know a **condition**
- Relational operators are can create a boolean expressions:

```
x == y    # x is equal to y.
x != y    # x is not equal to y.
x > y     # x is greater than y.
x < y     # x is is less than y.
x >= y    # x is greater than or equal to y.
x <= y    # x is less than or equal to y.
```

REMEMBER!

= Is an assignment operator: e.g. **number = 7**

== Relational operator: **2 + 2 == 4**

Example Use Case

- Sorting values to find certain ones without needing to know the exact value: e.g. plots of land greater than a certain area for planning purposes.
- Sorting through humanitarian programming budgets so see which ones have funding left in them.
- Anything where “true” and “false” help you quickly sort information.

Modulus Operator (%)

- Use Cases: see if one value is divisible by another.

```
x % y == 0
```

- Evaluates true if and only if x is an exact multiple of y.
- Use Cases: test if x is an odd or even number.

```
6 % 3 == 0 # What's the output
7 % 3 == 0 # What's the output
```

- Remember… 0 is only telling you if there is a remainder or not… There is "0 remainder."

Logical Operators

- …can also be used to build *Boolean expressions*.
- Three logical operators are `and`, `or`, and `not`.
- True or false?

```
5 > 0 **and** 5 < 10
# True – 5 is greater than 0 and less than 10.
6%2 == 0 or 6%3 == 0
# True if either or both is true: if the number is
divisible by 2 or 3.
```

Example Use Case

- What about more complicated GIS where classes (e.g. urban, agricultural, forest) or scales (less suitable to more suitable) are used?
- How would I build a Boolean expression to find a plot of land greater in area than 155 units that has a suitability index of 0.9 or greater?
  - `x == area > 154 and suitability > 0.8`

Logical Operators

- Use `not` to negate a *Boolean expression*?
- True or false?

```
not (x > y)
# Only true if (x > y)is false e.g. 5 > 10
```

Conditional Execution

- We often need to check conditions and change the behavior of the program accordingly
- `if` statements have the same structure as function definitions: header followed by indented body.

```
#This is an if statement
water_per_person = 20 #in liters, daily
if water_per_person > 19: #20 is sphere standard
   print (water_per_person, ": post emergency levels.")

water_per_person = 13 #in liters, daily
if water_per_person < 15:
   print (water_per_person, ": EMERGENCY LEVEL.")
```

Conditional Exec. (if)

- There is no limit to the number of `if` statements in the body: but there must be at least one.
- Sometimes you may want a placeholder statement in the body for something you will do later): use the `pass` statement, which does nothing. E.g.

```
if x < 0:
    pass #Placeholder for something.
```

Alternative Execution

- `else` statements are used when there are two possibilities and the condition determines which runs.
- Write this in script window, save, press F5 to run in shell.

```
x = 16
if x % 2 == 0:
     print("x is even.")
else:
     print("x is odd.")
```

What's the result? Change one thing in the code to make the opposite.

Alternative Execution

- Define a function to do what we've just done. The function should take a value and tell you if it is even or odd.

```
def printParity(x):
     if x % 2 == 0:
          print (x, "is an even number.")
     else:
          print (x, "is an odd number.")
```

Alternative Execution

- Since these examples are either true or false **exactly one condition will run**: these alternatives are called branches.

```
x = 16
if x % 2 == 0:                      # This is branch 1.
     print (str(x) + " is even.")
else:                               # This is branch 2.
     print (str(x) + " is odd.")
```

Chained Conditionals

- … are used when there are several possibilities.
- `else` and `elif` statements allow to deal with this.
- `elif` is an abbreviation of "else if."
- No limit to number of `elif` statements. You can use the `else` statement in combination, but it must come at the end.

Chained Conditionals

- Define a function called `ifequal` that takes two values and: prints out if if one is greater than the other, less than the other, if if they are equal.
- Use `if`, `elif` and `else` in a script window,

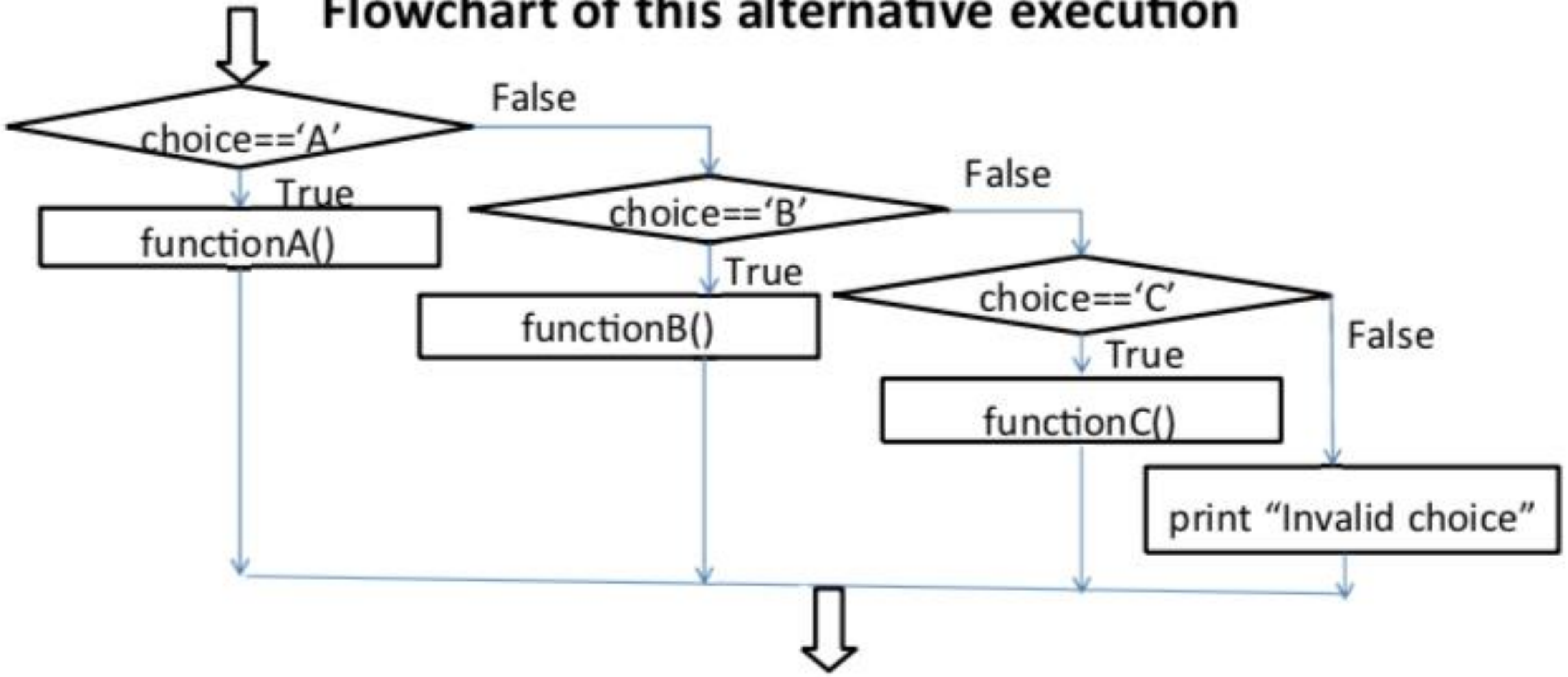
```
def ifequal(x,y):
    if x < y:
          print(x,"is less than", y)
    elif x > y:
         print(x,"is greater than", y)
    else:
         print(x,"is equal to", y)
```

Chained Conditionals

- Each condition is checked in order.
- If a condition is true, the corresponding branch is run and the statement ends!
- This happens even if more than one condition is true only the first runs!!!!

```
if choice == “A”:
   functionA()
elif choice == “B”:
   functionB()
elif choice == “C”:
   functionC()
else:
   print "Invalid choice."
```

Flowchart of this alternative execution



Nested Conditionals

- Avoid them if you can. Try and clean up your code…

```
if x == y:
    print x, "and", y, "are equal"
else:
    if x < y:
        print x, "is less than", y
    else:
        print x, "is greater than", y
```

- How can we clean this up?

- This is messy and example of “crisco” in the code…

```
if x == y:
    print x, "and", y, "are equal"
else:
    if x < y:
        print x, "is less than", y
    else:
        print x, "is greater than", y
```

- This is better!

```
if x == y:
    print x, "and", y, "are equal"
elif x < y:
    print x, "is less than", y
else:
    print x, "is greater than", y
```

Recursion

- Functions can call themselves to recur!
- Run in script window.

```
def countdown(n):
    if n == 0:              #This is the base case
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

Infinite Recursion

- If a recursion never reaches a "base case", it goes on making recursive calls forever, and the program never terminates.
- The base case is the case where the condition can be met without further recursion.

```
def countdown(n):
    if n == 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n) # Replaced "n-1" with "n".
```

The Return Statement

- The `return` statement will end a function and “return” or give back any value to whatever called it in the first place.

```
def cube(num):
    num*num*num

print(cube(3))
```

- The `print` statement doesn’t have anything to print because the function didn’t “return” the value to be printed. Try `return num*num*num`

The Return Statement

- Sometimes the `return` statement will end a function.

```
# Input: This function takes two integers as an argument
# Output: It prints "s" ntimes.

def print_n(s, n):
        if n <= 0:
                return # This stops the function!
        print(s)
        print_n(s, n-1)
```

- In this case it doesn't `return` anything.
- The `print` statement doesn't "return" or end, it just prints.

Exercise

Write a recursive Python function that returns the factorial of a positive integer.

1. What do we mean by this? Example using “4”
   - 4 + 3 + 2 + 1 = 10
2. Hint:

```
def sumInt(num):
    if num == 0:
        # write one line here
    else:
        # write one line here
```

Exercise

Answer:

```
def sumInt(num):
    if num == 0:
        return 0
    else:
        return num + sumInt(num - 1)

print sumInt(4)
```

Summary

- "Modulus Operator" returns a remainder: can be used to with a …
- Boolean expression, which tells if something is true or false and can be used in a…
- Conditional **if** statement and **while** statements: if a given condition is true they work
- Recursion does magical work but be careful using it.
- Reading for next week is on Moodle:

Keyboard Input

- `Input` allows us to take information from a user.
- It returns what the user typed as a string!
  - Why is this valuable?
- `\n` is a special character that calls for a new line. It's handy to use after input so that the user input appears on a different line. E.g

```
name = input('What is your name?\n')
```

Exercise 2

Make your .py program interactive by using the "input" function.

1. Syntax: `input()`
2. Using the previous code, create a line that makes r equal to input from the user.
   ```
   r = input()
   ```
3. Use type conversion to make sure integers are converted to floats, in case the users enters a whole number.

Exercise 2

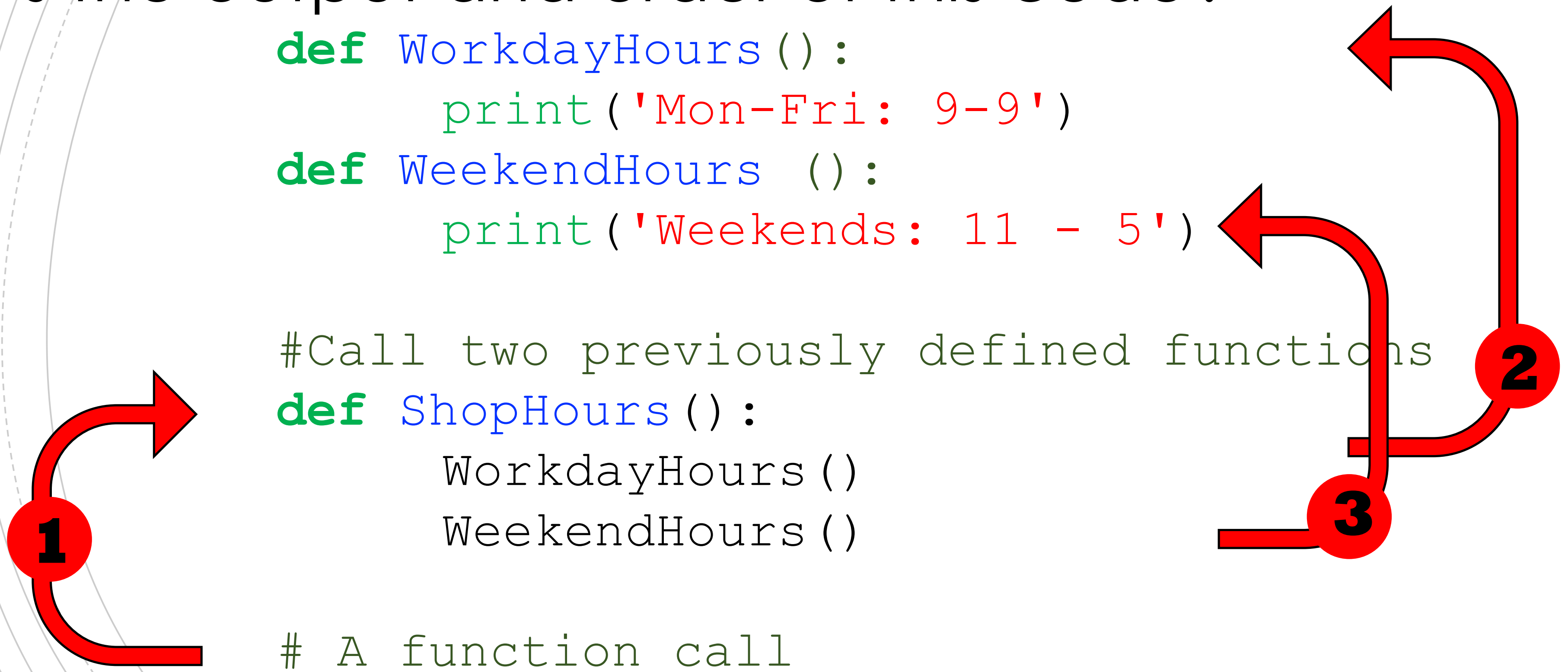
Example answer:

```
import math
def Circle_area (r):
      """
      Function defines area of a circle.
      Input: script prompts users to enter value for r
      """
      r = float(input ("Input the radius of the circle : "))
      a = r**2 * math.pi
      return a

 print(Circle_area(r))
```

Functions: Execution Flow

- What's the output and order of this code?



```
def WorkdayHours():
    print('Mon-Fri: 9-9')
def WeekendHours ():
    print('Weekends: 11 - 5')

#Call two previously defined functions
def ShopHours():
    WorkdayHours()
    WeekendHours()

# A function call
ShopHours()
```