Python #7
Making choices

Location: https://people.sc.fsu.edu/~jburkardt/classes/python_2022/python07/python07.pdf

Freely adapted from the Python lessons at https://software-carpentry.org/

Making choices

- *We may choose different actions based on some logical condition;*
- *The* `if`*,* `elif`*, and* `else` *statements define these choices;*
- *Conditions can use numeric comparisons like* `<`*,* `¡=`*,* `==`*,* `!=` *;*
- *Conditions can be combined using* `and`*,* `or`*,* `not`*;*
- *Conditions are logical expressions with the value* `True` *or* `False`*;*
- *A complicated decision may require the use of nesting.*

1 When actions depend on conditions

We could describe most of our programs so far as a sequence of statements that say `do this, then this, then this!`. However, it is sometimes true that, before we carry out certain actions, we need to check that some condition is true.

For instance, we are familiar with the mathematical rules that you can compute the inverse of a number `x`, if `x` is not zero. Similarly, you can compute the square root of a number and expect a real number result ... if the number is not negative.

As a typical coding instance, suppose we have numbers `a` and `b`, and we want to print them in order. What we do next depends on the relation between the two values:

```
  if a < b then
    print ( a, b )
  otherwise
    print ( b, a )
```

We will need to be able to express choices like this in our Python computations. Our tools will be `if`, `elif`, `else`, the numeric comparison operators, and logical operators `and`, `or`, `not`.

2 Buying a bus ticket

The basic bus fare in Pittsburgh depends on your age. Children 5 and under are free, ages 6 through 11 pay $1.35 and other riders pay $2.50. Suppose a computer program checks the age of a rider getting on the bus, and has to compute the fare. In Python, this would look something like this:

```
if ( age <= 5 ):
  fare = 0.0
elif ( age <= 11 ):
  fare = 1.35
else:
  fare = 2.50
```

This combination is an example of an `if`-structure, which allows Python to choose one of a set of actions based on specified conditions.

Check this code by putting it inside the following loop:

```
  for age in [ 65, 3, 9 ]:
    ...
    print ( 'Age ', age, ' pays ', fare )
```

Notice that this structure contains three possible fare computations. But only one of them will be carried out. We apply the first fare if the first condition is true. If the first condition is not true, then we apply the second fare if the second condition is true. Otherwise, we apply the third fare.

You should observe a few details in this pattern: Whereas a human would say "else if", the Python word is `elif`. The statements `if` and `elif` are each followed by a condition, which is enclosed in optional parentheses. The `if`, `elif`, and `else` statements all terminate with a colon.

You might think that the elif statement should read

```
  elif ( 5 < age <= 11 ):      # Warning!  Not legal Python!
```

because that's how we described the half price fare. While this looks natural mathematically, it is not a legal Python statement. Python can't compare one number to both a lower and upper limit at the same time. To get the effect you want, you need to use an `and` statement:

```
  elif ( 5 < age and age < 11 ):
```

This more elaborate statement is correct, but is not actually necessary, since if the age is less than or equal to 5, the previous `if` statement would have become active.

Our triple-decker bus fare example involves all three possible conditional statements. However, in practice, we might just have a single `if` statement, or one `if` statement followed by several `elif` statements that each pick up a special case. An `else` statement is not required, but if used must show up as the last conditional statement in an `if`-structure.

3 The Collatz conjecture

Start with any (positive) integer n. If n is even, divide it by 2, but if n is odd, multiply it by 3 and add 1. If the result is 1, stop. Otherwise repeat the process, if necessary, forever.

The Collatz conjecture: No matter what value of n you start from, you will reach 1 in a finite number of steps.

The Collatz conjecture, made in 1937, remains unproved to this day, and is the subject of research by such prominent mathematicians as Terence Tao.

Let's look at a typical computation that starts with `n` = 37:

```
 1   37
 2  112 <- 3*37 + 1
 3   56 <- 112/ 2
 4   28 <- 56 / 2
 5   14 <- 28 / 2
 6    7 <- 14 / 2
 7   22 <- 3*7+1
 8   11 <- 22 / 2
 9   34 <- 3*11 + 1
10   17
11   52
12   26
13   13
14   40
15   20
16   10
17    5
18   16
19    8
20    4
21    2
22    1
```

How would we express such a computation? Clearly, we need to start by initializing `n`. Then we need to define a loop that is carried out as long as `n` is not 1. Inside the loop, we have to choose which operation to carry out, depending on whether the current value of `n` is odd or even. Here's one way to do all this:

```
n = 37
i = 1
while ( n != 1 ):
  if ( n % 2 == 0 ):
    n = n // 2    # Use // to guarantee that Python sees n is always an integer
  else:
    n = 3 * n + 1
  i = i + 1
print ( 'Went through', i, 'values to reach 1' )
```

You might not be familiar with a few things in this program. The `while(condition)` statement is similar to a `for` statement; it begins a loop. But instead of relying on a loop index or counter, the `while()` statement checks a certain condition before carrying out the actions that follow.

For one thing, this means that if we initialize `n` to 1, the loop will not be carried out at all. (You can check what happens, and see if you think this is the right behavior.)

For another thing, each time we execute the statements inside the loop, we are changing the value of `n`. So after the loop has executed the commands once, the `while` statement has to check the value of `n` before deciding whether to let the commands be executed again.

Think about one danger when using a `while` statement for our program. Suppose that there is some number `n` which never settles down to 1. (This would mean that the Collatz conjecture is false. It would also mean that you are now famous.) In that case, the loop would run forever. This would raise another mathematical question. Does the infinite sequence that you generate eventually start to repeat, in which case it must begin to cycle through a fixed finite set of numbers, or does it go on forever producing new numbers, although never reaching 1? This question would give your children something to work on!

4 Nested conditionals

Suppose we have numbers `a`, `b`, and `c`, and we want to print them in ascending order. So if `a` = 10, `b` = 3, `c` = 6, we want to determine that `b <= c <= a` and issue the command `print ( b, c, a )`. Of course, there are six different possibilities for the ordering of three numbers. How could we do this using our conditional statements?

One approach is a two step process. First we determine whether `a` is less than `b` or not. Once we know the ordering of two objects, our second step determines whether to place `c` in front, middle, or behind them.

A natural way to do this involves a nested set of conditional statements.

```
if ( a < b ):

  if ( c < a ):
    print ( c, a, b )
  elif ( c < b ):
    print ( a, c, b )
  else:
    print ( a, b, c )

else:

  if ( c < b ):
    print ( c, b, c )
  elif ( c < a ):
    print ( b, c, a )
  else:
    print ( b, a, c )
```

The proper indenting is very important! Test this code using as input the six ways of ordering the values 1, 2 and 3:

```
for a, b, c in [ [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1] ]:
```

The code has one mistake in it, and so one line of your output will be wrong. Can you find it?

5 Logical values

A condition like `a < b` is a logical expression, and hence has the value "true" or "false". Python has a special class of logical data, and so we can create a variable name, if we wish, and store the value of a condition that way. This means that as part of our Collatz computation, we could have created a logical variable called `even` and used it as follows:

```
even = ( n % 2 == 0 )
if ( even ):
```

```
    n = n // 2
  else
    ...
```

Like any Python variable, we could print `even`, in which case we would see one of the values `True` or `False`. (The initial capital letter here is important).

One "crazy" use of the logical constant `True` is to create a `while` loop that theoretically runs forever. It looks something like this:

```
while ( True ):
  commands to run "forever"
```

In fact, inside such a loop, we can escape at any time by issuing the `break` command.

Why would we want to create such a loop? Consider the Euclidean algorithm, which finds the greatest common divisor (GCD) of numbers p and q.

1. If $p < q$, exchange p and q.
2. Divide p by q and get the remainder, r. If $r = 0$, report q as the GCD.
3. Replace p by q and replace q by r. Return to the previous step.

Here's a sample calculation for the GCD of ... and ...:

```
    P     Q     R
34578  6215  3503
 6215  3503  2712
 3503  2712   791
 2712   791   339
  791   339   113
  339   113     0
```

And we can conclude that the greatest common divisor of 34,578 and 6,215 is 113.

To write this as a Python code, we need to use a loop, and it can't be a `for` loop. We might like to use a while loop, but we need to check the value of `r` in the middle of the loop, not at the beginning or end. And as soon as we detect that `r` is zero, we want to break from the loop. So here is how a logical value of `True` allows us to loop forever, or more precisely, as long as we need to.

```
print ( 'seeking greatest common divisor of', p, 'and', q )
while ( True ):

  r = ( p % q )    # The percent sign computes the remainder when p is divided by q

  if ( r == 0 ):
    break

  p = q
  q = r

print ( 'gcd is', q )
```

This implementation assumes that $q \leq p$. Can you insert some lines of code so that the calculation will work even if $p < q$?

6 How to steal a bicycle

Suppose that a bicycle lock has a 3 digit lock, whose digits can only go from 0 to 7. Thus, 304 is a possible combination, but 394 is not. Now suppose that we want to unlock the bicycle, and don't know the code. We

are going to try all possible legal combinations, in order, but we are lazy, so every day we just do the next one.

Can we write a code which is given `h, t, u`, the digits between 0 and 7 of yesterday's combination, and produces the combination we should try today? If yesterday we tried 304, then today we want to try 305. Similarly, yesterday's 309 becomes today's 310. Of course, if yesterday we tried the last possible combination, then today we just want to give up.

So assume we have values for `h, t, u` from yesterday. How can we determine the next combination? Obviously, we want to increase `u` by 1. If the resulting `u` is no greater than 7, we're done. But if the new value of `u` is 8, we need to reset `u` to 0, and "carry" a 1 to the tens column, that is, we need to update the value of `t`. A similar process goes on with `t`, and that might also involve a carry that will modify `h`. But if even `h` would need a carry operation, what do we do?

To test your program, try each of the following starting points and report the next combination that your code comes up with:

```
h, t, u = 4, 0, 4
h, t, u = 5, 1, 7
h, t, u = 6, 7, 7
h, t, u = 7, 7, 7
```

You could have initialized the digits with three separate statements, but Python allows you to assign separate values to multiple variables using commas, and this makes the presentation a little neater for this example.

In order to do all four tests efficiently, recall how we can create a single `for` statement that will loop over each set of initial combination digits.