Making decisions with if() statements
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm
Room A202 Langley Hall

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2025/if/if.pdf

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 )
  end

# This is NOT quite legal Python code!
```

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 Logical constants, variables, and operators

Python reserves the words `True` and `False` to represent the values assigned to variables which are called boolean or logical.

A variable can be set directly to one of these values, but more often, it is defined as the truth value of a given comparison operator, as in

```
  x_is_odd = ( x % 2 == 1 )
  x_is_smaller = ( x < y )
  x_is_not_y = not y
  unequal = ( x != y )
```

We can use `and` and `or` to make more complicated conditions. The values A, B, and C make a triangle if

```
  is_a_triangle = ( A + B <= C and B + C <= A and C + A <= B )
```

3 Leap years

It's usually true that a year y is a leap year, with an extra day, if y is divisible by 4. So we might set the number of days in a year to 365, but then check if we need to make an adjustment:

```
days = 365

if ( y % 4 == 0 ):
  days = 366
  print ( 'It is a leap year!' )

# More code follows, which is NOT indented
```

This is the simplest version of the `if()` statement. It consists of an initial statement of the form

```
  if ( condition ) :
```

where `( condition )` is some logical quantity, usually a numeric comparison involving one of the operators `==, !=, <, <=, >, =>`. There follows one or more statements, indented, executed only if `( condition )` is true.

Actually, the determination of leap years is more complicated. A year is regular unless it is divisible by 4, unless it is divisible by 100, unless it is divisible by 400! Suppose we want to set the logical variable `is_leap` for a given year `y`. One way to express this is with a nested `if/elif/else` statement that makes the decision in what seems a backwards order:

```
    if ( y % 400 == 0 ):
      is_leap = True
    elif ( y % 100 == 0 ):
      is_leap = False
    elif ( y % 4 == 0 ):
      is_leap = True
    else:
      is_leap = False
```

4 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 $2.75. In this situation, when the first condition is not true, we can't simply use an `else` because there are more decisions to be made. Instead, we check a second condition, with an `elif()` statement. And if that fails, then we have a final `else` statement to deal with. This would look like:

```
if ( age <= 5 ):
  fare = 0.0
elif ( age <= 11 ):
  fare = 1.35
else:
  fare = 2.75
```

Although we don't know what a `for()` statement is yet, I want to use it here to make it convenient to check our work. As written, this `for` statement carries out the following commands for the three cases of age 65, 3, and 9:

```
for age in [ 65, 3, 9 ]:

  if ( age <= 5 ):
    fare = 0.0
  elif ( age <= 11 ):
    fare = 1.35
  else:
    fare = 2.75

  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.

Finally, note that the indentation is very important here. The `for()` statement is going to repeat all the following statements that are indented. The `if()`, `elif()` and `else` statements each control a single indented statement that follows them. The `print()` statement is not part of the block of `if()` statements, but it is in the `for()` block, so it is executed once for each value of `age`.

Python uses indentation to define blocks of statements, whereas other languages use curly brackets { and }, or statements like `end` or `end if` to do so.

5 Is x in interval $[a, b]$?

The quantity used as a condition can be a more complicated expression involving the logical operators `or`, `and`, `not`. Suppose we wanted to evaluate a logical variable `inside` which is to be set to `True` if the number x is anywhere in the interval $[a, b]$. In other words, $a \leq x \leq b$. It is tempting to think you could write this as

```
if ( a <= x <= b ):

 # Warning!  This is not the way to express this condition!
```

but this sort of expression is not legal in Python. Instead, we need to write this as two comparisons, joined with an `and`.

```
if ( a <= x and x <= b ):
  inside = True
else
  inside = False
```

Similarly, if we were setting a variable `outside`, which is to be True if x is not inside the interval $[a, b]$, we could use an `or` operator:

```
if ( x < a or b < x  ):
  outside = True
else
  outside = False
```

If we already defined `inside`, then we could instead easily set `outside` as the negation of `inside`:

```
outside = not inside
```

Because this particular example is very simple, you could even evaluate `inside` in a single statment, without using an `if()` statement:

```
inside = ( a <= x and x <= b )
```

If your condition becomes complicated, it will be important to use parentheses so that Python does what you want. For instance, if you are testing whether x is in either interval $[a, b]$ or $[c, d]$, you might write

```
if ( a <= x and x <= b or c <= x and x <= d ):

# Danger!  This probably does not express what you want
```

but you should realize that this statement is ambiguous. Python will probably read it from left to right. Can you think of an example whiere it will not make the correct conclusion?

The statement needs to be rewritten as

```
if ( ( a <= x and x <= b ) or ( c <= x and x <= d ) ):
```

6 Do two intervals intersect?

Since we have been thinking about intervals, can we use an `if()` statement to determine whether the intervals $[a, b]$ and $[c, d]$ intersect? It is not, at first, clear what has to happen. But let's suppose the first interval is in a fixed position, and imagine the second interval sliding along from left to right. There is no intersection as long as $d < a$. Then the two intervals will overlap for a while. The value d will sail out the other end, but that's OK, until the value c just exceeds b. So there is no intersection if $d < a$ or if $b < c$.

```
if ( d < a or b < c ):
  intersect = False
else:
  intersect = True
```

and we could rewrite this in one line as

```
intersect = not ( d < a or b < c )
```

This statement is a bit complicated. It's essentially telling us when something will not happen, and then negating it. We'd rather describe the situation in positive terms. So when will we have an intersection. We can't have $d < a$ AND we can't have $b < c$, in other words,

```
if ( a <= d and c <= b ):
  intersect = True
else:
  intersect = False
```

and we could rewrite this in one line as

```
intersect = ( a <= d and c <= b )
```

You will sometimes find that you want to describe the negation (reverse) of a given logical condition. The simplest rule is that a pair of statements joined by an AND turn into a pair of the negated statements joined by an OR. Similarly, if the pair is joined by an OR, then the negated pair are joined by AND.

7 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 )

# Warning: This code is slightly incorrect!
```

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?

8 Playing Craps

Here is another example in which the `if/else` statements contain another layer of `if/elif`'s. A game of craps begins with a player rolling the dice to define the point. If by chance this is a 7 or 11, the player immediately wins the bet, and if the point is 2, 3, or 12, the player loses the bet and the game is over. Otherwise, the player now rolls repeatedly until either getting the point again, and winning, or getting a 7, and losing.

In the following code, the `while()` statement repeats the commands until a `break` statement is encountered to end the game.

```
roll = 0
bet = 100

while ( True ):

  roll = roll + 1
  dice = rng.integers ( low = 1, high = 6, size = 2, endpoint = True )
  rolled = sum ( dice )

  if ( roll == 1 ):

    point = rolled

    if ( point == 7 or point == 11 ):
      result = + bet
      break
    elif ( point == 2 or point == 3 or point == 12 ):
      result = - bet
      break

  else:

    if ( rolled == point ):
      result = + bet
      break
    elif ( rolled == 7 ):
      result = - bet
      break

  roll = roll + 1
```

9 The Collatz conjecture

Start with any (positive) integer `n`, and start the Collatz iteration:

```
If n is even, divide it by 2,
else if n is odd, multiply it by 3 and add 1.
If n is now 1, stop.
Otherwise repeat the process, if necessary, forever.
```

The Collatz conjecture is:

For any positive initial value `n`*, the Collatz iteration terminates in finitely many steps.*

The Collatz conjecture was made in 1937. It 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:

```
 #    n    operation
--  ---    ---------
 0   37
 1  112 <- 3*37 + 1
 2   56 <- 112/ 2
 3   28 <- 56 / 2
 4   14 <- 28 / 2
 5    7 <- 14 / 2
 6   22 <- 3*7+1
 7   11 <- 22 / 2
 8   34 <- 3*11 + 1
 9   17
10   52
11   26
12   13
13   40
14   20
15   10
16    5
17   16
18    8
19    4
20    2
21    1
DONE in 21 steps, highest value was 112
```

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 = 0

while ( n != 1 ):

  if ( n % 2 == 0 ):
    n = n // 2              # // forces integer division!
  else:
    n = 3 * n + 1

  i = i + 1

print ( 'From', n, ' to 1 in', i, 'steps.' )
```

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. We will worry about `while()` statements that go crazy when it is time to talk about them seriously.

However, suppose that, for some special value of n, our Collatz program did run forever. What could be going on? There are two possibilities. We are generating an infinite sequence using a deterministic rule. Either the sequence consists of an infinite number of distinct integers, which means that eventually the numbers become arbitrarily large. Or it is possible that the sequence forms a loop, repeating the same values over and over. If so, could this perpetual loop be just a single magic value n? (There is a quick answer to this question!)