EC2401: INTRODUCTION TO DATA SCIENCE

LECTURE 1: CONTROL FLOW -- BRANCHING, ITERATION, FUNCTIONS

Evan Piermont

Problem of the Week

Two prisoners each have a hat: either red or black. Each can see the other's hat but not her own. They are given the opportunity of release if just one of them can work out the color of her own hat. They can come up with a strategy beforehand, but cannot communicate once the hats are put on. What is their best strategy?

The difference between a calculator and a computer is the latter's ability to string together calculations (manipulation of bits) depending on the state-of-the-system.

A fundamental building block is the ability to different calculations depending on some condition.

We need a way to branch:

```
if CONDITION (do A) else (do B).
```

Where `CONDITION` evaluates as a Boolean.

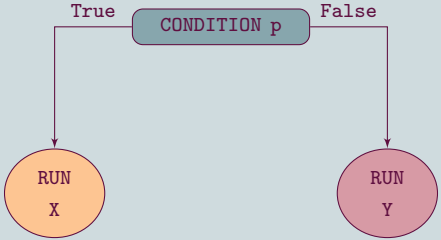
Booleans as Branches

Consider a Boolean p: we want to do two separate things, depending on the truth of p

- if `p` is `True` then run program `X`
- if `p` is `False` then run program `Y`

In python this is:

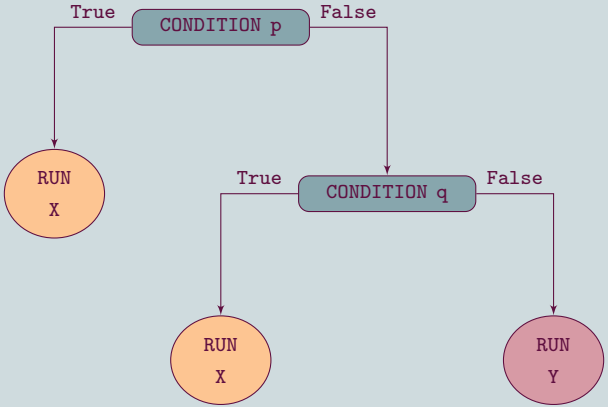
```
if  p:
    X
else:
    Y
```


True
CONDITION p
False
RUN
X
RUN
Y

Booleans as Branches

What if we want to

- do `X` if `p` is true
- but, if `p` is false:
  - do `X` if `q` is true
  - do `Y` if `q` is false


True
CONDITION p
False
RUN
X
True
CONDITION q
False
RUN
X
RUN
Y

Booleans as Branches

There are multiple ways to do this:

```
if  (p or q):
    X
else:
    Y
```

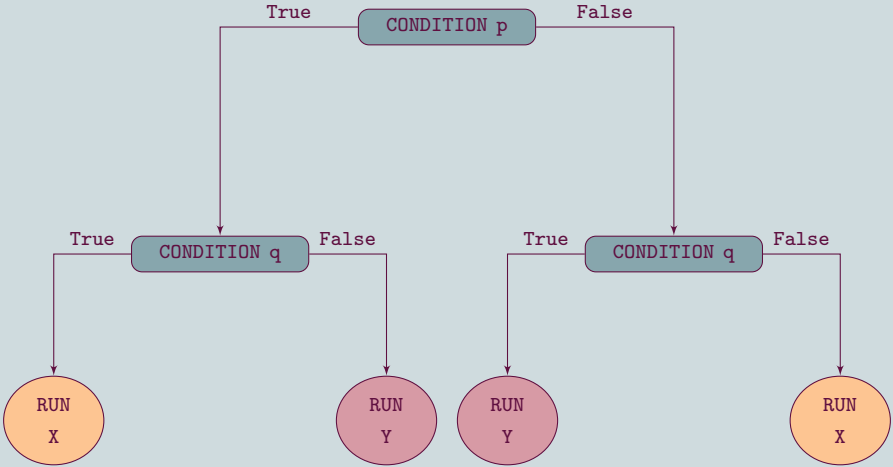
```
if  p:
    X
elif q:
    X
else:
    Y
```

```
if p:
    X
else:
    if q:
        X
    else:
        Y
```

What if we want to

- if `p` is true:
  - do `X` if `q` is true
  - do `Y` if `q` is false
- if `p` is false:
  - do `Y` if `q` is true
  - do `X` if `q` is false

This seems like it needs lots of conditions.


True
CONDITION p
False
True
CONDITION q
False
True
CONDITION q
False
RUN
X
RUN
Y
RUN
Y
RUN
X

We can recreate this tree 'exactly' as

```
if  p:
    if  q:
        X
    else:
        Y
else:
    if  q:
        Y
    else:
        X
```

Or we can construct a new Boolean variable '`p==q`'. This is true when the values of `p` and of `q` coincide.

```
if  (p==q):
    X
else:
    Y
```

Iteration

We often want to do something over and over again:

- Calculating the average for each group
- Constructing a new variable for each observation
- Plotting each data point on a graph

For physical calculations, this often amounts to a linear increase in work.

Perhaps we want the average of each of list in a lists:

`students = [[5,7,6],[2,2,8],[4,6,2]]`

```
avg = []
grades0 = students[0]
total0 = sum(grades0)
avg.append(total0/len(grades0))
grades1 = students[1]
total1 = sum(grades1)
avg.append(total1/len(grades1))
grades2 = students[2]
total2 = sum(grades2)
avg2 = avg.append(total2/len(grades2))
```

While the computer's work must increase, our work need not. We can use iteration.

Iteration is the reuse of a single piece of code over and over: write once, execute multiple times.

For Loops

The simplest case of iteration is a for loop. The syntax is:

```
for i in  iterable:
    CODE THAT DEPENDS ON i
```

- An iterable is a Python object that can be iterated over: lists, ranges, dictionaries, etc
- `i` is a variable that will change it value, setting once to each element of the iterable
- The inner code will be executed once for each value of `i`

```
for i in [x,y,z]:
    X(i)
```

- Set `i = x`:
  - run `X(x)`
- Set `i = y`:
  - run `X(y)`
- Set `i = z`:
  - run `X(z)`

List Comprehension

If we want to loop over a list to construct another list there is an easy shorthand syntax:

```
newList = [CODE THAT DEPENDS ON i for i in oldList]
```

is the same as

```
newlist = []
for i in oldList:
    out = CODE THAT DEPENDS ON i
    newList.append(out)
```

While Loops

Another type of loop is a while loop. The syntax is:

```
while p:
    CODE THAT MODIFIES p
```

- `p` is a Boolean variable
- The inner code will be executed as long as `p==True`
- The loop ends when `p==False`

```
while p:
    X
```

1. Check `p`:
   - if `True`: `run X` and go to step 1
   - if `False`: go to step 2
2. `END LOOP`

Here is a link to live examples. .

Functions

Often we want to run some program many times but in different contexts.

A **function** is a program that takes inputs and returns outputs (just like a mathematical function).

We can construct functions and name them, so as to call them later.

To create a function, the syntax is:

```
def funcName(x):
     CODE THAT DEPENDS ON x
     AND DETERMINES z
    return z
```

```
def funcName(x):
     CODE THAT DEPENDS ON x
     AND DETERMINES z
    return z
```

- The variable `x` is the input to the function
  - There can be multiple input variables
  - Use: `def funcName(x,y,z)`
- The returned value `z` is the output of the function
  - A function returns a single value, but it can be a list, tuple, etc
- We can then call the function by `funcName(a)` which will evaluate to the returned value setting `x=a`

```
def inc(x):
    return x+1
```

- `inc(0)` is equivalent to `1`
  - since the function `inc` returns `1` on a input of `0`
- Likewise `inc(5)` is equivalent to `6`
  - so `inc(0) + inc(5)` will yield `7`
- `inc` is a function object, this is just another data type

```
def printInc(x):
    print(x+1)
```

When you call `printInc` this, it might seem the same as calling `inc(0)`: both will display an output of `1` in your terminal.

They are not the same!

- `inc(0)` is equivalent to `1`
- `printInc(0)` is just a print statement
  - This is not a piece of data
  - It cannot be used in further calculations

Why use functions?

- Don’t repeat yourself!
- Help keep track of which part of code does what (use evocative names!)
- Allows for abstraction — solve two similar problems with the same parts
- Propagate changes globally.

Functions

Just like a mathematical function, a Python function has a

- Domain The type of objects that can be inputs
- Co-Domain The type of objects that can be inputs

For example, the function `split()`:

$$\texttt{STRING} \times \texttt{STRING} \rightarrow \texttt{LIST}$$

$$(\text{‘the big cat’}, \text{‘ ’}) \mapsto [\text{‘the’}, \text{‘big’}, \text{‘cat’}]$$

$$(\text{‘abcde’}, \text{‘x’}) \mapsto [\text{‘abcde’}]$$

$$(\text{‘zzxzzxzz’}, \text{‘x’}) \mapsto [\text{‘zz’}, \text{‘zz’}, \text{‘zz’}]$$

Some functions are polymorphic they act on many different types. For example `append`:

$$\texttt{LIST} \times \texttt{ANY} \rightarrow \texttt{LIST}$$

([1,2,3], 4) ↦ [1,2,3,4]	`INT`
(['a','b','c'], 'd') ↦ ['a','b','c', 'd']	`STR`
([x,y], [z]) ↦ [x,y,[z]]	`LIST`

It is best practice (and often strictly necessary) to consider the space where your functions act when constructing them. What are the domains and co-domains of the following build in functions?

- `replace`
- `sum`
- `len`

Functions can be called from within other functions and composed with on another. For example:

```
def square(x):
    return x*x

def halve(x):
    return x/2

x = square(halve(8))
y = halve(square(8))
```

will evaluate `x = 16` and `y = 32`

Functions are objects!. This means that they are just another type of data:

- If you have take set theory/discrete math classes this is no surprise (functions are points in a product space)
- This means functions can treated as data—in particular passed as inputs to other functions

```
def twice(f,x):
    return  f(f(x))
```

- The above takes a function and an argument and applies the function twice
- For instance, `twice(halve, 16)` will output 4

There are some helpful built in Python functions that take functions as inputs.

- `map` applies a function to each element of an iterable
- `filter` extracts elements from an iterable for which a function returns True
- `sorted` sorts the elements of a given iterable in a specific order according to the output
- `reduce` apply a function to an iterable to reduce it to a single cumulative value

Map

$$\texttt{MAP:} \qquad \texttt{FUNC} \times \texttt{LIST} \to \texttt{LIST}$$
$$(f, [x, y, z]) \mapsto [f(x), f(y), f(z)]$$

$$\begin{array}{ccccc} [& x, & y, & z &] \\ & \Downarrow & \Downarrow & \Downarrow & \\ [& f(x), & f(y), & f(z) &] \end{array}$$

```
words = [`Another', `List', `Of', `Strings']
list(map(len, words))
```

- The above applies `len`, which returns the length of a string, to each element of ‘words’
- The final line would output as `[7,4,2,7]`
- The whole thing is wrapped in the `list` function because ‘map’ actually returns a generator object

```
ns = [1, 2, 3, 4, 5, 6]

def even(n):
    if n % 2 == 0:
        return True
    return False

list(filter(even, ns))
```

- The above applies `even` to each element, and only keeps those that return `True`
  - Recall, many objects evaluate to `True` when converted to Booleans
- The final line would output as `[2,4,6]`

Recursion

Recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem.

- In particular: by using functions that call themselves from within their own code.
- Essentially, allows for iteration within a function definition

Consider the problem of finding the factorial of n: `fact(n)` $= 1 \times 2 \times \ldots \times n$.

- Perhaps we could solve for each n in order

Consider the problem of finding the factorial of n: `fact(n)` $= 1 \times 2 \times \ldots \times n$.

- Perhaps we could solve for each n in order
- We start with 1, then `fact(1)` is by definition 1

Consider the problem of finding the factorial of n: `fact(n)` $= 1 \times 2 \times \ldots \times n$.

- Perhaps we could solve for each n in order
- We start with 1, then `fact(1)` is by definition 1
- Then `fact(2)` is $1 \times 2 =$ `fact(1)` $\times 2 = 2$

Consider the problem of finding the factorial of n: `fact(n)` $= 1 \times 2 \times \ldots \times n$.

- Perhaps we could solve for each n in order
- We start with 1, then `fact(1)` is by definition 1
- Then `fact(2)` is $1 \times 2 =$ `fact(1)` $\times 2 = 2$
- Then `fact(3)` is $1 \times 2 \times 3 =$ `fact(2)` $\times 3 = 6$

Consider the problem of finding the factorial of n: `fact(n)` $= 1 \times 2 \times \ldots \times n$.

- Perhaps we could solve for each n in order
- We start with 1, then `fact(1)` is by definition 1
- Then `fact(2)` is $1 \times 2 =$ `fact(1)` $\times 2 = 2$
- Then `fact(3)` is $1 \times 2 \times 3 =$ `fact(2)` $\times 3 = 6$
- Notice the pattern: `fact(n)` = `fact(n-1)` $\times n$

```
def fact(n):
    if n == 1:
        return  1
    return fact(n-1)*n
```

- The above applies exactly the logic from the last slide:
- If $n = 1$, return 1 by definition
- Otherwise, recursively call `fact` in the smaller problem

Here is a link to live examples.