Functions and Variables

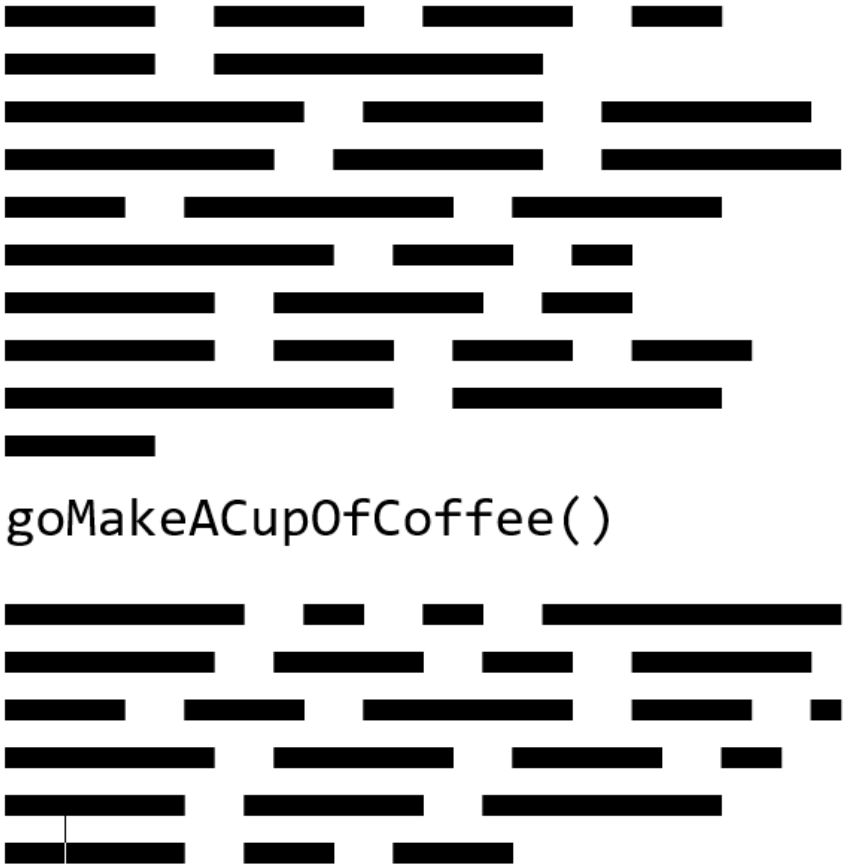
GETTING DATA INTO AND OUT OF FUNCTIONS

Functions (Recap)

Last time we saw that functions are a way of ‘going off and doing something’ then ‘coming back’ to where we were before.

So we can think of it as:

1. Reading a book,
2. Pausing to go and make a cup of coffee,
3. Returning and resuming reading.



Functions (Recap)

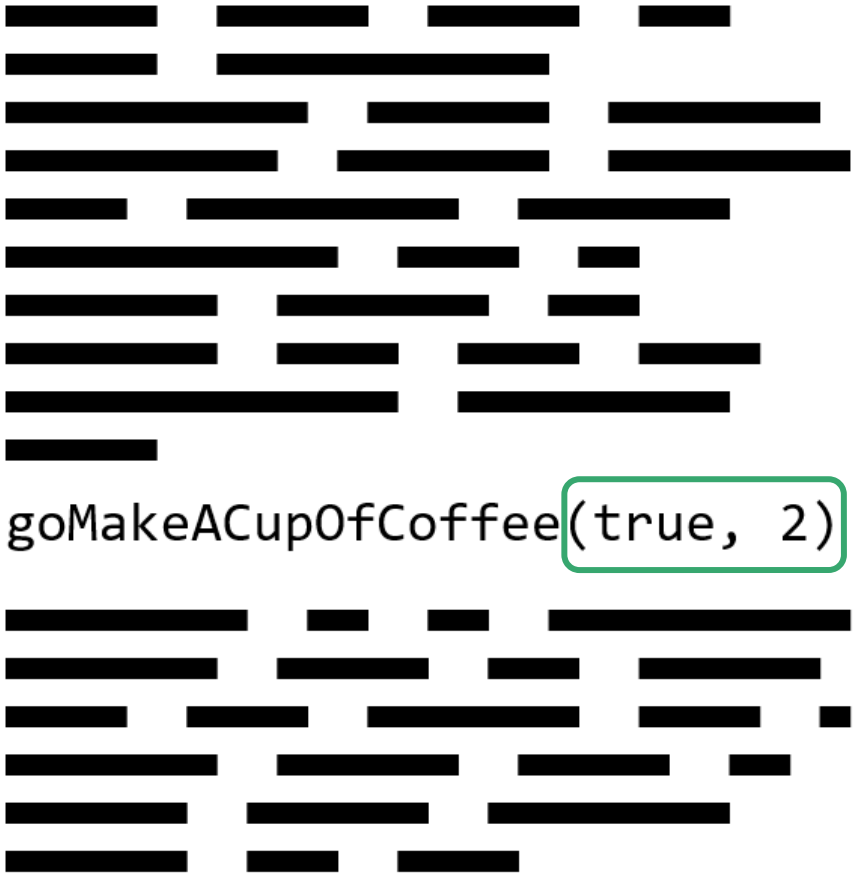
The most important thing to remember is: calling a function means ‘going off and doing something’ ***then*** ‘coming back’ to where we were before.

So, if you have code that looks like this, you will **not** make two cups at the same time, you will instead make one cup, **then** come back and think “what’s next” “oh yes make a cup of coffee”…

Sending Data **IN**

Often, we want to send data *into* the function. We do this by including it between the brackets when we call the function.

Here we are calling sending '**true**' and '**2**' into the function call.



Using that Data

Looking at the goMakeACupOfCoffee function, we see that the first line tells us the type and the name of the function that it is known us as well as the type of data used within the function.

We see that the function; gets a cup *(which may be another function)*, then uses the data that was *handed* in, to make some decisions.

```
void goMakeACupOfCoffee(bool milk, int sugar) {
  ...
  ...
  ...
  getCup();
  if (milk) {
    addMilk();
  }
  for (int i = 0; i < sugar; i++) {
    addSugar();
  }
  ...
}
```

Using that Data

The first *argument* ‘true’ is of type bool and being locally called “milk”.

We can see that if it is *true* we add milk and if it is *false* we don’t.

```
void goMakeACupOfCoffee(bool milk, int sugar) {
  ...
  ...
  ...
  getCup();
  if (milk) {
    addMilk();
  }
  for (int i = 0; i < sugar; i++) {
    addSugar();
  }
  ...
}
```

Using that Data

The second *argument* '2' is of type int and being locally called "sugar".

We see that we get to a *loop*, start a local *counter* and then go round that loop as many times as we were told to.

If sugar was '0' we would not go round the loop at all, so no sugar would be in the coffee.

As it is 2 we loop twice and add 2 sets of sugar

```
void goMakeACupOfCoffee(bool milk, int sugar) {
  ...
  ...
  ...
  getCup();
  if (milk) {
    addMilk();
  }
  for (int i = 0; i < sugar; i++) {
    addSugar();
  }
  ...
}
```

Getting Data **OUT**

Equally often, we want to get data *out of* the function. We do this by *returning* it as we exit the function.

Here we saying that goMakeACupOfCoffee will return a value and that we want to take that data and store it in the variable called “itWorked”.

Returning that Data

But, to do that we need to change our function.

Firstly we need to say that it will return some data, we do this by declaring a type in the function *signature*.

Previously the function didn't return anything so the return type was void but now it will return true or false so it is bool.

```
bool goMakeACupOfCoffee(bool milk, int sugar) {
  ...
  ...
  ...
  getCup();
  if (milk) {
    addMilk();
  }
  for (int i = 0; i < sugar; i++) {
    addSugar();
  }
  ...
}
```

Returning that Data

Next we need to actually return the value. We are later storing it in “itWorked”, so we can assume that it will be ‘true’ if we end up with a cup of tea, and ‘false’ if something went wrong.

The thing we want to check might be if we have any clean cups.

We can add an if statement that checks that first.

```
bool goMakeACupOfCoffee(bool milk, int sugar) {
  if (noCleanCups()) {
    return false;
  }
  getCup();
  if (milk) {
    addMilk();
  }
  for (int i = 0; i < sugar; i++) {
    addSugar();
  }
  return true;
}
```

Returning that Data

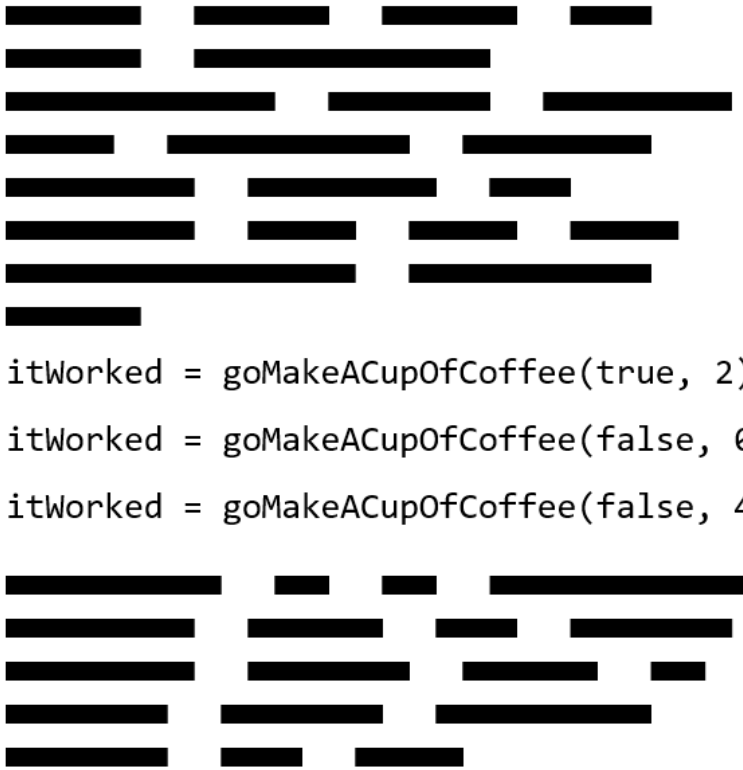
Note: If the computer ever gets to a return statement in a function it *immediately* exits that function.

So, we can read the program as:
If we don't have and clean cups give up saying 'false',
but if we do then carry on…

Finally, as the function **must** return a bool we need to remember to add a line to return 'true' if all was fine.

```
bool goMakeACupOfCoffee(bool milk, int sugar) {
  if (noCleanCups()) {
    return false;
  }
  getCup();
  if (milk) {
    addMilk();
  }
  for (int i = 0; i < sugar; i++) {
    addSugar();
  }
  return true;
}
```

Following it all – Worked example



```
itWorked = goMakeACupOfCoffee(true, 2)
itWorked = goMakeACupOfCoffee(false, 0)
itWorked = goMakeACupOfCoffee(false, 4)
```

It is very important to be able to follow the code *execution flow* through a program and that is often best done on paper or a whiteboard…

```
bool goMakeACupOfCoffee(bool milk, int sugar) {
  if (noCleanCups()) {
    return false;
  }
  getCup();
  if (milk) {
    addMilk();
  }
  for (int i = 0; i < sugar; i++) {
    addSugar();
  }
  return true;
}
```

Returning that Data

We will assume that we have enough coffee milk and sugar but we only have 2 clean cups…

As we follow the code we see that the function is called first with true and 2, and we know that we currently have 2 clean cups.
We follow the code through:

1. There is at least 1 clean cup so it **skips** that ‘return false’ line
2. Milk is true so it does run that line and adds milk
3. Sugar is 2 so it loops twice incrementing the counter each time
4. Finishes returning true

Returning that Data

Then the function is called again but we know that we now only have 1 clean cups

This time the function is called with false and `0`.
We again follow the code through:

1. There is at least 1 clean cup so it skips that ‘return false’ line
2. Milk is false so this time it skips that line and doesn’t add milk
3. Sugar is `0` so it **doesn’t** loop at all, so no sugar added
4. Finishes returning true

Returning that Data

Finally, the function is called one last time but we know that we don't have any clean cups, lets see how that changes things

This time the function is called with false and 4.
As usual we follow the code:

1. There are no clean cups so it **runs** that 'return false' line – **immediately** exiting returning the value as false

The later lines to do with milk and sugar are **never** run.