User-defined Functions
Case study

A function to return the average of readings on an analog input channel

ME 121: Portland State University
Scenario is reading a nominally steady signal that has some noise.

Average = 681
Scenario is reading a nominally steady signal that has some noise.

Average = 681

Analog input value

Time (s)
Compute the average of $n$ readings

The standard formula is \[ \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \]

In order for the code to work for any $n$, the calculation needs to be implemented as an accumulating sum:

```
float ave, sum;
sum = 0.0
for (int i=1; i<=nread; i++) {
    sum += analogRead(pin);
}
ave = sum/float(nread);
```

Suppose the $x_i$ come from analog readings:

\[
\begin{align*}
 s & \leftarrow 0 \\
 s & \leftarrow s + x_1 \\
 s & \leftarrow s + x_2 \\
 & \vdots \\
 s & \leftarrow s + x_n \\
 \bar{x} & = \frac{s}{n}
\end{align*}
\]
Our Goal

Write a function that contains the code for averaging a reading

Inputs: pin, nread

```c
float sum, ave;

sum = 0.0;
for (int i=1; i<=nread; i++) {
    sum += analogRead(pin);
}

ave = sum/float(nread);
```

Output: ave
Our Goal

Once it is written, the function is a black box

Inputs: pin, nread

Output: ave
User-defined functions have many benefits

1. Code is debugged once and then reused
   a. Can be treated like a black box
   b. Code modules can be remixed in similar codes

2. Functions make main function (loop) more compact
   a. Easier to read and understand
   b. Avoid copy/pasting of identical or nearly-identical code

3. Structured code is more robust
   a. Computations inside function only affected by inputs
   b. Modules reduce accidental modifications to working code
   c. More easily used in new applications
The end result is a function called `sensorAverage`

```c
float sensorAverage(int pin, int nread) {
    float ave, sum;

    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }
    ave = sum/float(nread);
    return(ave);
}
```
A simple program to make average readings on an analog input channel

```c
void setup() {
    Serial.begin(9600);
}

void loop() {
    int sensor_pin = 3, nave = 15;
    float reading;

    reading = sensorAverage(sensor_pin, nave);
    Serial.println(reading);
}

float sensorAverage(int pin, int nread) {
    float ave, sum;

    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }
    ave = sum/float(nread);
    return(ave);
}
```
Every function has a name

```cpp
void setup() {
    Serial.begin(9600);
}

void loop() {
    int sensor_pin = 3, nave = 15;
    float reading;

    reading = sensorAverage(sensor_pin, nave);
    Serial.println(reading);
}

float sensorAverage(int pin, int nread) {
    float ave, sum;

    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }
    ave = sum/float(nread);
    return(ave);
}
```

Function name is a character string limited to the characters that can be used as Arduino variables.
Every function has a return type

```cpp
void setup() {
    Serial.begin(9600);
}

void loop() {
    int sensor_pin = 3, nave = 15;
    float reading;

    reading = sensorAverage(sensor_pin, nave);
    Serial.println(reading);
}

float sensorAverage(int pin, int nread) {
    float ave, sum;
    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }
    ave = sum/float(nread);
    return(ave);
}
```

Type of return value can be any valid variable type: int, long, float, char, ...

Inside body of the function, a variable or expression needs to be created that is consistent with the return type.
The return type specifies what can be received in the calling function

```cpp
void setup() {
    Serial.begin(9600);
}

void loop() {
    int sensor_pin = 3, nave = 15;
    float reading;

    reading = sensorAverage(sensor_pin, nave);
    Serial.println(reading);
}

float sensorAverage(int pin, int nread) {
    float ave, sum;

    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }
    ave = sum/float(nread);
    return(ave);
}
```
The function body is a block of code enclosed in a matching pair of brackets.

void setup() {
    Serial.begin(9600);
}

void loop() {
    int sensor_pin = 3, nave = 15;
    float reading;
    reading = sensorAverage(sensor_pin, nave);
    Serial.println(reading);
}

float sensorAverage(int pin, int nread) {
    float ave, sum;
    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }
    ave = sum/float(nread);
    return(ave);
}
Indent the body of the function. Not required by Arduino, but **Strongly Recommended**

```cpp
void setup() {
    Serial.begin(9600);
}

void loop() {
    int sensor_pin = 3, nave = 15;
    float reading;

    reading = sensorAverage(sensor_pin, nave);
    Serial.println(reading);
}

float sensorAverage(int pin, int nread) {
    float ave, sum;
    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }
    ave = sum/float(nread);
    return(ave);
}
```
Input arguments are local variables that are visible only in the function body

```cpp
void setup() {
    Serial.begin(9600);
}

void loop() {
    int sensor_pin = 3, nave = 15;
    float reading;

    reading = sensorAverage(sensor_pin, nave);
    Serial.println(reading);
}

float sensorAverage(int pin, int nread) {
    float ave, sum;

    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }
    ave = sum/float(nread);
    return(ave);
}
```

Input arguments must be defined by type and name
User-defined functions

void setup() {
    Serial.begin(9600);
}

void loop() {
    int sensor_pin = 3, nave = 15;
    float reading;

    reading = sensorAverage(sensor_pin, nave);
    Serial.println(reading);
}

float sensorAverage(int pin, int nread) {
    float ave, sum;

    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }

    ave = sum/float(nread);
    return(ave);
}

Input arguments are local variables that are visible only in the function body

Function input values are copied by position. The variable name is not passed into the function.

Input argument names and values are local to the function.
Input arguments are local variables that are visible only in the function body.

```cpp
void setup() {
    Serial.begin(9600);
}

void loop() {
    int sensor_pin = 3, nave = 15;
    float reading;
    reading = sensorAverage(sensor_pin, nave);
    Serial.println(reading);
}

float sensorAverage(int pin, int nread) {
    float ave, sum;
    sum = 0.0;
    for (int i=1; i<=nread; i++) {
        sum += analogRead(pin);
    }
    ave = sum/float(nread);
    return ave;
}
```

Function input values are copied by position. The variable `name` is not passed into the function.
Impatient programmers use global variables instead of inputs

1. Global variables bypass input arguments
   a. Can be a quick fix, but quick fixes often cause problems later after the quick benefit wears off
   b. Simple use of digital outputs require global variables
      ➡ Object-oriented code can avoid this problem

2. Restrict communication to input and output variables
   a. May require more work at first
   b. Code will be more robust

Wherever possible, avoid global variables. Use function input and output arguments.
Summary of user-defined functions

Function type determines what is returned

❖ void functions return nothing
  ‣ Examples: print values to serial monitor, update LCD panel

❖ int, float functions return numerical values
  ‣ Examples: compute average of an input; evaluate a calibration equation

Function inputs are determined by position

❖ Type of variable in the calling sequence should match type of variable in input argument

❖ The names of arguments do not need to match
  ‣ Functions are like black boxes: we don’t need to know how the function works as long as we use inputs and outputs correctly.
Summary of user-defined functions

Function bodies are isolated code blocks

- Functions isolate tasks and make the code modular
- The body of the function has an isolated scope
  - Variables in the function are limited to the scope of the function
  - Local variables cannot be affected by external code blocks unless the local variables are global — Avoid global variables

Functions allow reuse

- Function code can be copies into new sketches.
- Function code can be put into external files
- Function code can be built into libraries