From Mathematics we know that *functions* perform some operation and return *one* value.
Functions “encapsulate” the performance of some operation, so it can be used by others. (for example, the sqrt() function)

Consider a function to convert Celsius to Fahrenheit:

Formula: \[ F = C \times 1.8 + 32.0 \]

Functional notation: \[ F = g(C) \text{ where } g(C) = C \times 1.8 + 32.0 \]
Invocation

Math: $F = g(C)$
Python: $F = g(C)$

Terminology: argument “C”

Definition

Math: $g(C) = C \times 1.8 + 32.0$
Python:
```python
def g(C):
    return C*1.8 + 32.0
```

Terminology: parameter “C”
Terminology

- **Invocation (Call)**
  \[ F = g(C) \]

- **Definition**
  \[
  \text{def } g(C): \\
  \text{return } C*1.8 + 32.0
  \]

Python function call (invocation):

- **function name; must have previously been defined**
  \[ F = g(0.0) \]

- **in parenthesis is a list of arguments (expressions)**
F = g(C)

1. Call copies argument C to parameter Temp

2. Control transfers to function “g”

def g(Temp):
    return Temp*1.8 + 32.0

F = g(C)

3. Expression in g is evaluated

def g(Temp):
    return Temp*1.8 + 32.0

4. Value of expression is returned to the invoker
def celsius2Fahrenheit(x):
    return x*1.8 + 32.0

c = 100
f = celsius2Fahrenheit(c)
print(f)

Operation

Defines celsius2Fahrenheit (associates the function body to the name celsius2Fahrenheit)

c = 100
f = celsius2Fahrenheit(c)
print(f)

Associates the value 100 to the variable c.
```python
def celsius2Fahrenheit(x):
    return x*1.8 + 32.0

c = 100
f = celsius2Fahrenheit(c)
print(f)
```

Invocation:
- Creates a local namespace for `celsius2Fahrenheit`.
- Associates the value of the parameter `x` to argument `c` (associates 100 with `x`).

Operation
- Transfers control to function `celsius2Fahrenheit`.
- `c` is associated with `100`.
- `x` is associated with the value of the parameter.

Transfers control to function `celsius2Fahrenheit`.
def celsius2Fahrenheit(x):
    return x * 1.8 + 32.0

c = 100
f = celsius2Fahrenheit(c)
print(f)

Operation

Return command:
evaluates the expression

Parameter x exists only in the function celsius2Fahrenheit’s namespace

f added to namespace and associated with 212.
Function namespace deleted.
Operation

```
def celsius2Fahrenheit (x):
    return x*1.8 + 32.0;
    ...
c = 100
    ...
f = celsius2Fahrenheit(c)
    print f
    ...
```

Name space

celsius2… = …
c = 100
f = 212

Continues sequential execution

Verbose Python Invocation

Math: \( F = g(C) \)
Python:

```
cels_temp = 100
fahr_temp = celsius_to_fahrenheit(cels_temp)
print(cels_temp,'in Fahrenheit is:',fahr_temp)
```
Example 16
simpleFns

This example illustrates the construction and use of several functions.

Verbose Python Definition

Math: \( g(C) = C \times 1.8 + 32.0 \)
Python:

```python
def celsius_to_fahrenheit(cels_temp):
    return cels_temp*1.8 + 32.0
```
Program Abstraction

1. Get Temperature
2. Convert Temperature
3. Display Temperature

Needed Behavior

Please enter a temperature in degrees Celsius: 19.5
Original: 19.5 C
Equivalent: 67.1 F
def get_temp():
cels_temp = input("Please enter \ntemperature in degrees \nCelsius: ")
return float(cels_temp)

def celsius_to_fahrenheit(cels_temp):
result = cels_temp*1.8+32.0
return result
Display results

```python
def display(cels_temp, fahr_temp):
    print("Original: ", cels_temp)
    print("Equivalent: ", fahr_temp)
```

The function arguments are passed *in order* to the function parameters. The names need not match.
When a function runs, it defines a new namespace. The names in the function’s namespace are only available to the function.

Passing arguments by reference:
The first argument passes its namespace reference to the first parameter, second to second, and so on.

What gets passed?
my_int = 25
my_function(my_int)
print(my_int)
def my_function(an_int):
    print(an_int)

Both namespaces now refer to the same object: the reference got passed.

What happens if the function changes that variable?
The object (int) is immutable: it wasn’t changed.

The function namespace simply updated its reference to a new object. The reference in the calling program was unaffected.
Local Objects

A function has its own namespace ("local"), so
- a parameter is in the function’s namespace so any use outside the function is an error.
- any variable assigned in the function is in the function’s namespace so it is not available outside of the function.
Why Functions?
Functions for better design

- Functions are very useful to break the program down into small, understandable, maintainable pieces
- Example:
  - `def get_temp(None)
  - def celsius_to_fahrenheit(temp)
  - def display(fahr_temp, cels_temp)``

Software engineering

- There is a discipline of computer science dedicated to the systematic development and maintenance of software.
- There are a number of approaches that SE use: modularization, proveability, testing, refactoring and others.
Refactoring:
reexamine code to improve its readability while not changing its functionality.

For example, extracting complicated code into multiple functions, creating better abstractions.

**Function “rules of thumb”**

- Should do one thing.
  A function *abstracts* one idea.
- Should not be overly long (~one page of code).
- Best if generic so it could be reused elsewhere.
Functions

- Reuse
- Abstraction (Encapsulation)

PerfectNumbers Fn
List passed but not returned

- In `classify_value`, a list is passed but never returned.
- This is because the calling program and the function share the same reference, and since the list is mutable a change in the function is a change to the calling program.

Our own data structure

- The variable `result` defines our own data structure:
  - the first element is a list
  - the second and third are integers
- We add perfect numbers to the first list, and keep the abundant and deficient in the two counts.
isolate complicated printing

- print_results isolates the messy formatting of the output
- hiding complicated details is one thing that functions do well.

mileage puzzle
Car Talk Puzzler

The Palindromic Odometer

Driving along, Terry notices that the last four digits on the odometer are palindromic. A mile later, the last five digits are palindromic. A mile later, the middle four digits are palindromic. One mile after that, all six are palindromic. What was the odometer reading when Terry first looked at it?

making a main function

- it is common to create the main function that starts the whole program running
- now when we run our file, it defines the main function which we must call manually.
extensive use of continue

- checks for ‘non-palindromes’ under the required circumstances and continues
- makes the process more efficient.

checking time

```
import time
start = time.time()
#... do stuff ...
end = time.time()
print('It took:', end-start, 'seconds')
```
refactoring

- what if you want to check a different approach to palindrome?
- It is easy to refactor this program. Provide a new function with a different definition to see the effect.
- functions make refactoring easier