Modules and Libraries

Modules and Libraries

Bulding and Using

Review of Functions

Review of Functions

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```

Review of Functions

parameters

↓ ↓

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```

Review of Functions

parameters

↓ ↓

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```
← *return statement*

Review of Functions

parameters

↓ ↓

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```

← *return statement*

```
out1, out2 = my_func(3, 2)
```

Review of Functions

parameters

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```
← *return statement*

return values *arguments*

```
out1, out2 = my_func(3, 2)
```

Review of Functions

parameters

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```
← *return statement*

return values *arguments*

```
out1, out2 = my_func(3, 2)
print(out1) # prints 5
```

Review of Functions

parameters

```
def my_func(a, b):
    x = a + b
    y = a - b
    return x, y
```
← *return statement*

return values *arguments*

```
out1, out2 = my_func(3, 2)
print(out1) # prints 5
print(out2) # prints 1
```

Procedural Programming

Procedural Programming

```
def func1(x, y):
    statement1
    statement2
        ⋮
    return z

def func2():
    statement50
    statement51
        ⋮
    return

    ⋮
```

Procedural Programming

```
def func1(x, y):
    statement1
    statement2
        ⋮
    return z

def func2():
    statement50
    statement51
        ⋮
    return

    ⋮
```

```
a few input statements
out1 = func1(input1,
input2)
func2()
out2 = func3(out1)
    ⋮
```

Procedural Programming

```
def func1(x, y):
    statement1
    statement2
        ⋮
    return z

def func2():
    statement50
    statement51
        ⋮
    return

    ⋮
```

```
a few input statements
out1 = func1(input1,
input2)
func2()
out2 = func3(out1)
    ⋮
print(func10(out9, out10))
```

Modularity

Modularity

Procedural programs embody the engineering principle of *modularity*:

Modularity

Procedural programs embody the engineering principle of *modularity*:

- Break a problem up into smaller problems

Modularity

Procedural programs embody the engineering principle of *modularity*:

- Break a problem up into smaller problems
- Build individual modules to solve each smaller problem

Modularity

Procedural programs embody the engineering principle of *modularity*:

- Break a problem up into smaller problems
- Build individual modules to solve each smaller problem
- Assemble modules to build full solution

Modularity

Procedural programs embody the engineering principle of *modularity*:

- Break a problem up into smaller problems
- Build individual modules to solve each smaller problem
- Assemble modules to build full solution
- Individual modules should have a standardized interface, so that they are easily interchangable or replacable

Modular temperatures.py

Modular temperatures.py

```
def F_to_C(tempF):
    return (tempF - 32) * 5 / 9
```

Modular temperatures.py

```
def F_to_C(tempF):
    return (tempF - 32) * 5 / 9

def C_to_F(tempC):
    return tempC * 9 / 5 + 32
```

Modular temperatures.py

```
def F_to_C(tempF):
    return (tempF - 32) * 5 / 9

def C_to_F(tempC):
    return tempC * 9 / 5 + 32

choice = input("Will your input be in deg F or C? ")
temp = float(input("Enter the temp to convert:  "))
if choice == "F":
    result = F_to_C(temp)
    print(f"That's {result} degrees C")
elif choice == "C":
    result = C_to_F(temp)
    print(f"That's {result} degrees F")
else:
    print("Sorry that was not a legal input")
```

Function Signature

Function Signature

The “signature” is what you need to know to *call* a function.

Function Signature

The “signature” is what you need to know to *call* a function.

Example:

Function Signature

The "signature" is what you need to know to *call* a function.

Example:

`F_to_C(tempF): (returns tempC)`

Function Signature

The "signature" is what you need to know to *call* a function.

Example:

```
F_to_C(tempF): (returns tempC)

Args:
    tempF (float):  The temperature in degrees F

returns:
    float:  The temperature in degrees C
```

temp_mod.py

temp_mod.py

```
def F_to_C(tempF):
    """ This function converts temp in deg F to C
    Args:
        tempf (float):  The temperature in degrees F
    Returns:
        float:  The temperature in degrees C
    """
    return (tempF - 32) * 5 / 9
```

temp_mod.py

```
def F_to_C(tempF):
    """ This function converts temp in deg F to C
    Args:
        tempf (float):  The temperature in degrees F
    Returns:
        float:  The temperature in degrees C
    """
    return (tempF - 32) * 5 / 9

def C_to_F(tempC):
    """ This function converts temp in deg C to F
    Args:
        tempC (float):  The temperature in degrees C
    Returns:
        float:  The temperature in degrees F
    """
    return tempC * 9 / 5 + 32
```

importing temp_mod.py

importing temp_mod.py

```
import temp_mod
```

importing temp_mod.py

```
import temp_mod
degC = temp_mod.F_to_C(50)
```

importing temp_mod.py

```
import temp_mod
degC = temp_mod.F_to_C(50)

from temp_mod import F_to_C, C_to_F
```

importing temp_mod.py

```
import temp_mod
degC = temp_mod.F_to_C(50)

from temp_mod import F_to_C, C_to_F
degC = F_to_C(50)
degF = C_to_F(10)
```

importing temp_mod.py

```
import temp_mod
degC = temp_mod.F_to_C(50)

from temp_mod import F_to_C, C_to_F
degC = F_to_C(50)
degF = C_to_F(10)

from temp_mod import *
```

importing temp_mod.py

```
import temp_mod
degC = temp_mod.F_to_C(50)

from temp_mod import F_to_C, C_to_F
degC = F_to_C(50)
degF = C_to_F(10)

from temp_mod import *
degC = F_to_C(50)
degF = C_to_F(10)
```

Concept Check!

Consider the following function signature:

`F_to_K(tempF): (returns tempK)`
stored in the module/file `kelvins.py` in the library/directory `temps/`

Which of the following are legal ways to import and call this library?

1.
```
import temps
tempK = F_to_K(50)
```
2.
```
import temps
tempK = temps.kelvins.F_to_K(50)
```
3.
```
from temps import kelvins
tempK = kelvins.F_to_K()
```
4.
```
from temps.kelvins import *
tempK = F_to_K(50)
```

Concept Check!

Consider the following function signature:

`F_to_K(tempF): (returns tempK)`
stored in the module/file `kelvins.py` in the library/directory `temps/`

Which of the following are legal ways to import and call this library?

1. ```
   import temps
   tempK = F_to_K(50)
   ```
2. ```
   import temps
   tempK = temps.kelvins.F_to_K(50)
   ```
3. ```
   from temps import kelvins
   tempK = kelvins.F_to_K()
   ```
4. ```
   from temps.kelvins import *
   tempK = F_to_K(50)
   ```

✗

Concept Check!

Consider the following function signature:

`F_to_K(tempF): (returns tempK)`
stored in the module/file `kelvins.py` in the library/directory `temps/`

Which of the following are legal ways to import and call this library?

1. ✗
```
import temps
tempK = F_to_K(50)
```
2. ✓
```
import temps
tempK = temps.kelvins.F_to_K(50)
```
3.
```
from temps import kelvins
tempK = kelvins.F_to_K()
```
4.
```
from temps.kelvins import *
tempK = F_to_K(50)
```

Concept Check!

Consider the following function signature:

`F_to_K(tempF): (returns tempK)`
stored in the module/file `kelvins.py` in the library/directory `temps/`

Which of the following are legal ways to import and call this library?

1. ✗
```
import temps
tempK = F_to_K(50)
```
2. ✓
```
import temps
tempK = temps.kelvins.F_to_K(50)
```
3. ✗
```
from temps import kelvins
tempK = kelvins.F_to_K()
```
4.
```
from temps.kelvins import *
tempK = F_to_K(50)
```

Concept Check!

Consider the following function signature:

`F_to_K(tempF): (returns tempK)`
stored in the module/file `kelvins.py` in the library/directory `temps/`

Which of the following are legal ways to import and call this library?

1. ✗
```
import temps
tempK = F_to_K(50)
```
2. ✓
```
import temps
tempK = temps.kelvins.F_to_K(50)
```
3. ✗
```
from temps import kelvins
tempK = kelvins.F_to_K()
```
4. ✓
```
from temps.kelvins import *
tempK = F_to_K(50)
```