Procedural Abstraction:
Functions

also known as
procedures, methods, subroutines

 Chucking
 Very limited working capacity
 Nearly unlimited complexity

Modular Structure

Brain-size chunks
 Not too many details. Not too many interactions.
 Something to focus on.

Units of work
 Good size for a work assignment. Good size to build
 and test before moving on.

Units of change
 Something that can be replaced

O, be some other name! – *Juliet*

```python
def diff( a, b ):
    return a - b

... *then in main ...*

x = 5
y = 7
c = diff( y, x )

# # # # # # What happens```
O, be some other name! – *Juliet*

```python
def diff ( a, b ) :
    return a – b

... then in main ...
a = 5
b = 7
c = diff( b, a )
    ## ??? What happens
```

```python
def diff ( int a, int b ) :
    return a – b

... then in main ...
a = 5
b = 7
c = diff( b, a )

The called method gets its own copies of the inputs, by position (not by name).
```

“Pass by value”

```python
foo( x, y, 37.489, “e tu, Brute?” )
def foo( a, b, c, d ) :

The “actual arguments” `x, y, 37.489, “et tu, Brute”`
are copied into
the “formal arguments” of the function.
The copies become distinct, local variables.
What does it print?

def foo( x, y):
    x = 17
    y = 19

...  
x = 3  
m = 22  
foo(x, m)  
print “Now ”, x, “ and ”, m

Function input and output

The input to a function is it’s arguments
The output of a function is what it returns

A function may also have an effect
(e.g., sort(ar) has the effect of sorting ar)
If a function prints something, that is an effect of the function, and output of the program.

Documenting functions

def mid(x, y):
    """Return an integer midway between x and y.
    args:
    x: An integer
    y: An integer
    returns
    an integer value approximately midway
    between x and y
    """
    return (x + y) // 2

More realistic examples ... next week

def days_in_month(mm, yyyy):
    returns the number of days in month mm
    (1-12) from of year yyyy (between 1900 and 2500)

is_leap(yyyy):
    return True if yyyy is a leap year
    (otherwise False)
What makes a good function?
Simplifies the code that calls it
Isolates a design decision (easier to change)
Used more than once
Can be tested separately

* A good function may have only some of these properties. Few have all.

---

Bad function smells
Complicated description

If the **simplest** description is “blah blah and blah and blah except blah or blah”, maybe it shouldn’t be a method

Have to keep looking back at it
I should be able to use the function without remembering details of how it works

---

References

\[ a = 42 \]
\[ m = [ 12, 13, 14 ] \]

**Environment:**

\[
\begin{array}{c}
| a | \hline
| 42 | \hline
| m | \hline
| 12 | 13 | 14 |
\end{array}
\]

\[ a = 42 \]
\[ m = [ 12, 13, 14 ] \]
\[ n = [ 12, 13, 14 ] \]

**Environment:**

\[
\begin{array}{c}
| a | \hline
| 42 | \hline
| m | \hline
| 12 | 13 | 14 |
\end{array}
\]

\[
\begin{array}{c}
| m | \hline
| 12 | 13 | 14 |
| n | \hline
| 12 | 13 | 14 |
\end{array}
\]
```python
def clobber(x):
    x[1] = 42
m = [12, 13, 14]
clobber(m)
m = [12, 13, 14]
```
Designing functions

A function can **return a result**
A function can **modify an object**

**Rule of thumb:**

*Do one or the other ... don’t do both*

---

Any other ways for a function to affect other variables?  

*(yes)*

```python
evowels = ['a', 'e', 'i', 'o', 'u']

def messup():
    vowels[1] = 'x'
    return

def main():
    print(vowels)
    messup()
    print(vowels)

main()
```

---

Slightly better:

```python
evowels = ['a', 'e', 'i', 'o', 'u']

def messup():
    **global vowels**
    vowels[1] = 'x'
    return

def main():
    print(vowels)
    messup()
    print(vowels)

main()
```

---

```python
def bar(x):
    x = x * 3
    return x

def foo(x):
    y = bar(x)
    x = y + y
    return x

x = 2
y = foo(x)

(main) x 2
```
def bar(x):
    x = x * 3
    return x

def foo(x):
    y = bar(x)
    x = y + y
    return x

x = 2
y = foo(x)

(main) x 2

def bar(x):
    x = x * 3
    return x

def foo(x):
    y = bar(x)
    x = y + y
    return x

x = 2
y = foo(x)

(main) x 2

(bar) x 2

(bar) x 6

(foo) x 2

(foo) x 2

(main) x 2

(main) x 2
The stack of activation records (frames)

```
def bar(x):
    x = x * 3
    return x

def foo(x):
    y = bar(x)
    x = y + y
    return x

x = 2
y = foo(x)
```

```
def bar(x):
    x = x * 3
    return x

def foo(x):
    y = bar(x)
    x = y + y
    return x

x = 2
y = foo(x)
```
def bar(x):
    x = x * 3
    return x
def foo(x):
    y = bar(x)
    x = y + y
    return x

x = 2
y = foo(x)

Do this yourself ...
Go to http://www.pythontutor.com/visualize.html
(seems to work best in Chrome)

Edit code, watch it execute