Functions as first-class citizens means that we can pass functions around in a program as **values**, not much different than an integer or real value!

When functional languages first appeared in the late 1970’s and the 1980’s this was a radical concept.

Today almost all modern languages support this, e.g.
- Asteroid, Python, JavaScript, Rust, Go
Functions as First-Class Citizens

---

```plaintext
1  -- first-class functions
2
3  function inc with i do
4     return i+1.
5  end
6
7  let foo = inc.  -- foo now holds a function value
8  let x = foo(1).  -- execute the function value with argument 1.
9  assert (x == 2).
```
Python supports functions as first-class citizens

```python
>>> def inc(i):
...     return i+1
...
>>> foo = inc
>>> foo(1)
2
```
Higher-order programming refers to the fact that we take advantage of functions as values in our algorithms.

Note: Higher-Order programming does not refer to applying functional programming to more difficult problems.
One interesting consequence of functions as values is that we can write generic functions whose behavior we can influence by passing in functions.

In the following ‘c’ is a generic function whose behavior we can influence by passing in specific functions.
Generic Functions

```plaintext
1    -- first-class functions
2
3    function inc with i do
4        return i+1.
5    end
6
7    function dec with i do
8        return i-1.
9    end
10
11    -- c expects a function f and a value v and
12    -- returns the value of applying f to v.
13    function c with (f,v) do
14        return f(v).
15    end
16
17    -- we can now modify the behavior of c by
18    -- passing in different functions
19    let x = c(inc,1).
20    assert(x==2).
21
22    let y = c(dec,1).
23    assert(y==0).
```
# first-class functions

# define our increment function
def inc(i):
    return i+1

# define our decrement function
def dec(i):
    return i-1

# c expects a function and a value
def c(f,v):
    return f(v)

# we can modify c's behavior depending what kind of
# function we pass it.
x = c(inc,1)
assert(x == 2)

y = c(dec,1)
assert(y == 0)
Another powerful idea from higher-order programming is the idea of function dispatch tables. Here we store functions in a table indexed by some sort of key. Given a key we retrieve the associated function and execute it.
Function Dispatch Tables

-- program to demonstrate function dispatch tables
load system hash.

-- functions to be put into the dispatch table
function good_morning with name do
  |  return ("Good morning, "+name+"!").
end

function good_afternoon with name do
  |  return ("Good afternoon, "+name+"!").
end

function good_evening with name do
  |  return ("Good evening, "+name+"!").
end

-- create our dispatch table
let myhash = hash @hash()
myhash @insert ("morning", good_morning).
myhash @insert ("afternoon", good_afternoon).
myhash @insert ("evening", good_evening).

-- test out dispatch table
let greeting_function = myhash @get ("morning").
assert(greeting_function ("Joe") == "Good morning, Joe!").
We can do the same thing in Python!

```python
# program to demonstrate function dispatch tables

# functions to be put into the dispatch table
def good_morning(name):
    return ("Good morning, "+name+"!")

def good_afternoon(name):
    return ("Good afternoon, "+name+"!")

def good_evening(name):
    return ("Good evening, "+name+"!")

# create our dispatch table
myhash = dict()
myhash.update({"morning":good_morning})
myhash.update({"afternoon":good_afternoon})
myhash.update({"evening":good_evening})

# test out dispatch table

greeting_function = myhash["morning"]
assert(greeting_function("Joe") == "Good morning, Joe!")
```
The Lambda Function

- The most well-known feature of higher-order programming is the *lambda* function.
- A lambda function is a function definition without a name.
- In functional-style programming this is often used for functions that are so trivial that they don’t warrant a full function definition.

```
Asteroid Version 1.1.3
(c) University of Rhode Island
Type "asteroid -h" for help
Press CTRL-D to exit
[ast> let y = (lambda with x do x+1) 1.
   2
  ast> ]

Python 3.9.6 (default, Sep 13 2022, 22:03:16)
[Clang 14.0.0 (clang-1400.0.29.102)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> y = (lambda x : x+1) (1) ]
[>>> y ]
  2
[>>> ]
```
Lambda functions are values!

The true power of lambda functions only becomes apparent when combined with other higher-order programming features.
The Map Function

- The map function allows you to replace iteration over a list with mapping a function onto the list.
- The map function is a higher-order function since it expects a function as a parameter.
The Map Function

Asteroid

iteration

```haskell
let a = [1,2,3].
let b = [].

-- iterate over the list
for e in a do
  b @append(e+1).
end

assert(b == [2,3,4]).
```

mapping

```haskell
1  -- compute a list whose elements are incremented
2  -- by one compared to the input list
3
4  let a = [1,2,3].
5  let b = [].
6
7  -- using map
8  let b = a @map(lambda with i do i+1).
9
10 assert(b == [2,3,4]).
```
The Map Function

Python

```python
# compute a list whose elements are incremented
# by one compared to the input list

a = [1, 2, 3]
b = []

# iterate over the list
for e in a:
    b.append(e+1)

assert(b == [2, 3, 4])
```

```python
# compute a list whose elements are incremented
# by one compared to the input list

a = [1, 2, 3]
b = []

# using map
b = list(map((lambda x : x+1), a))

assert(b == [2, 3, 4])
```
The Map Function

- One way to think about map is that it applies the given function to each element of the list.

```haskell
[1,2,3] @map(lambda with i do i+1)

[1, 2, 3]
```

```haskell
[lambda with i do i+1] 1,
(lambda with i do i+1) 2,
(lambda with i do i+1) 3
```

```haskell
[2,3,4]
```
The Map Function

- The lists themselves can consist of structured objects – the supplied function must be able to handle the elements of the list as arguments.
- The return value of the function being mapped can be different from its input values.

```haskell
-- applying map to a list of tuples
let l = [(1,2),(3,4),(5,6)] @map(lambda with (x,y) do x+y).
assert(l == [3,7,11]).
```
The Map Function

- Map is not restricted to lambda functions
- You can map any appropriate function onto a list.

Advantage of this approach
- No iteration
- A quick way to transform a list
The Map Function

-- show that map will map any function onto a list
-- here we map a greeting onto a list of names
-- the result is a list of greetings

let names = ["Joe","Bridget","Peter"].

function greeting with name do
  return "Hello "+name+"!".
end

let greetings = names @map greeting.
assert(greetings == ["Hello Joe!","Hello Bridget!","Hello Peter!"]).
The Map Function

# show that map will map any function onto a list
# here we map a greeting onto a list of names
# the result is a list of greetings
	names = ["Joe","Bridget","Peter"]

def greeting(name):
    return "Hello " + name + "!

greetings = list(map(greeting, names))
assert(greetings == ["Hello Joe!","Hello Bridget!","Hello Peter!"])
Class Exercise

- Given the Asteroid program on the right do the following:
  - Rewrite inc_list as a recursive function using multi-dispatch.
  - Rewrite inc_list as a function that utilizes the list ‘@map’ function to accomplish the computation.
  - Demonstrate that your functions work.

```groovy
function inc_list with input_list do
  let output_list = [].
  for e in input_list do
    output_list @append(e+1).
  end
  return output_list.
end

let l = [1,2,3].
let new_list = inc_list(l).
assert(new_list == [2,3,4]).
```

https://replit.com/@LutzHamel1/asteroid-csc301-classexercise-inclist#examples/inclist.ast
Assignment

Assignment #4 – See BrightSpace