15-110 Principles of Computing – F19

Lecture 16: Functions 1

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Functions: callable, named subprograms (procedures)

- **Function:** informally, a *subprogram,* we write a sequence of statements and give that sequence a name. The instructions can then be executed at any point in a program by referring to the function name

```
def function_name(arguments):
    function_body
    return something
```

User function definition

Calling (invoking) the function

```
def happy():
    print("Happy birthday to you!")

name = 'Fred'
print('Hello ' + name + '!')
happy()
happy()
happy()
```

Built-in function (python standard library)

All functions return something

```
def function_name(arguments):
    function_body
```

is implemented as:

```
def function_name(arguments):
    function_body
    return None
```

All function calls return something, `None` when left unspecified in the function body

Built-in functions (Python standard library)

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

https://docs.python.org/3/library/functions.html

Categories of functions

Returning values or not

- Functions that do something and do *not* return anything (return `None`)
- Functions that do something and do return something different than `None`

Input arguments or not

- Functions that do something based on input parameters
  - Functions that also change the value of the input parameters
  - Functions that do not change the value of the input parameters
- Functions that do something *not* based on input parameters

Functions: organizing the code, putting aside functionalities

- Functions are a fundamental way to *organize* the code into **procedural elements** that can be *reused*
- Functions provide **structure and organization,** that facilitate:

Decomposition: example

Goal: design a simple *lottery system*, where the user inputs three integer numbers and the system checks if she/he has won the lottery or not!


```
value_list = []
n = 0
while n < 3:
    input_str = input('Input an integer number: ')
    if input_str.isdigit():
        v = int(input_str)
        value_list.append(v)
        n += 1
prod = 1
for v in value_list:
    prod *= v
if not (prod % 13):
    print("You won!!!")
else:
    print("You didn't win...")
```

Decomposition: example

get three numbers from user inputs and put them in a list

make the product of the numbers in the list

check the winning condition and output the message

```
value_list = []
n = 0
while n < 3:
    input_str = input('Input an integer number: ')
    if input_str.isdigit():
        v = int(input_str)
        value_list.append(v)
        n += 1
prod = 1
for v in value_list:
    prod *= v
if not (prod % 13):
    print("You won!!!")
else:
    print("You didn't win...")
```

Decomposition: example

- ✓ It can be observed that the overall task is (naturally) **composed by at least three independent *sub-tasks***
- ✓ In fact, the defined computation is composed by three groups of statements each aimed at realizing a specific sub-task
  → These three groups of statements can be conveniently *isolated*, organized into *different functions*, and then *merged* to obtain the same overall result
- ✓ Each function corresponds to the creation of a **primitive** for the task, that can be used in the computation

Decomposition: example

```
def get_user_numbers():
    value_list = []
    n = 0
    while n < 3:
        input_str = input('Input an integer number: ')
        if input_str.isdigit():
            v = int(input_str)
            value_list.append(v)
            n += 1
    return value_list

def make_product(values):
    prod = 1
    for v in values:
        prod *= v
    return prod

def check_lottery(input_val):
    if not (input_val % 13):
        print("You won!!!")
    else:
        print("You didn't win...")
```

```
numbers = get_user_numbers()
lottery_number = make_product(numbers)
check_lottery(lottery_number)
```

Decomposition: example

```
def get_user_numbers():
    value_list = []
    n = 0
    while n < 3:
        input_str = input('Input an integer number: ')
        if input_str.isdigit():
            v = int(input_str)
            value_list.append(v)
            n += 1
    return value_list

def make_product(values):
    prod = 1
    for v in values:
        prod *= v
    return prod

def check_lottery(input_val):
    if not (input_val % 13):
        print("You won!!!")
    else:
        print("You didn't win...")
```

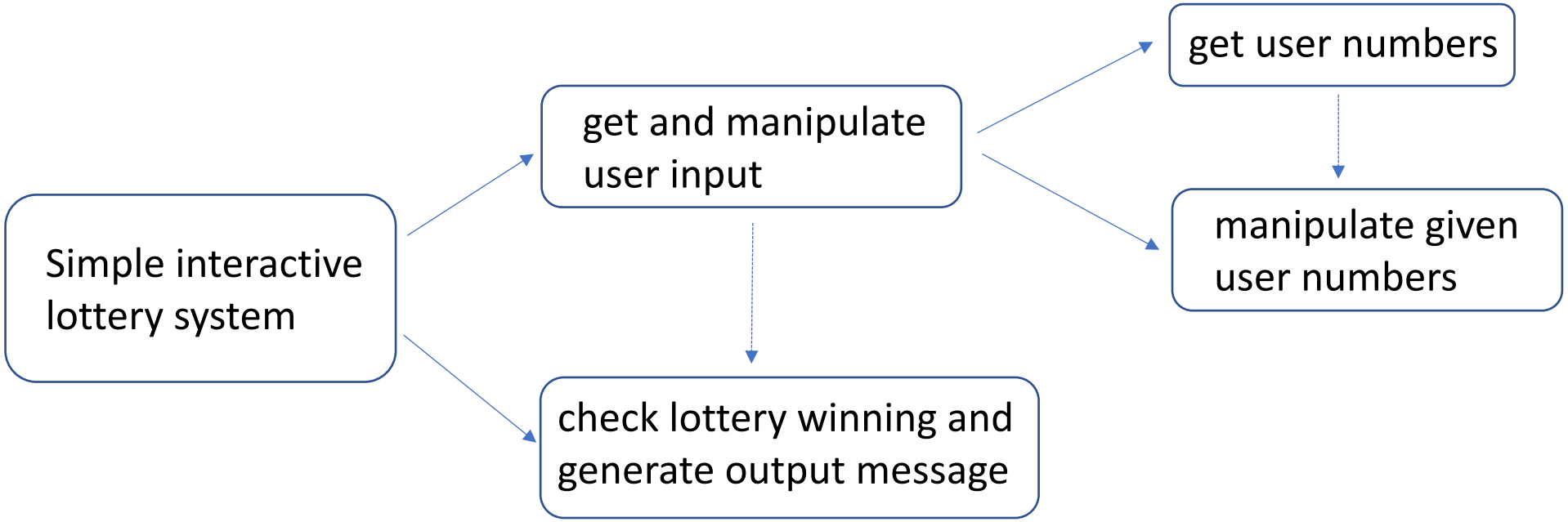
```
numbers = get_user_numbers()
lottery_number = make_product(numbers)
check_lottery(lottery_number)
```

- Even better:

```
def get_lottery_number():
    values = get_user_numbers()
    number = make_product(values)
    return number

lottery_number = get_lottery_number()
check_lottery(lottery_number)
```

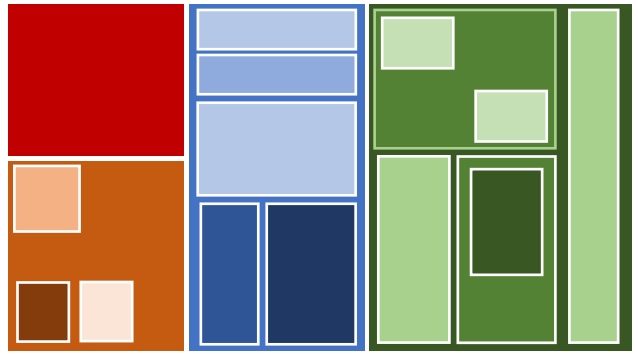
Decomposition using functions



Problem to solve (monolithic view)

Decompose in multiple (smaller / easier) sub-problems, possibly nested, and then merge the sub-problems

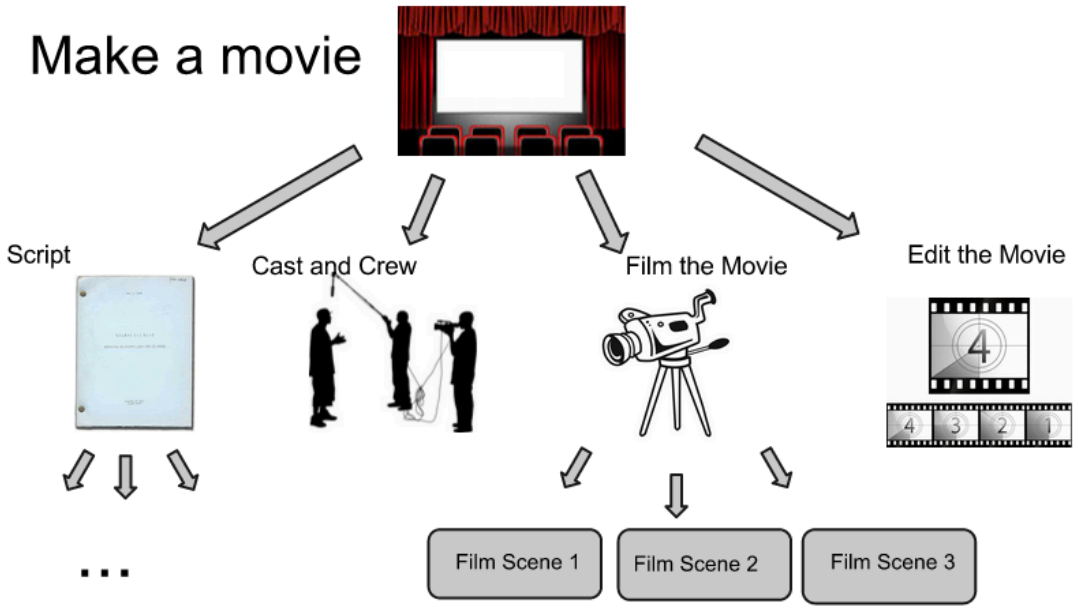
Divide-and-conquer

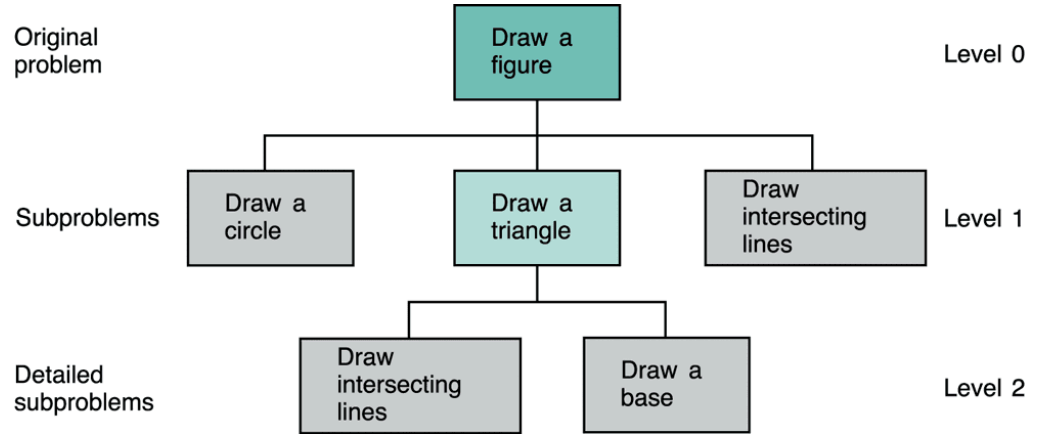


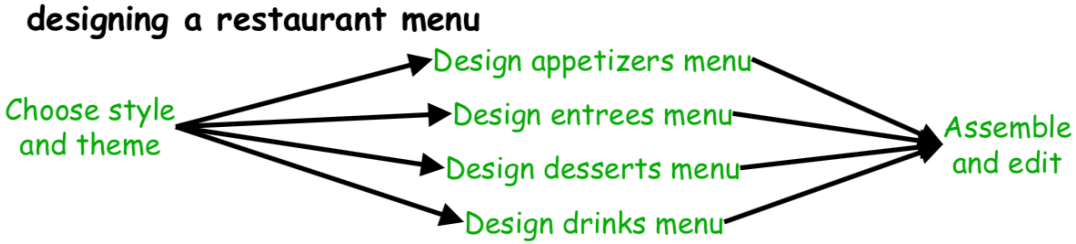
Gains from decomposition using functions

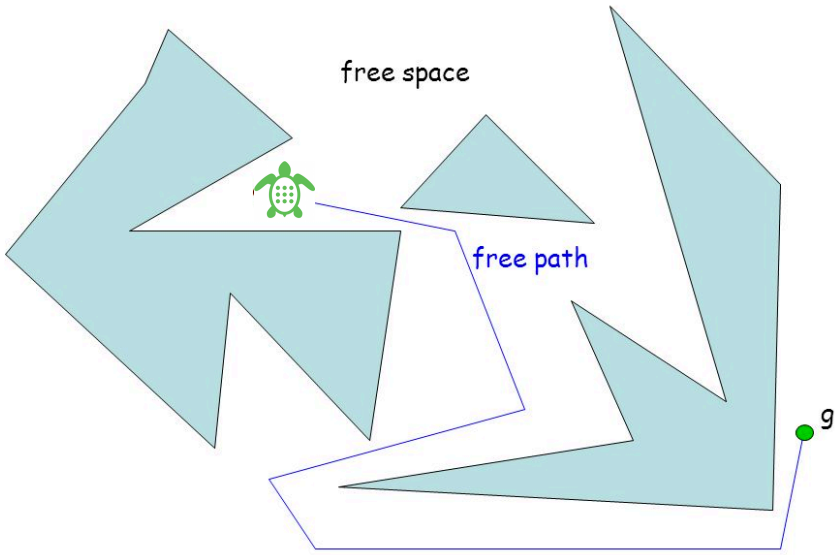
- Gaines resulting from decomposing the problem in functional blocks and defining functions / primitives to handle the computation in each block:
  - ✓ A better **understanding of the task** and of **our computational approach** to its solution
  - ✓ **Readability**: we can now *read* what's going on (using self-explanatory names for the functions)
  - ✓ **Easiness of testing**: each function can be tested for correctness / performance in standalone
  - ✓ **Easiness of maintenance and updating**: we can improve / change the code of each new primitive function independently from other program parts (as long as inputs and return types stay the same)
  - ✓ We can now even **split the job** to design the individual functions to multiple programmers*, in parallel!*

Examples of problem decompositions



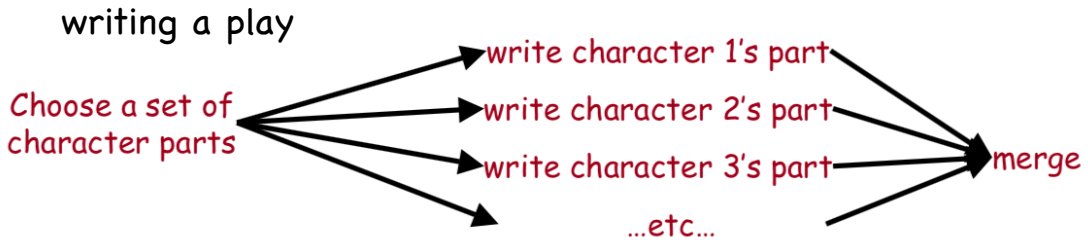






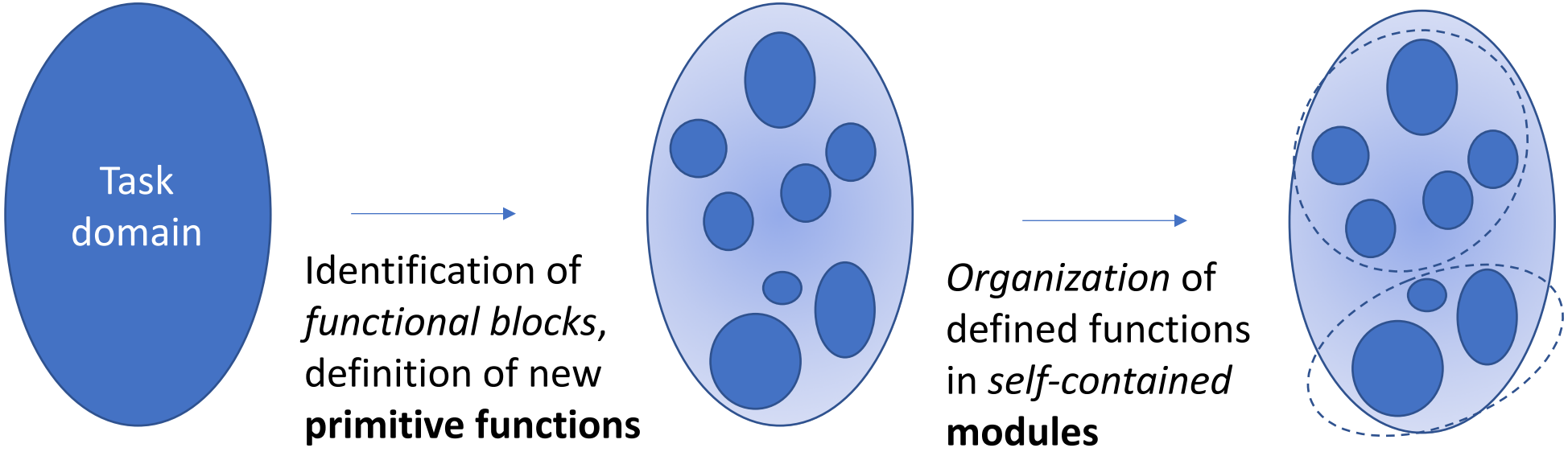
rotate in place $+110^\circ$
move forward 5m
rotate in place + $+80^\circ$
move forward 6m
rotate in place $+95^\circ$
move forward 7.5m
rotate in place -95°
move forward 2.5m
rotate in place -105°
move forward 9m
rotate in place -85°
move forward 2.8m

Not always fully feasible or straightforward!



Decomposition by creation of self-contained modules

- Decomposition can be realized at different granularity, depending on the task domain



A function is a module with one single component

Each module provides a set of (related) functionalities:

- ✓ **Reusability** in different tasks or different parts of the program
- ✓ High-level of **abstraction**

Reusability

- **Code reusability:** In addition to convenient code decomposition in functional blocks, the use of functions is essential to avoid to repeat the same fragment of code over and over in the same program, as well as to allow to reuse the same functional blocks in different programs

- E.g, we want to write a program that prints out the lyrics to the “Happy Birthday” song:

  Happy birthday to you!
  Happy birthday to you!
  Happy birthday, dear <insert-name>. Happy birthday to you!

- Without functions we would write something like the following code to wish happy birthday to Fred:

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear", 'Fred.')
print("Happy birthday to you!")
```

- If we want to wish happy birthday to Lucy too, now we have to duplicate the code … ☹

Reusability

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear", 'Fred.')
print("Happy birthday to you!")
```

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear", 'Lucy.')
print("Happy birthday to you!")
```

We don't want all this duplicated code!

```
  def happy():
      print("Happy birthday to you!")

  def sing(person):
      happy()
      happy()
      print "Happy Birthday, dear", person + "."
      happy()
```

- ✓ Now `sing(person)` can be reused for wishing happy birthday to *any* person!
- ✓ `sing()` and `happy()` could be included in a new `birthday` *module* …