Modularization

- So far we've seen key essential concepts of programming: input, processing, output, sequence, selection & repetition
- it is impossible to write any program without these

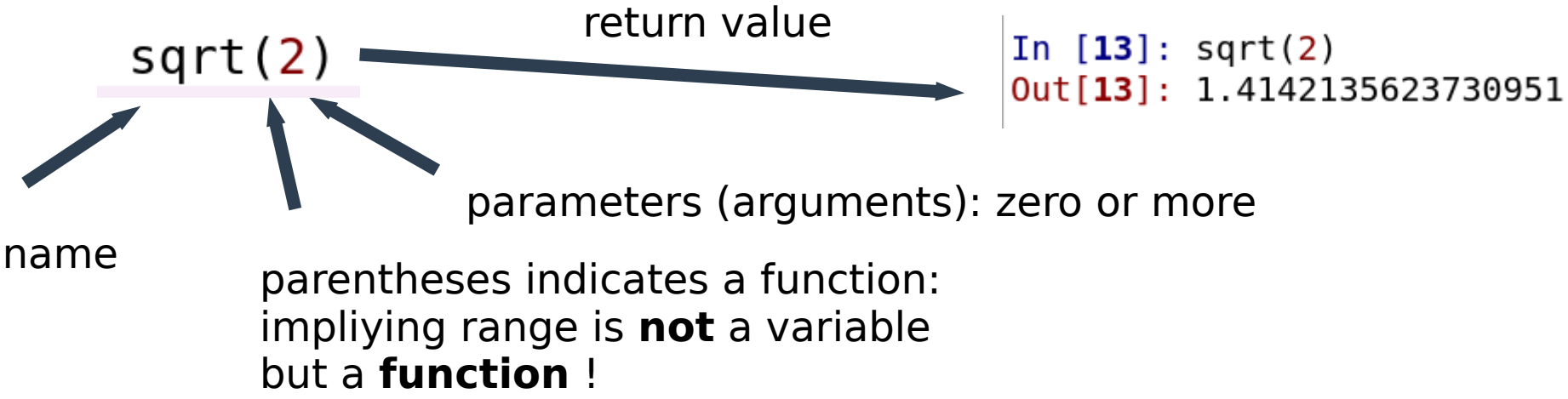
- **modularization** is another key concept
- although its optional, it is indeed essential
- examples:
  - **functions**, **classes**, and **modules**

- modularization is important because:
  - **reduces complexity**
  - enables **creation of reusable fragments of code**

- Reducing complexity is important:
  - divide a big programming task into smaller, more manageable pieces
  - carefully created modules can be reusable
  - human brain can't focus on more than a handful things at a time

Functions

- We've already used many built-in functions in python:
- range(), math.sqrt(), random.randint()



- return value can be **None**
- there can be as few as zero number of arguments, **still () are required!**

Writing functions: is_even()

Write a function by starting with how we want to use it.

Imagine that we're working on a game, and wish we could write:

```
num = random.randint(0,100)
if is_even(num):
    print("Good news your magic number is even!\
    10 bonus points for you.")
else:
    print("Bad news, your magic number is odd.")
```

specifying the use of our function before writing it is that we now know,

- name of the function should be **is_even()**
- it takes only one parameter, **num**
- returns a Boolean value, **True** or **False**

From previous experience, we know how to implement a piece of code determining if a number is even or odd

```
if n%2 == 0:
    # It's even
else:
    # It's odd
```

Writing functions: is_even()

- Keyword **def** marks the beginning of a function,

- followed by function name, paranthesis, parameters, and **“:”**

- code to be executed is indented

- **return** statement terminates the execution of function

- more than one **return** is possible

```
import random

def is_even(n):
    if n%2 == 0:
        return True
    else:
        return False

num = random.randint(0,100)
if is_even(num):
    print("Good news your magic number is even!\
    10 bonus points for you.")
else:
    print("Bad news, your magic number is odd.")
```

- the names of the parameter in the program and in the function don’t need to be the same:
  num in the program is referred to as **n** in the function definition!

- What is the point? why not replace “if is_even(num):” with “if num% == 0:”?

Writing functions: is_even()

- What is the point? why not replace
“if is_even(num):” with “if num% == 0:”?

- function is reusable by other programs!

- if name of the parameter in func definition and
in the program were required to be the same,
it would be very hard to reuse that function!

- **n** is an alias to value of **num**


```
import random

def is_even(n):
    if n%2 == 0:
        return True
    else:
        return False

num = random.randint(0,100)
if is_even(num):
    print("Good news your mag
    10 bonus points for you.'
else:
    print("Bad news, your mag
```

- as soon as function is terminated, any memory it used is freed

- hence variables, constants, etc in a function definition are local to that function!

Writing functions: Syntax Summary

```
def function-name( parameter-list ):
    ...
    ... function-body
    ...
    return expression
    ...
    return expression
```

- it is preferable to have a single **return** expression or value,

- and it is preferable to have it at the last line of the function

- if no **return** is included, function will return **None**

‘

- it is preferable to have a blank **return** than having none.
    (showing our intention of returning None as opposed to forgetting to write a return expression)

Example: is_odd()

```
def is_odd(n):
    if n%2 != 0:
        return True
    else:
        return False
```

is there a better way?

Example: is_odd()

```
def is_odd(n):
    return not is_even(n)
```

```
def is_odd(n):
    if n%2 != 0:
        return True
    else:
        return False
```

- less lines of code

- reuse

- if someday a better way of determining is_even() is found

    - only one function needs to rewritten

    - improvements in is_even() automatically passed on to is_odd():
      this case is trivial,
      but in many computationally complex programs, this happens all the time!

Programming Exercise: Dice_roll()

Write a program rolling a pair of dice and returning the total.
Modularize dice roll section of the code via use of a function like:

```
total = dice_roll() + dice_roll()
```

Programming Exercise: Dice_roll()

```
import random

def dice_roll():
    return random.randint(1,6)

total = dice_roll() + dice_roll()
print("On your first roll you got:", total)
```

Programming Exercise: Dice_roll()

```
import random

def dice_roll():
    return random.randint(1,6)

total = dice_roll() + dice_roll()
print("On your first roll you got:", total)
```

What if there is more than 6 sides? Can we generalize this?

Programming Exercise: Dice_roll()

```
import random

def dice_roll(sides):
    return random.randint(1,sides)

total = dice_roll(6) + dice_roll(6)
print("On your first roll you got:", total)
```

Default arguments

```
import random

def dice_roll(sides):
    return random.randint(1,sides)

total = dice_roll(6) + dice_roll(6)
print("On your first roll you got:", total)
```

- we improved dice rolling function via generalizing it to more than 6 sides

- however for the most common case, 6 sides, the usage got a bit complicated because now we have to specify number of sides!

- To avoid this complication, we can specify default values of arguments:
  default value: a value to be used when no value is given/specified.

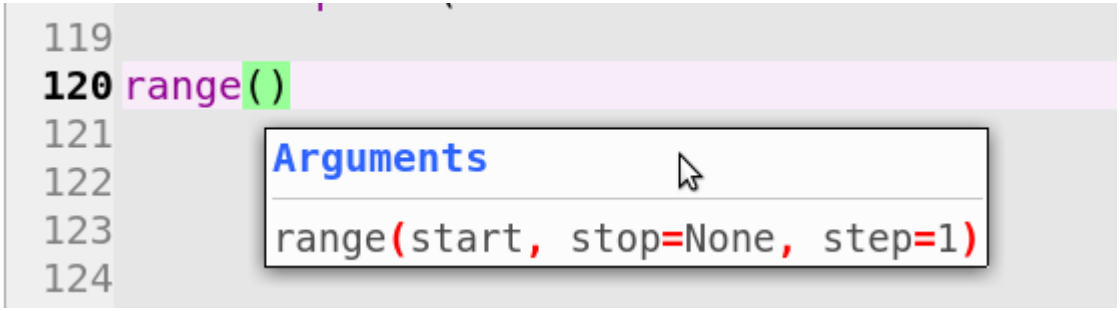
Default arguments

```
import random

def dice_roll(sides = 6):
    return random.randint(1,sides)

print('Your results are:')
print('6-sided die:', dice_roll())
print('24-sided die:', dice_roll(24))
```

yields:

```
Your results are:
6-sided die: 1
24-sided die: 15
```

default argument values are very common in Python



Example: playing card functions

Let’s use functions (and later on classes) to save time when working with playing cards.

Create functions for common tasks: getting suit & face value from card number

```
SUITS = ('Clubs', 'Diamonds', 'Hearts', 'Spades')
FACE_VALUES = ('Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
               'Seven', 'Eight', 'Nine', 'Ten', 'Jack',
               'Queen', 'King')

def suit(cardnum):
    return SUITS[cardnum // 13]

def face_value(cardnum):
    return FACE_VALUES[cardnum % 13]

card = 15
print("Card", card, "is the", face_value(card), "of", suit(card))
```

Example: playing card functions

We will also need to display card lable very often (“Three of diamonds” etc)
So, let’s create a function for that too

```
SUITS = ('Clubs', 'Diamonds', 'Hearts', 'Spades')
FACE_VALUES = ('Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
               'Seven', 'Eight', 'Nine', 'Ten', 'Jack',
               'Queen', 'King')

def suit(cardnum):
    return SUITS[cardnum // 13]

def face_value(cardnum):
    return FACE_VALUES[cardnum % 13]

def label(cardnum):
    return face_value(cardnum) + " of " + suit(cardnum)

card = 15
print("Card", card, "is the", face_value(card), "of", suit(card))
print("Card", card, "is the", label(card))
```

output is:

```
Card 15 is the Three of Diamonds
Card 15 is the Three of Diamonds
```

Reusing functions

Now that we've written some reusable functions dealing with playing cards, how do we go about reusing them?

- imagine these functions are needed in a mobile app, and in a webapp

- Option #1: copy & paste these functions to mobile-app and webapp programs

- What can go wrong?

Reusing functions

Now that we've written some reusable functions dealing with playing cards, how do we go about reusing them?

- imagine these functions are needed in a mobile app, and in a webapp

- Option #1: copy & paste these functions to mobile-app and webapp programs

    - improve these functions, **re-**copy & paste them to mobileapp, webapp,

    - a bug found and fixed. Where was that up-to-date list of all programs using these functions?

    - it would be nice if there was a way to automate this process, so that programs always benefit from improvements to these functions.

    What would that be?

Reusing functions

Now that we've written some reusable functions dealing with playing cards, how do we go about reusing them?

- Programmers found that the best way to reuse code is to include the contents of one file in another

- Python does this by using **modules** and the **import** command

- We've already used **import** to use functions from **math** and **random** modules

- Now let's see how we can do this for our own functions

Modules

- A python program is a module
- and other programs can use the functions, variables, etc in it
- but we need to follow some standard practices for the code to be reused easily
- Lets put playing card functions in **playing_cards.py**
-
- our new program **blackjack.py** have a hand of card numbers
  - we would like to display the cards in the hand:

```
# blackjack.py
...
print 'You are holding,'
for card in hand:
    print 'The', label(card)
...
```

expected output

```
You are holding,
The Three of Diamonds
The Four of Spades
The Nine of Clubs
```

Modules: First Try

```
# blackjack.py
import playing_cards

hand = [15, 42, 8]
print('You are holding,')
for card in hand:
    print('The', playing_cards.label(card))
```

Note, when importing we omit *.py extension

just as we didn’t do “math.py.sqrt” but **math.sqrt**

What do you think we should get? [demo]

Modules: A problem

```
>>>
Card 15 is the Three of Diamonds
Card 15 is the Three of Diamonds
You are holding,
The Three of Diamonds
The Four of Spades
The Nine of Clubs
>>>
```

- we are getting extraneous output from **playing_cards.py**
- modules are run upon import!
- Code is interpreted and definitions in the module become available to current program
- Should we eliminate print statements? What if they're necessary?

Modules: A solution

- Every module has a **__name__** attribute
- When run on its own **__name__** is assigned the name **__main__**
- When imported, **__name__** is assigned the name of the module: **playing_cards**
- Lets run the following, and see what we get [demo]

```
14
15 def suit(cardnum):
16     return SUITS[cardnum // 13]
17
18 def face_value(cardnum):
19     return FACE_VALUES[cardnum % 13]
20
21 def label(cardnum):
22     return face_value(cardnum) + " of " + suit(cardnum)
23
24 print('My name is', __name__) # Line 24!
25 card = 15
26 print("Card", card, "is the", face_value(card), "of", suit(card))
27 print("Card", card, "is the", label(card))
```

Modules: A solution

When we run **playing_cards.py**, we get

```
My name is __main__
Card 15 is the Three of Diamonds
Card 15 is the Three of Diamonds
```

When we run **blackjack.py**, we get

```
My name is playing_cards
Card 15 is the Three of Diamonds
Card 15 is the Three of Diamonds
You are holding,
The Three of Diamonds
The Four of Spades
The Nine of Clubs
```

Modules: A solution

When we run **playing_cards.py**, we get

```
My name is __main__
Card 15 is the Three of Diamonds
Card 15 is the Three of Diamonds
```

When we run **blackjack.py**, we get

```
My name is playing_cards
Card 15 is the Three of Diamonds
Card 15 is the Three of Diamonds
You are holding,
The Three of Diamonds
The Four of Spades
The Nine of Clubs
```

Solution:

put an **if** statement:

only if __name__ == “__main__”

then print stuff

Hence, when imported by a program

we won’t see these extra lines!

Modules: A solution

```
SUITS = ('Clubs', 'Diamonds', 'Hearts', 'Spades')
FACE_VALUES = ('Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
               'Seven', 'Eight', 'Nine', 'Ten', 'Jack',
               'Queen', 'King')

def suit(cardnum):
    return SUITS[cardnum // 13]

def face_value(cardnum):
    return FACE_VALUES[cardnum % 13]

def label(cardnum):
    return face_value(cardnum) + " of " + suit(cardnum)

if __name__ == '__main__':
    card = 15
    print("Card", card, "is the", face_value(card), "of", suit(card))
    print("Card", card, "is the", label(card))
```

Documenting modules and functions

So far we did:
- block of header comments identifying module, programmer, date etc
- inline comments to explain tricky points

However, modules and functions require further documentation
for which primary mechanism is docstrings, i.e. the triple quote text

```
# module_docn.py
'''This is the module documentation pointing out that
this is an artificial test module.'''

def test_fn_1():
    '''This is the first test function.
    It doesn't do anything.'''
    return

def test_fn_2():
    '''This is the second test function which also does nothing.'''
    return
```

Documenting modules and functions

Python has builtin commands that extract this documentation from module

```
In [4]: import module_docn

In [5]: dir(module_docn)
Out[5]:
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'test_fn_1',
 'test_fn_2']
```

```
In [6]: print(module_docn.__doc__)
This is the module documentation pointing out that
this is an artificial test module.

In [7]: print(module_docn.test_fn_1.__doc__)
This is the first test function.
    It doesn't do anything.

In [8]: print(module_docn.test_fn_2.__doc__)
This is the second test function which also does nothing.
```

docstrings: strings that get assigned to the __doc__ attributes

Documenting modules and functions

An easy way to access docstrings is to use **help()**

```
In [10]: help(module_docn)
Help on module module_docn:

NAME
    module_docn

DESCRIPTION
    This is the module documentation pointing out that
    this is an artificial test module.

FUNCTIONS
    test_fn_1()
        This is the first test function.
        It doesn't do anything.

    test_fn_2()
        This is the second test function which also does nothing.

FILE
    /home/sbulut/github/cpsc128/code/python3/module_docn.py
```

Summary: Module layout

```
# filename.py
# ...
''' Module docstring '''

# import statements
import ...
import ...
...

# Function definitions
def name( args ):
    ''' Function docstring '''
    ...

def name( args ):
    ''' Function docstring '''
    ...

...

if __name__ == '__main__':
    # Print statements, simple examples etc
    ...
```

Demo: playing_cards_2

- See playing_cards_2.py

- run it, and observe the output

- also import it

- also look at help() page

```
In [16]: runfile('/home/sbulut/github/cpsc128/code/python3/
playing_cards_2.py', wdir='/home/sbulut/github/cpsc128/code/python3')
New deck: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51]
Shuffled deck: [51, 7, 43, 34, 2, 39, 14, 8, 40, 42, 45, 15, 35, 48, 13,
17, 19, 36, 26, 31, 38, 16, 21, 22, 30, 3, 41, 27, 0, 24, 11, 23, 32, 12,
46, 6, 20, 37, 47, 5, 50, 9, 44, 18, 49, 10, 29, 4, 33, 25, 1, 28]
Dealing a card...
    Card number: 28
    Face value: Three
    Suit: Hearts
    Description: Three of Hearts
Deck after dealing: [51, 7, 43, 34, 2, 39, 14, 8, 40, 42, 45, 15, 35, 48,
13, 17, 19, 36, 26, 31, 38, 16, 21, 22, 30, 3, 41, 27, 0, 24, 11, 23, 32,
12, 46, 6, 20, 37, 47, 5, 50, 9, 44, 18, 49, 10, 29, 4, 33, 25, 1]
```

import tricks

```
card=10

import playing_cards
print(playing_cards.label(card))

import playing_cards as pc
print(pc.label(card))

from playing_cards import label
print(label(card))

from playing_cards import *
print(label(card))

from playing_cards import label as card_name
print(card_name(card))
```

Scope of variables (a technical issue)

Let’s analyze this snippett

```
...
SUITS = ('Clubs', 'Diamonds', 'Hearts', 'Spades')
...

def suit(cardnum):
    '''Returns the suit name, e.g. 'Clubs', of the card it is passed.'''
    return SUITS[cardnum // 13]

def suit(cardnum):
    SUITS = ('Spades', 'Hearts', 'Diamonds', 'Clubs' )
    '''Returns the suit name, e.g. 'Clubs', of the card it is passed.'''
    return SUITS[cardnum // 13]
```

checkout demo:

scope_eg.py

Python functions follow a clear process in trying to resolve names.
- Python first looks in the **Local** context, i.e. the current function.
- Then it looks in any **Enclosing** functions, i.e. functions that contain the current one.
- Then it looks in the **Global** context, i.e. the module.
- Finally it checks the **Built-in** names for a match.

Be careful, when local definitions are overriding globals, or globals overriding built-in ones

Perils of mutability (a technical issue)

```
# increment.py
def increment(n):
    n = n + 1

num = 48
increment( num )
print(num)
```

```
def increment(seq):
    seq.append(42)

lst = [48]
increment( lst )
print(lst)
```

What is the output?

Perils of mutability (a technical issue)

Unlike numbers and strings, lists are mutable so they can be affected when passed to functions, e.g.

```
l = [ 'Tim' ]
n = l
l.append( 'Joyce' )
print(l)
print(n)
```

```
a = 'Tim'
b = 'Tom'
lst = [b, a]
a = 'Matt'
lst[1] = lst[0]
print(lst)
```

Don’t get tripped by mutability!

Copying a list: deep vs shallow

For a shallow copy: do

```
In [63]: l1 = [1, 2]

In [64]: l2 = list(l1) ; l3=l2[:]

In [65]: l2[0] = 3 ; l3[1] = 4

In [66]: print(l1); print(l2) ; print(l3)
[1, 2]
[3, 2]
[1, 4]
```

Shallow copy sort of works for list of lists

```
In [97]: g =[ [1,2], 3]

In [98]: c = list(g)

In [99]: c.append('hello')

In [100]: print(g); print(c)
[[1, 2], 3]
[[1, 2], 3, 'hello']

In [101]: c[0][0] = 10

In [102]: print(g); print(c)
[[10, 2], 3]
[[10, 2], 3, 'hello']
```

Copying a list: deep vs shallow

Solution: use **deepcopy** from **copy** module

```
In [112]: import copy

In [113]: g = [ [1,2], 3]

In [114]: dc = copy.deepcopy(g)

In [115]: dc[0][0] = 'x'

In [116]: print(g)
[[1, 2], 3]

In [117]: print(dc)
[['x', 2], 3]
```