Chapter 5.
Functions

Starting out with Python

Code Examples with Jupyter Lab

Kyu Lee. Ph. D.

Computer Science, Diablo Valley College

Starting out with Python

Navigator

- Local Variables
- Passing Arguments to Functions
- __main__
- Passing Arguments : call-by-reference
- Keyword Arguments
- Global Variables
- Returning multiple variables
- *args, **kwargs
- More Labs
- Yield
- lambda
- Assignments

- Lab 1
- Lab 2
- Lab 3
- Lab 4
- Lab 5
- Lab 6
- Lab 7
- Lab 8
- Lab 9
- Lab 10
- Lab 11
- Lab 12
- Lab 13
- Lab 14

- Lab 15
- Lab 16
- Lab 17
- Lab 18

click the subtitle to go inside

Local Variables

go to navigator

First Example of Function

- Defining and Calling Function

Calling

```
functionname( ):
```

defining a function

```
def functionname( ):
    statements
    statements
    return
```

all variables in this function are local

First Example of Function

- Make a function to take a user input and return it

Version 1

```
def      getinput():
        val = int(input())
        return val

userval = getinput()
print (userval)
```

Version 2

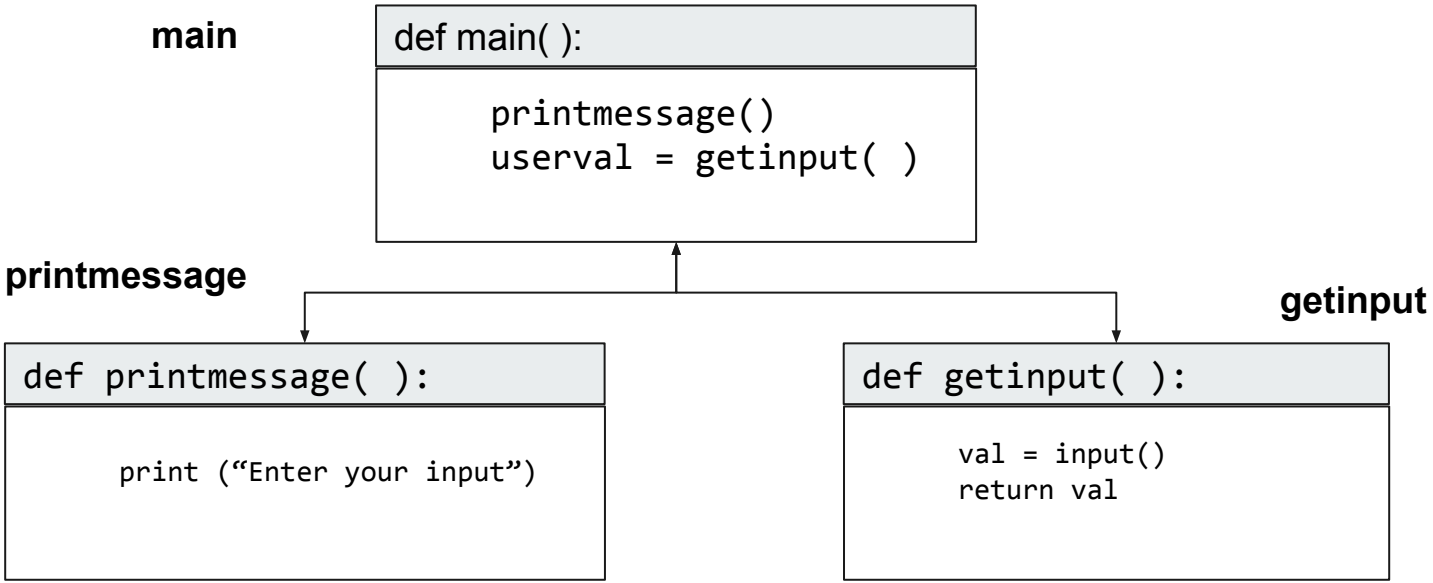
```
def      getinput():
        val = int(input())
        return val

def      main():
        userval = getinput()
        print (userval)


if    name    == '__main__':
        main()
```

First Example of Function

- Functional Diagram

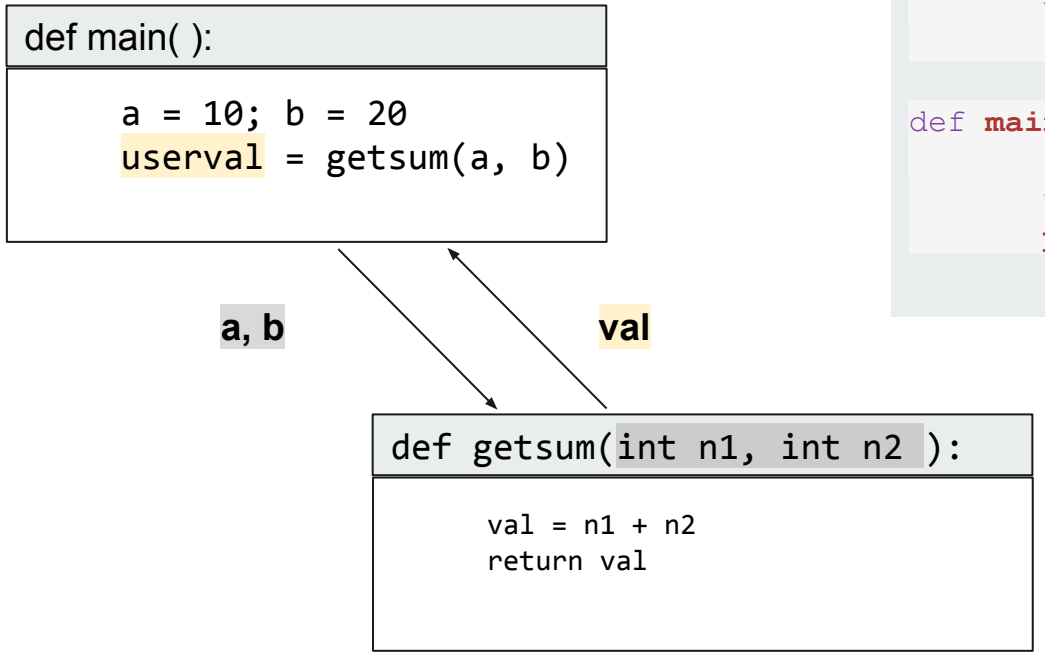


Passing arguments to functions

go to navigator

Passing arguments to Functions

- Functional Diagram

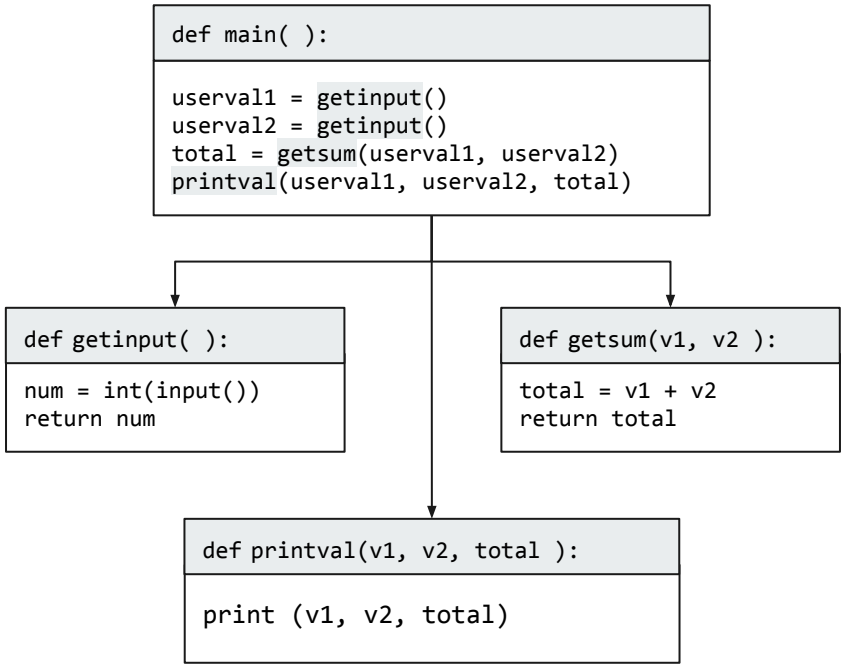



```
def getsum(n1, n2):
        val = n1 + n2
        return val

def main():
        a = 10; b = 20
        userval = getsum(a, b)
        print (userval)
```

Lab 1: Passing arguments to Functions

- Make function as following:
  - getinput( )
    - returns the user input
  - getsum(userval1, userval2)
    - returns the summation of a and b
  - printval(val)
    - prints the value
  - main()
    - call getinput() , getsum(), print()



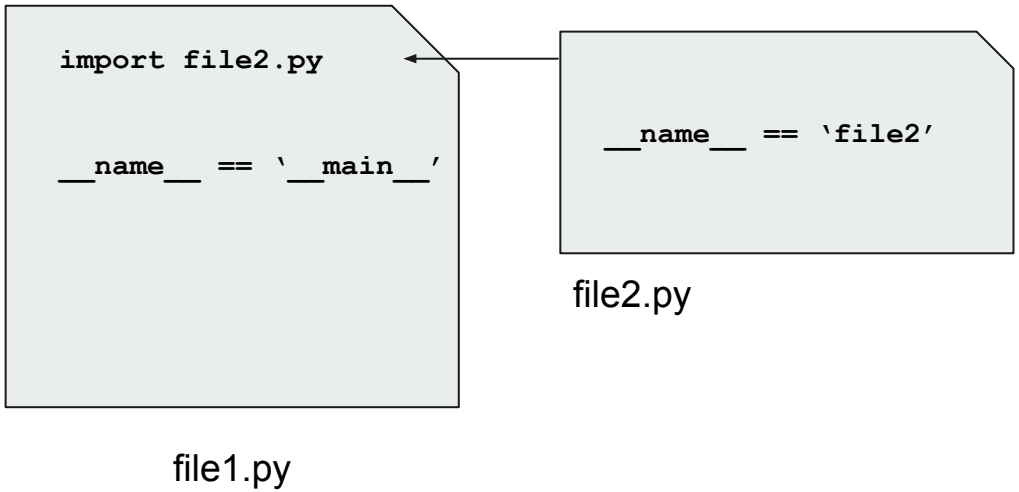
go to navigator

__main__

go to navigator

__main__

- When the file1.py executed directly
  - __name__ is '__main__'
- Otherwise
  - __name__ is 'filename' without extension

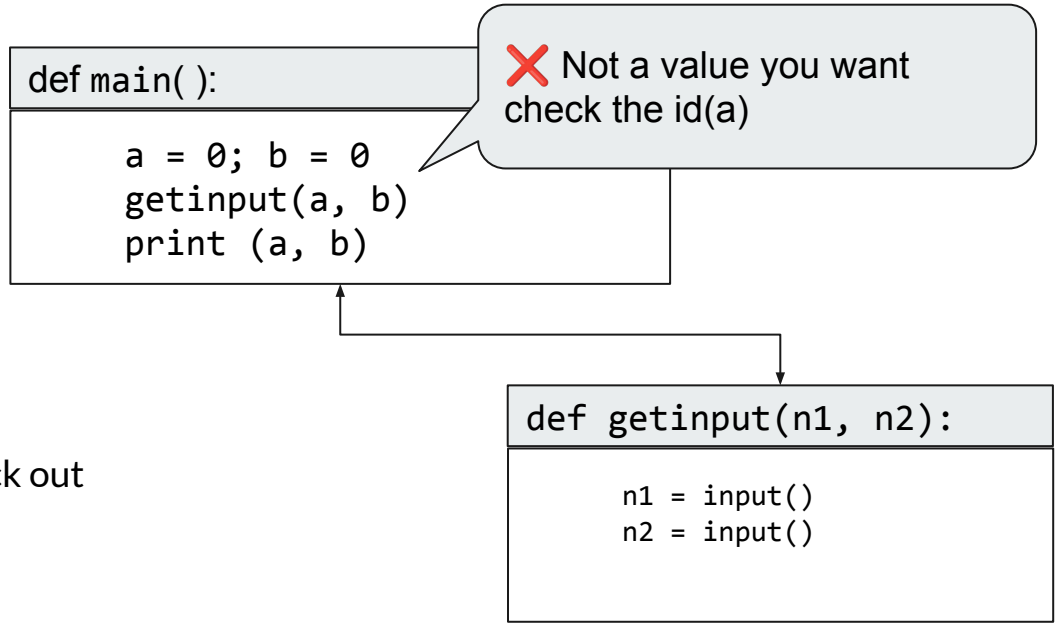


Passing arguments:
making changes to Parameters

go to navigator

Lab 2: Making Changes to Parameters 1

- Argument value sent to the function
  - In the function,
  - change the parameter value
- In caller function,
  - print the arguments



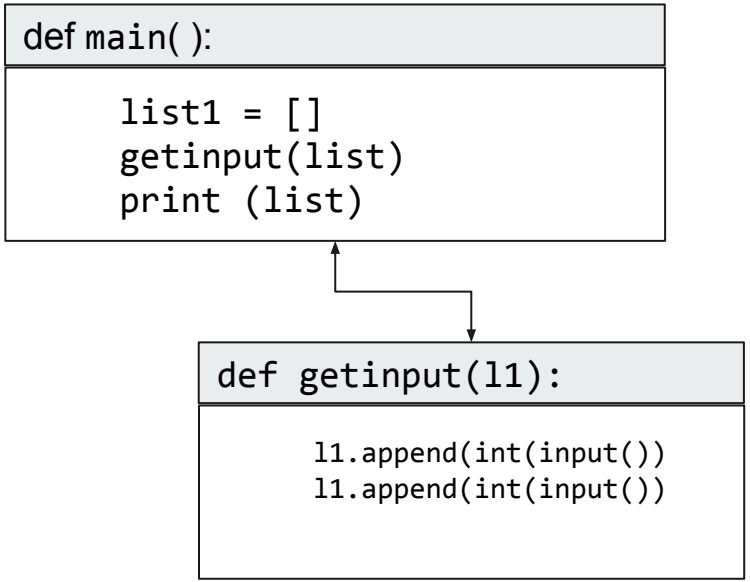
Try to make this example and check out the printing value in main()

go to navigator

Making Changes to Parameters 2

- When an **argument** is **list**

Making changes to parameter
reflects to the arguments in function main()



Making Changes to Parameters 3

- Depends on the **mutable** or **immutable** objects
  - Call-by-**Value** or
  - Call-by-Object Reference

Mutable objects:

list, dict, set, byte array

Immutable objects:

int, float, complex, string, tuple, frozen set [note: immutable version of set], bytes.

Making Changes to Parameters 4

- Make a function that receives a list value
  - print the list
  - make some change on the list
  - return to main
  - print that list

***Mutable objects**:*

list, dict, set, byte array

Immutable objects:

int, float, complex, string, tuple, frozen set [note: immutable version of set], bytes.

Making Changes to Parameters 5

Function printlst(lst)

```
print (lst)
lst[0]= 100
lst[len(lst)-1] = 200
print (lst)
```

main()

```
lst = [ 1, 2, 3. 4. 5]
printlst(lst)
```

Lab 3: Make a function to find the min number

- Implement the function “mineven(numbers)”
  - Delete the **min** value among **even numbers** from the list
  - parameter numbers
    - list
  - **return** value
    - the min value among even numbers in the list
- Purpose of this lab
  - Check the list value after the function call
  - The list value has been changed in the function

```
def mineven(numbers):

...
return min
```

- Call Example
  - `numbers = [10, 20, 30, 15, 5]`
  - `result = mineven(numbers)`
  - `print (result)     # Expected: 10, numbers = [20, 30, 15, 5]`

go to navigator

Keyword Arguments

go to navigator

Keyword Arguments

- Write an argument as following:
  - parameter_name = value

```
def getsum(n1, n2):
        print (" N1 is " , n1);
        print (" N2 is " , n2);
        return n1 + n2

def main():
        total = getsum(n2=100, n1=200)
        print ("The value of total is {0}".format(total))

if __name__ == '__main__':
        main()
```

Global variables

go to navigator

Global Variable

- A global variable is **accessible to all the functions** in a program file.

```
gvariable = 100

def myfunction1():
        print ('Inside the function ', gvariable)

def main():
        global gvariable
        print ('Before call myfunction1 ', gvariable)
        gvariable += 11
        myfunction1()
        print ('After call myfunction1 ', gvariable)

if __name__ == '__main__':
        main()
```

Returning multiple values

go to navigator

Returning multiple values

```
def getinput():
        n1 = input()
        n2 = input()
        return n1, n2

def main():
        num1, num2 = getinput()
        print ("The value of num1 and num2 are {0} and
{1}".format(num1, num2))

if __name__ == '__main__':
        main()
```

Lab 4: Returning Multiple values

- Make a function minmax( ) to return the least and greatest value in the list
  - `minmax(numbers)`
    - parameter numbers is a list type
  - return two values in a list
    - the least and greatest value
- Call example
  - `numbers = [1,2,3,4,5]`
  - `minval, maxval = minmax(numbers)`

go to navigator

*args, **kwargs

1.)*args (Non-Keyword Arguments)

2.)**kwargs (Keyword Arguments)

go to navigator

Arbitrary Arguments

- *args
  - **args* allows you to take in more arguments
    - than the number of formal arguments that you previously defined.
  - ```
    def myFun(*args):
        for arg in args:
            print (arg)
    ```
- ****kwargs**
  - pass a keyworded, variable-length argument list.
  - the name *kwargs* with the double star.
    - The reason is because the double star allows us to pass through keyword arguments
  - A keyword argument is where you provide a name to the variable as you pass it into the function.

```
def myFun(**kwargs):
        for key, value in kwargs.items():
                print ("%s == %s" %(key, value))


myFun(first ='Liver', mid ='High', last='School')
```
```

Arbitrary Arguments

- **kwargs

```
def myFun(first, **kwargs):
        for key, value in kwargs.items():
                print ("%s == %s" %(key, value))


myFun('Python', mid ='C++', last='Java')
```

- Using *args and **kwargs to call a function

```
def myFun(arg1, arg2, arg3):
        print("arg1:", arg1)
        print("arg2:", arg2)
        print("arg3:", arg3)

args = ("Python", "C++", "Java")
myFun(*args)

kwargs = {"arg1" : "Python", "arg2" : "C++", "arg3" : "Java"}
myFun(**kwargs)
```

Lab 5: **kwargs

- Make the same code as below
  - Figure out the usage of **kwargs
  - Make a function that has the **kwargs parameter
  - Call “printscore()” with the different “kwargs” values

```
def printscores(**scores):
    for k,v in scores.items():
        print (f'Subject {k:>10}: {v:>10}')


kwargs = { 'Math': 90, 'English':100, 'Computer': 90}
printscores(**kwargs)
```

go to navigator

Example: *args

- There is a string
  - stringvalue = “Python Programming”
  -
- Make functions
    - printstring(*values)
    - printstring(values)
  - When you call this function
    - printString(*stringvalue)
    - printString(stringvalue)
- Describe the difference between two functions

```
def printfunction(*str):
    for v in str:
        print (v)
```

```
def printfunction(str):
    for v in str:
        print (v)
```

What if the value is integer list

go to navigator

Lab 6: *args

- Make the same code as below
  - Make two functions that prints the list values
  - Figure out the difference *str and str

```
def printfunction1(*str):
    print (str)
    for v in str:
        print (v)
```

```
def printfunction2(str):
    print (str)
    for v in str:
        print (v)
```

```
def main():
    str = 'Python Programming'
    printfunction1(*str)
    printfunction2(str)

    morestr = 'C++ Programming'
    printfunction1(str, morestr)
    # printfunction2(str, morestr)  # Error
```

go to navigator

Arbitrary Arguments

- **Using arbitrary arguments to save the returned values**
  - `first, *others, last = findEvenNumbers( lst )`

```
def retEven(lst):
        evenlst = []
        for v in lst:
                if v % 2 == 0:
                        evenlst. append(v)
        return evenlst

lst = [1,2,3,4,5,6]
first, *others, last = retEven(lst)
print (first)
print (last)
print (others)
```

Lab 7: Arbitrary Arguments *args

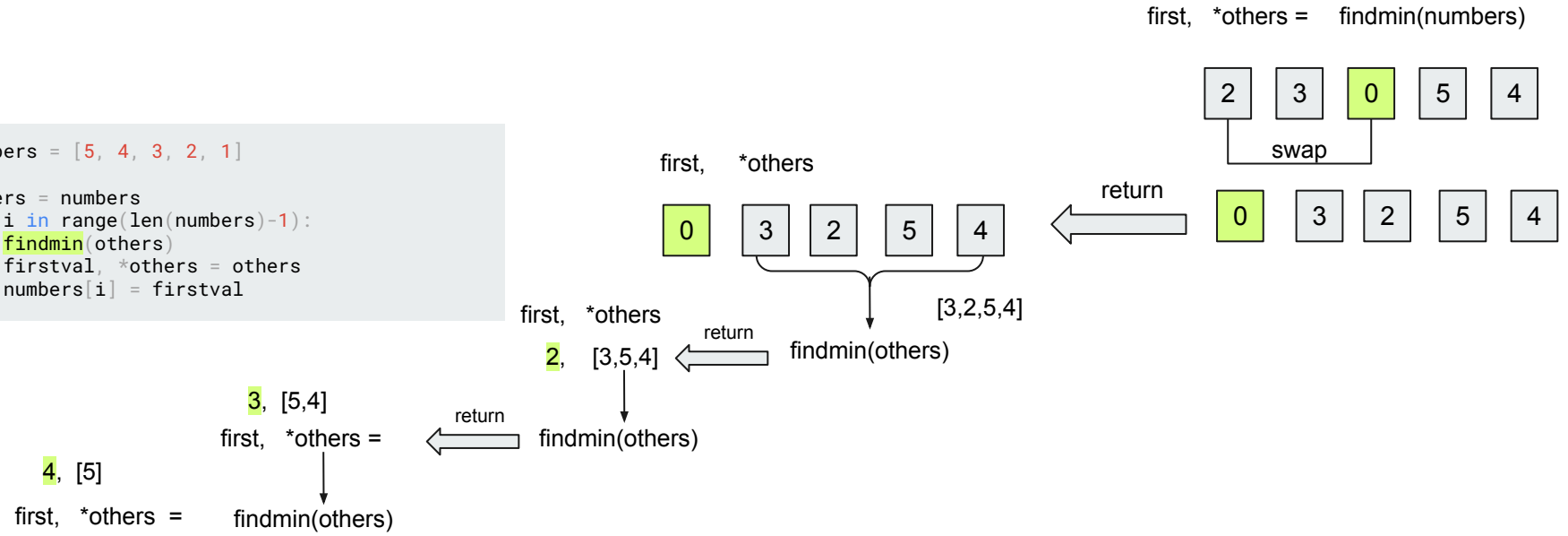
- [Step 1]
  - We are going to make the function "**`findmin`**`(numbers)`"
    - that **returns** the **list** which has the min value at the first position.
    - In this function, we are going to find min value and swap it with the first element.
- [Step 2]
  - after **calling findmin(numbers)**,
    - we will save the numbers as two variables by using "**arbitrary args**"
    - `first, *others = findmin(numbers)`
  - Repeat the function call "**`findmin(others)`**" until the len(others) = 1
    - while len(others) >= 2, or
    - for i in range(len(numbers) - 1 times)
- [Step 3]
  - print the list "numbers"
  - When you print numbers, you can find **the numbers has the sorted order.**
    - Do not use any **`sort()`** function

go to navigator

Lab 7: Arbitrary Arguments *args

```
numbers = [5, 4, 3, 2, 1]

others = numbers
for i in range(len(numbers)-1):
    findmin(others)
    firstval, *others = others
    numbers[i] = firstval
```



go to navigator

Call by Reference: list

- When we use the list as a parameter,
  - if the value of list are supposed to be changed in the function,
  - do we need to return the “list” to have the changed effect in the caller function?

```
numbers = [1,2,3]
myfunc(numbers)
# numbers[0] = 99?
```

caller

```
def myfunc(lst):
        lst[0] = 99
        return lst
```

return?

myfunc

What about integer?

Add the line to check ID.
If they have the same id,
it means “call-by-reference”

go to navigator

Lab 8: Call by reference : checking id values

- Make a program to check the ID of the list before/after calling the function
  - no test function in GitHub

```
numbers = [1,2,3]
print (id(numbers), 'numbers ID ')

# 1) retlst = myfunc(numbers)
# 2) a, *retlst = myfunc(numbers)

retlst[0] = 999
print (id(retlst), 'retlst ID: returned from myfunc')
```

```
def myfunc(lst):
    print (id(lst), 'ID of lst in myfunc() as soon as the function starts')
    lst[0] = 999
    print (id(lst), 'ID of lst in myfunc() after changing value ')
    return lst
```

- Figure out the difference between Run 1) and Run 2)
  - Remove one # at each try
- Explain the difference and Elaborate lessons learned.
  - Explain why we do not need to return the “lst” to let the “caller” see the changes

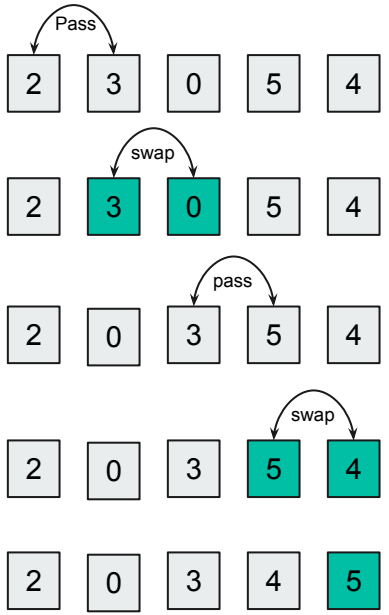
go to navigator

More Labs

go to navigator

Lab 9 : Function Bubble( )

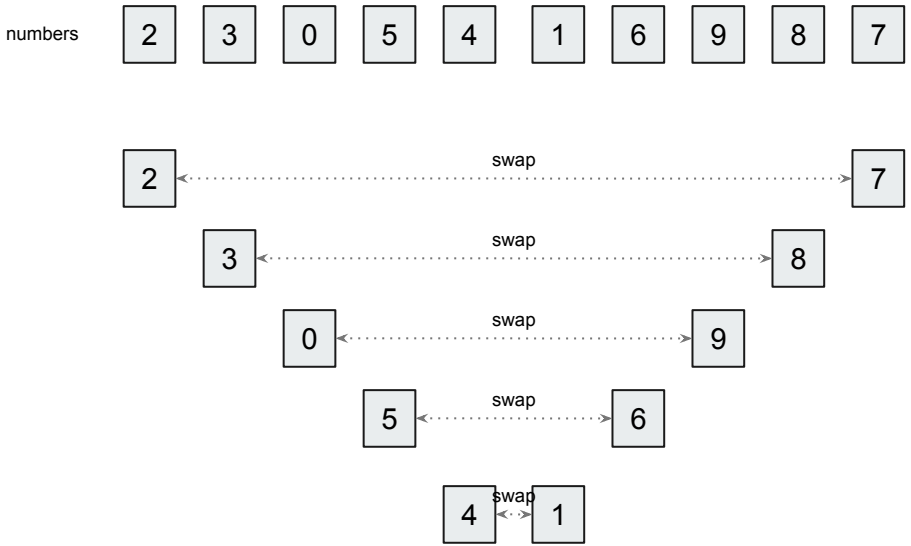
- Make a function “`def bubble(numbers)`”
  - Compare all adjacent pair values and if the left value is greater than the right one, swap two values
- Return value
  - nothing
- Make sure that
  - after function “bubble(numbers)” call
  - the last element is the greatest number



go to navigator

Lab 10 : Function foldandswap( )

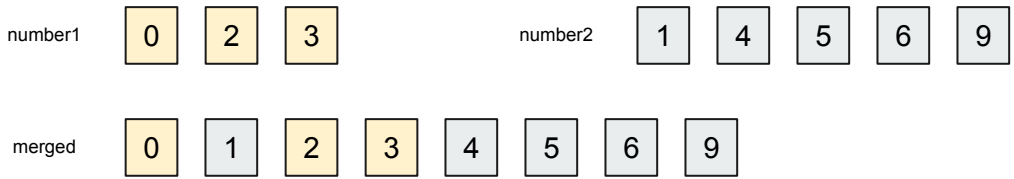
- Make a function "`def foldandswap(numbers)`"
  - Swap two values that face each other when folding the list
- Return value
  - nothing
  - Do **not** use any python **library functions**. Practice the implementation of your algorithm



go to navigator

Lab 11. Merge two list

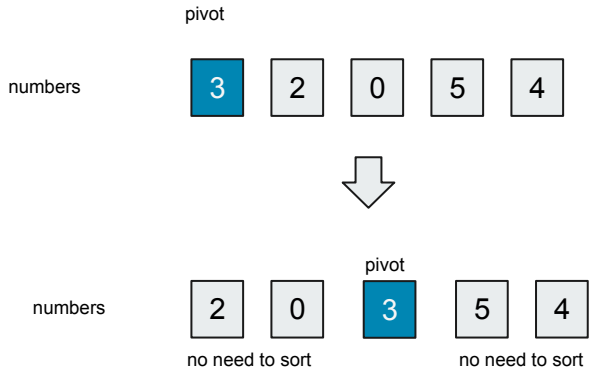
- make a function **`merge(number1, number2)`**
  - Two lists number1 and number2 has already sorted in ascending order
  - make a new list “merged” that is a merged list of two lists with keeping sorted order
    - Do not use sort() or sorted() functions.
- Return value
  - merged list



go to navigator

Lab 12. Split ( )

- make a function **`split(numbers)`**
  - select the **first** element as a **pivot** value
  - Split the list values based on the pivot value
    - (less than or equal to pivot values) + pivot + (greater than pivot values)
      - left and right are not required to be sorted
- Requirement
  - Do not use any sort(), sorted() functions
  - **One** for-loop or while loop.
    - O(N), N comparisons at most
- Return value
  - numbers



go to navigator

yield

iterator

generator

go to navigator

Iterator

https://realpython.com/introduction-to-python-generators/

- **Iterable**
  - all things you can use “for .. in ..” are iterable; **list, string, dict, file**

```
list1 = [1, 2, 3]
for i in list1:
```

- **Iterator**
  - is an object that is used to iterate over iterable objects

```
mytuple = (10, 20, 30)
myiterator = iter(mytuple)
for i in myiterator:
    print (i, end=' ')
print ()
```

```
mytuple = (10, 20, 30)
myiterator = iter(mytuple)
value = next(myiterator)
value = next(myiterator)
value = next(myiterator)
print (value)
```

```
myiterator = iter(mytuple)
while True:
    try:
        value = next(myiterator)
    except StopIteration:
        print ('Stop Iteration')
        break
    else:
        print (value)
```

Yield

https://realpython.com/introduction-to-python-generators/

- **Generators**
  - are iterator, you can iterate only over once
  - Generators do not store all the values in memory, **they generate the values on the fly**:
- Yield
  - return a generator

```
def create_gen():
      for i in range(3):
            yield (i*i)
mygen = create_gen()
for i in mygen:
      print (i)
```

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator:
...     print(i)
0
1
4
```

When do we use yield?

We should use yield when we want to iterate over a sequence, but don't want to store the entire sequence in memory.

Iterator vs Generator

https://realpython.com/introduction-to-python-generators/

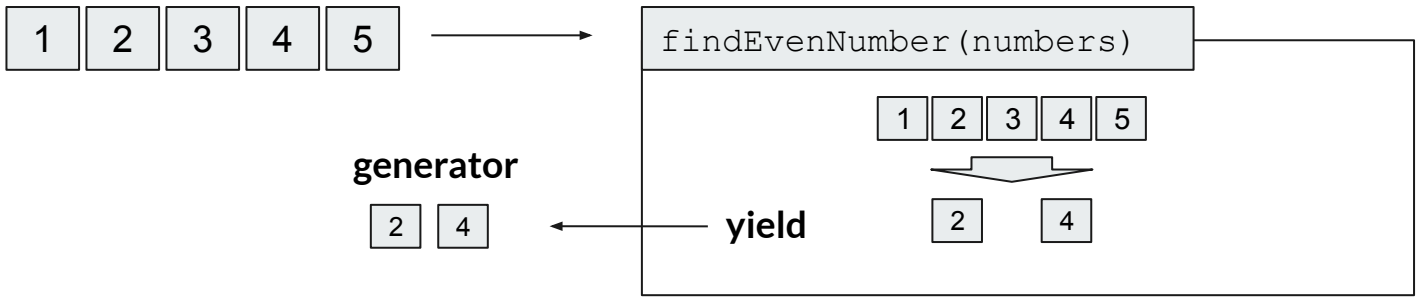
- Generator vs Iterator

Iterator	Generator
Class is used to implement an iterator	Function is used to implement a generator.
Iterators are used mostly to iterate or convert other objects to an iterator using iter() function.	Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop
Iterator uses iter() and next() functions	Generator uses yield keyword
Every iterator is not a generator	Every generator is an iterator

In summary:
Iterators are objects that have an __iter__ and a __next__ method.
Generators provide an easy, built-in way to create instances of Iterators.

Lab 13 : Yield : Generates Even Numbers

- Example of practicing yield statement
  - Make a function to return a generator for the even number in the list
  - “`findEvenNumber(numbers)`”
    - parameter numbers: the list of integer values
  - In the function “findEvenNumber( )”,
    - return the **generator** to iterate the all even numbers



go to navigator

Lab 14 : Yield. Fibonacci numbers

- Make a function to return the generator for the fibonacci series numbers
  - The definition of Fibonacci number

The Rule is $\mathbf{x_n = x_{n-1} + x_{n-2}}$
Starting Sequence $x_0 = 0$, $x_1 = 1$

- `def fibo(N):`

  `yield fibo_series`

X_{n-2} X_{n-1} X_n

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	…
0	1	1	2	3	5	8	13	21	34	55	…

i=1 i=2 i=3 …

go to navigator

Lab 15: yield: Consonant

- Make a function “consonant( )” that returns
  - a generator to iterate the consonant
- For example,
  - function consonant( ) will receive a string
    - “Python Programming”
    - and will return a generator to iterate
      - P, y, t, h, n, P, r, g, m, m, n, g

go to navigator

lambda

go to navigator

Lambda

- Lambda function
  - a small anonymous function
  - can take any number of arguments, but can only have one expression
  - `lambda arguments : expression`
  - return value : <function object>

```
lfn = lambda x : x + 10

a = lfn(20)
print (a)
```

Lambda

- Examples

Lambda function that returns x squared

```
squared = lambda x : x * x

print (squared(10))
```

Lambda function that returns x > y

```
greater = lambda x, y : x > y

print (greater(10, 20))
print (greater(20, 10))
```

Lab 16: Lambda 1

- Make a lambda function to
  - return the greater value between two values
    - `greater = lambda x, y: # your code`
  - return the values in the list which are greater than 50
    - `filter = lambda mylist: # your code`

go to navigator

Lambda 2 : Function that returns a lambda function

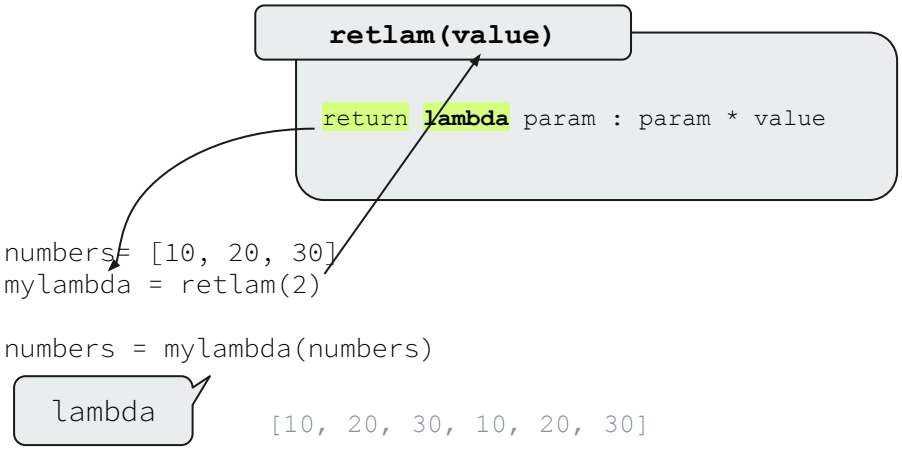
- Purpose
  - to make lambda function with the given parameter value
  - Customizing the lambda function

Returns a lambda (See Lab 17)

```
def retlam( value):
      return lambda parameter : parameter * value

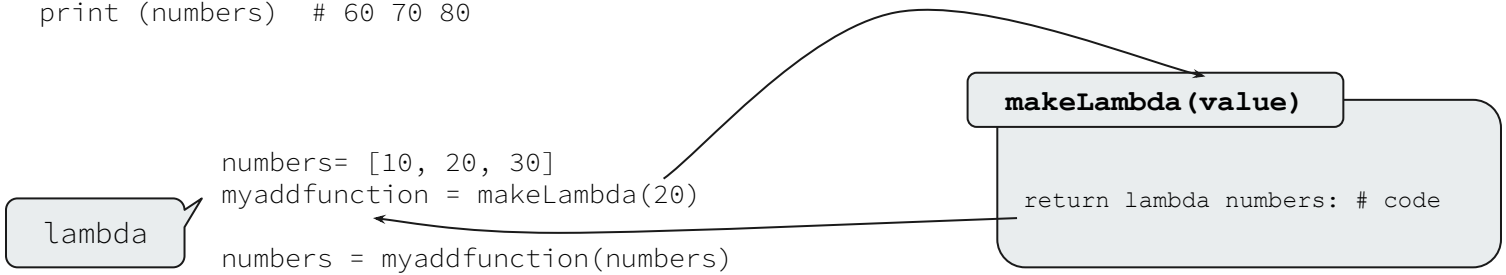
mylambda = retlam(10)
print (mylambda(20))

mylambda = retlam(20)
print (mylambda(20))
```



Lab 17: Lambda 2. return a lambda function

- Make a function `makeLambda(value)` to
  - return a lambda function that
    - add `value` to each element of list `numbers` that is a lambda function parameter
- Make a test code in main() to call your `makeLambda( )`
  - `numbers = [10, 20, 30]`
  - `myaddfunction = makeLambda(100)`
  - `numbers= myaddfunction(numbers)`
  - `print (numbers) # 110 120 130`
  -
  - `myaddfunction = makeLambda(-50)`
  - `numbers= myaddfunction(numbers)`
  - `print (numbers)  # 60 70 80`
  -



go to navigator

Lambda 3: Function name as a parameter

- Example 1

Receives a function as a parameter in a lambda function

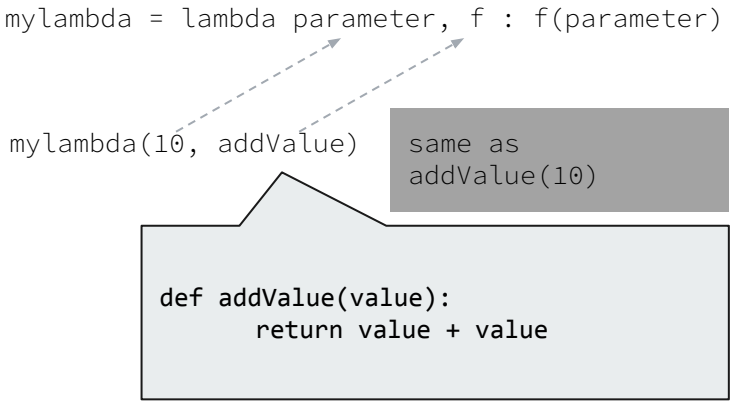
```
total = 0
def addValue(value):
      return total + value

mylambda = lambda parameter, f : f(parameter)

mylambda(10, addValue) # 10
```

```
def subtractValue(value):
      return total - value


mylambda(5, subtractValue) # 5
```



Lambda 3: function name as a parameter

- Example 2 **Receives a function as a parameter in a lambda function**

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

mylambda1 = lambda x, f : len(f(x))

mylambda1(numbers, collectOddElm)

collectOddElm = lambda numbers: [ numbers.pop(i) for i in range(len(numbers)//2) ]

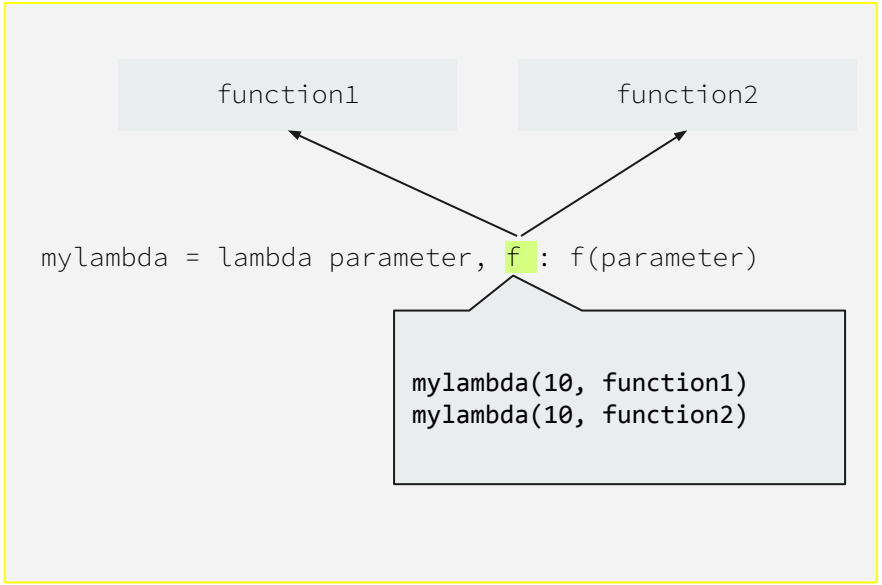
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

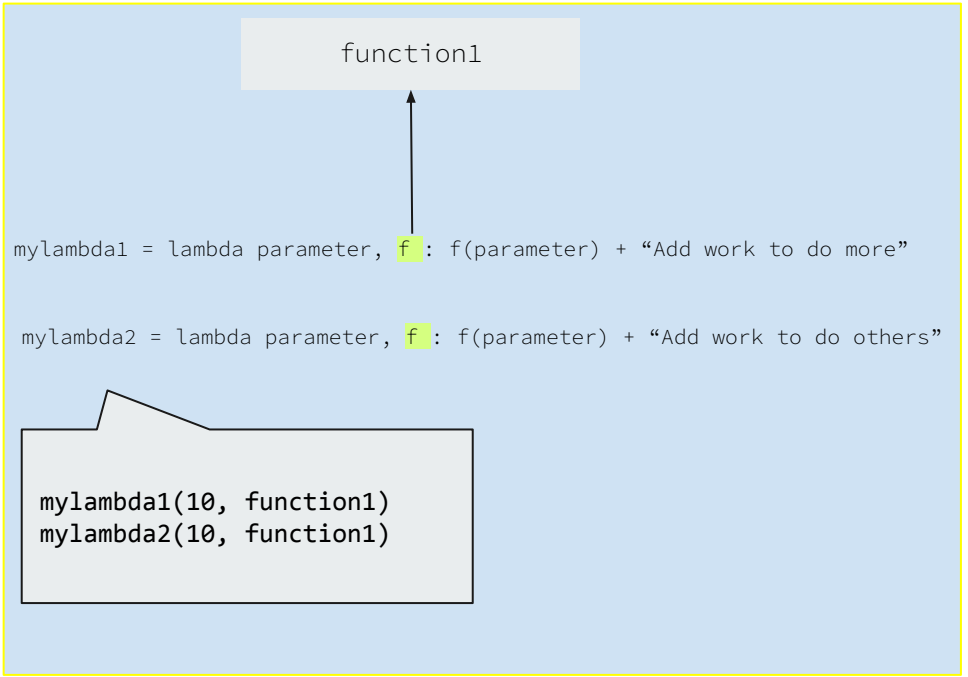
```
mylambda2 = lambda x, f : max(f(x))

mylambda2(numbers, collectOddElm)
```

lambda 3: function name as a parameter

- Summary





Lab 18: Lambda 3. function name as a parameter

- Implement 3 lambda functions in Example 2 (page 57)
  - **colloectOddElm**
  - **mylambda1**
  - **mylambda2**

- Make a test code in main() to call your `lambda functions`

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print (numbers)
print (mylambda1(numbers, collectOddElm))

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print (numbers)
print (mylambda2(numbers, collectOddElm))
```

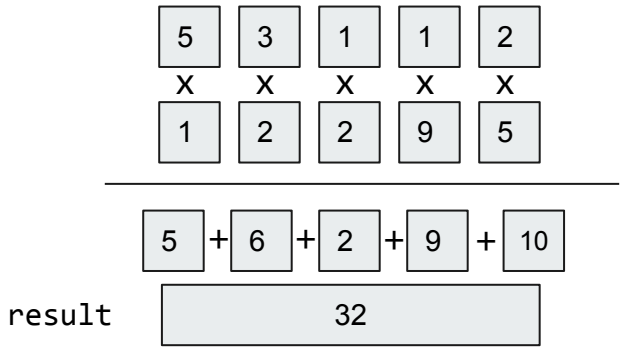
go to navigator

Assignments

go to navigator

Assignment 5-1 : Sum of Product

- Make a function “sumProduct()” that
  - receives two lists of integers and returns the sum of multiplying the corresponding list items
  - Assumption: two lists must be the same length




```
def sumProduct(l1, l2):
     return sumP
```

```
result = sumProduct(list1, list2)
```

go to navigator

Assignment 5-2 : Prime numbers list

- Make a function “primeNumbers(begin, end)” that
  - find the all prime number between ‘begin’ and ‘end’ (inclusive)
  - return the prime numbers as a list

```
def primeNumbers(begin, end):
      // Make your code here

      return plist
```

Inside the function `primeNumbers()`, it is better to call another function `isPrime()`

```
begin = 10; end = 20
result = primeNumbers(begin, end)
print (result)
```

go to navigator

Assignment 5-3 : Shift left / right

- Make a function “shiftN(stringvalue, direction, N)” that
  - Move the **string value** left or right based on the direction (0 or 1) to the N position
  - stringvalues example
    - 00011100
  - direction
    - 0 : left
    - 1 : right
  - N
    - shift count (e.g, N = 3, direction = 0, it means that move values to the 3 left position)
      - 00011100 → 11100000
  - Zero-padded
    - When you move to left/right, the space from left/right will be appended with 0 values
    - Example
      - 00011100, Shift Left 2 => 011100 00 **zero-padded**
      - 00011100, Shift right 2 => 00 000111
  - Do not use any binary function such as bin ( )
    - make your iteration form to get “shift-left/right string”

```
shift-left means multiplication
shift-right means division
```

go to navigator

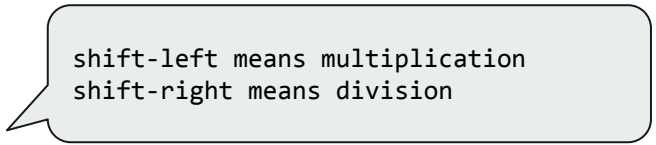
Assignment 5-3 : Shift left / right

- Make a function “shiftN(stringvalue, direction, N)” that
  - Move the **string value** left or right based on the direction (0 or 1) to the N position

```
def shiftN(stringvalue, direction, N):
      // Make your code here

      return shiftedstring
```

```
str = '001100'
print (rstr)     # 001100
print (int(str, 2)) # 12 = decimal of 001100
rstr = shiftN(str, 0, 2)
print (rstr)     # 110000
print (int(str, 2)) # 48 = 12 * 2 * 2
```



go to navigator

Assignment 5-4 : lambda function

- Write a function "**`mystrip(strvalue)`**" that uses a **lambda** function **`isspace`**
  - strip the **space(' ')** from the original string and then return the stripped string
    - no need to consider any other white character like '\t', '\n', and so on
  - the lambda function "isspace()" will check the letter is **space character** or not, and then return true or false
  - the function "mystrip()" will make the string except the space letter using the lambda function, then return the result string.

```
# lambda function
isspace = lambda .........

# function mystrip()
def mystrip(strval):
.........# we will call the lambda function isspace() here
```

```
strval = 'Python programming section 2'
res = mystrip(strval)
print (res)
```

go to navigator

Assignment 5-5 : yield

- Write a function that returns the generator for the non-space alphanumeric letter from the original string
  - make a function "**getalnum()**" that returns the generator (you should use "**yield**" statement)
  - Extract all the non-space **alphanumeric** values (use **isalnum**()) from the original parameter string value
    - e.g., "Python Programming" → this function will return the **generator** for "PythonProgramming"

```
# function getalnum()
def getalnum(strval):
      yield ……
```

```
msg = 'Python programming section 2'
res = getalnum(strval)
for v in res:
      print (v)
```

go to navigator

Quiz

Introduction to Python Programming

go to navigator