Tirgul 2

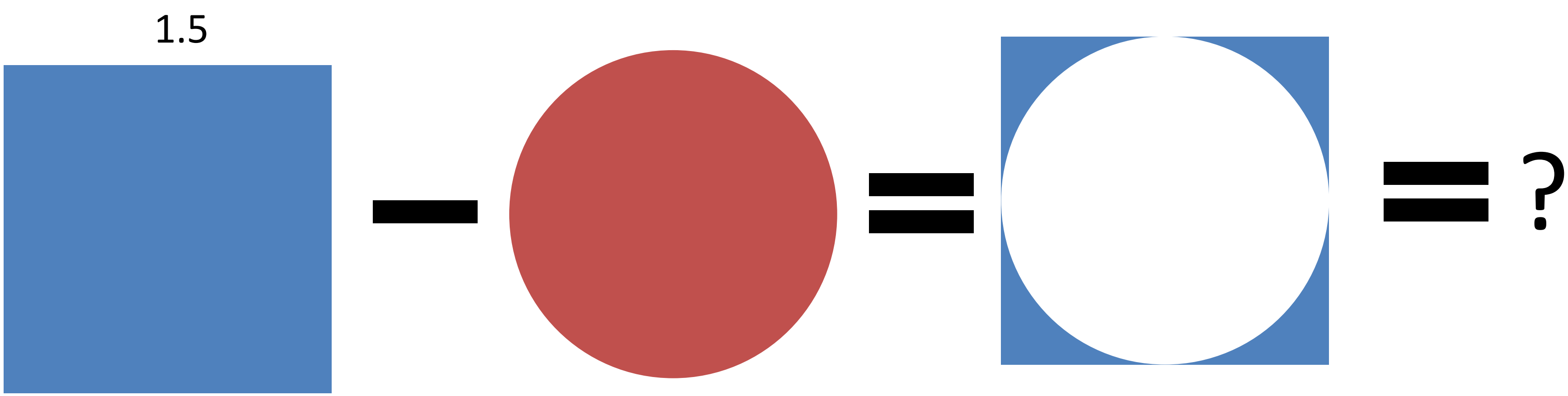
Basic programming

Overview

- Variables
- Types : int, float, string
- User input
- Functions with input and output
- The Boolean type and Boolean operations
- Conditional operation (if…else…)

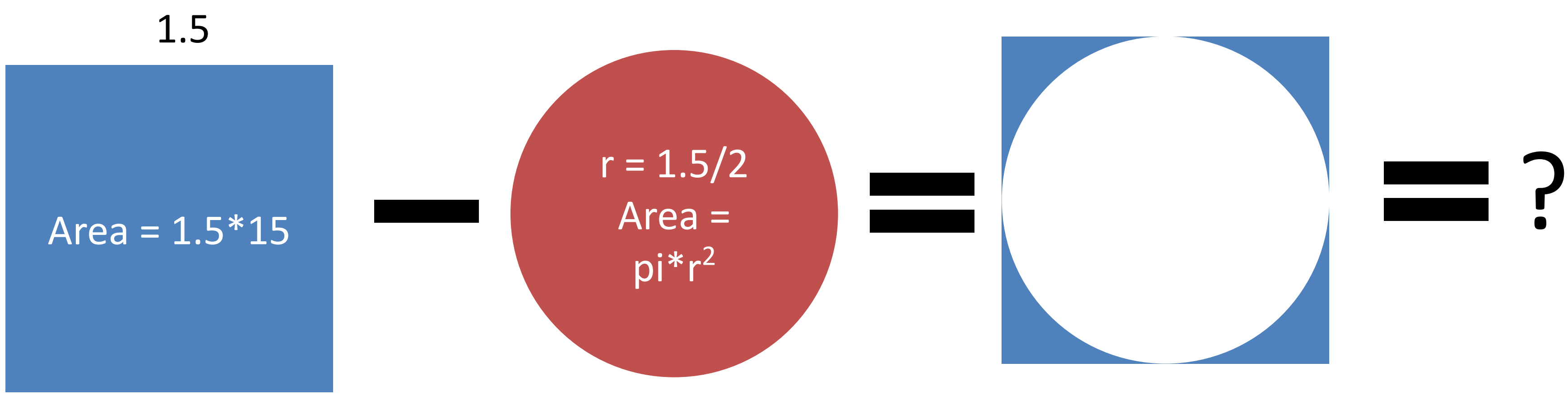
Variables - Motivation

- Write a program which calculates the difference in the areas of a square with side = 1.5 and the circle enclosed within it.



Variables - Motivation

<u>1.5*1.5 –3.14*(1.5/2)2</u>**



Variables - Motivation

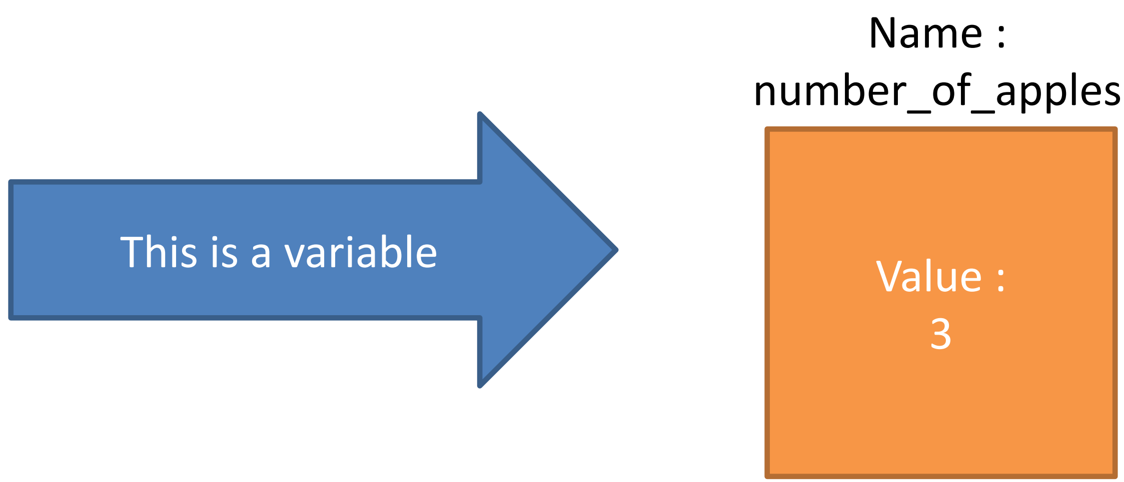
- Two problems :
  - The expression *1.5*1.5 –3.14*(1.5/2)**2*
  - is really **difficult to understand** :
    - When you get back to it after one week
    - When debugging
  - When the side of the square changes. Should you have **an expression per side-length**?
    - Side=1.5 : *1.5*1.5 – 3.14*(1.5/2)**2*
    - Side=3.7 : *3.7*3.7 – 3.14*(3.7/2)**2*
    - Side=9 : *9*9 – 3.14*(9/2))**2*

Variables - Motivation

- Wouldn’t it be much more ***readable, modular, easy to modify*** in this format :
  - side = 1.5, PI = 3.14
  - square_area = side*side
  - radius = side/2
  - circle_area = PI*r^2
  - answer = square_area – circle_area

Variables

- Variables let us define “memory units” which can “remember” values.
- Variables have 2 main components :
  - name
  - value



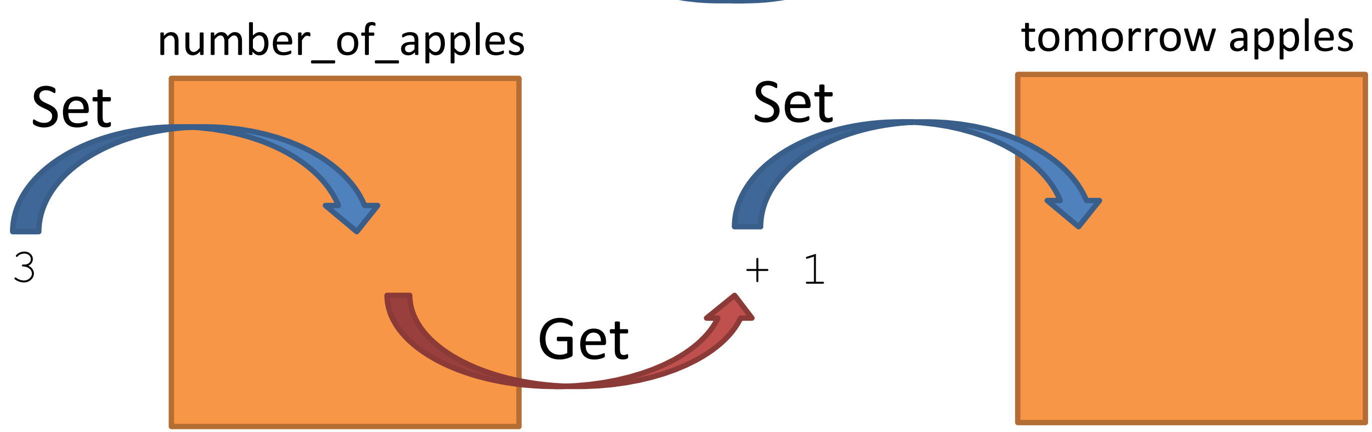
Variables

- Variables have 2 main functionalities :
  - Set their value

```
number_of_apples = 3
```

  - Get their values

```
tomorrow_apples = number_of_apples + 1
```



Variables – Naming conventions

- Use lower case letter
  - `number,apples`
- Separate multiple words with underscore
  - `word_and_more_words`
- Use meaningful names for names (don’t be shy to open a dictionary)
  - `z = x/y ???`
  - `words_per_page = words_in_book/number_of_pages` ☺
- Use capitals for constants (variables which do not change their value after first initialization)
  - PI = 3.14, ERROR_MESSAGE = ‘You had an error’

Types

- Can we perform the following command ?
  - `x = 3 + 5`
- And this one ?
  - `x = 3 + “hello”`
- Why not? *3* and *‘hello’* are not of the same category. The name Python gives to the categories which differentiate between objects such as *3* and ‘*hello*’ are called **type**.

Types

- `int` (Integer) : represent an Integer number (מספר שלם).
  - E.g. 1024, 13, 92,0
- `float` : represent a fractional number.
  - E.g. : 0.0, 15.62545, 3.14
- `str` (String) : represent text, a list of characters. Defined between a couple of apostrophe or quotes (equivalent).
  - E.g. ‘hello’, “hello”, ‘13’

Types

- The `type()` function receives a value and return its type.
  - `type(3)` int
  - `type(3.0)` float
  - `type('3.0')` str
- What happens when we **mix types**?
  - `type(1 + 0.5)` float
  - `type(1 + 'some string')` ?

Types

- What happens when we **mix types**?

```
type(1 + 'some string')
TypeError: unsupported operand
type(s) for +: 'int' and 'str'
```

This is an error message which tells us we have tried to execute a command not supported by the language.

Error message

- **Error messages are our friends**, they help us detect bugs in our program and point out how to fix them.
- When you get an error "*keep calm and read the error message*".

Error message - Example

```
>>> x = 49
>>> x/(49**0.5 - 7)
Traceback (most recent call last):
  File "C:/my_python/test.py", line 9, in
<module>
    x/(49**0.5 - 7)
ZeroDivisionError: float division by zero
```

Remember - "*keep calm and* ***read*** *the error message*"

Error message - Example

```
Traceback (most recent
call last):
  File
"C:/my_python/test.py",
line 2, in <module>
    x/(49**0.5 - 7)
ZeroDivisionError: float
division by zero
```

The error occurred when we ran the program saved at this file.

Error message - Example

```
Traceback (most recent
call last):
  File
"C:/my_python/test.py",
line 2,
    x/(49**0.5 - 7)
ZeroDivisionError: float
division by zero
```

The error is located in this line in the file.

Error message - Example

```
Traceback (most recent
call last):
  File
"C:/my_python/test.py",
line 2, in <module>
    x/(49**0.5 - 7)
ZeroDivisionError: float
division by zero
```

The command which caused the trouble was this.

Error message - Example

```
Traceback (most recent
call last):
  File
"C:/my_python/test.py",
line 2, in <module>
    x/(49**0.5 - 7)
ZeroDivisionError:
division by zero
```

The error itself is of type *Zero division*

Error message - Example

```
Traceback (most recent
call last):
  File
"C:/my_python/test.py",
line 2, in <module>
    x/(49**0.5 - 7)
ZeroDivisionError: float
division by zero
```

And it happened because we have tried to divide a float by 0

Error message - Example

```
Traceback (most recent call last):
  File "C:/my_python/test.py", line 2, in <module>
    x/(49**0.5 - 7)
ZeroDivisionError: float division by zero
```

No matter what, **you are going to have bugs**. Error messages make the debugging process much more productive.
With time, you'll meet more types of errors and you'll get better in understanding their meaning, locating your bugs and fixing them.

But what if we do want to mix types?

```
my_apples = 3
print('I have ' + my_apples + ' apples')
TypeError: Can't convert 'int' object to str implicitly
```

- The error message tells us we have tried to **convert** an `int` to a `str` but we cannot do this *implicitly*.
- So let’s do it *explicitly*.

Converting types (casting)

- `int`, `float` and `str` are not only names of types but also names of functions which **convert between types**.
- Example :
- `str(5)` □ `'5'`
- `float(5)` □ `5.0`
- `int('5')` □ `5`

Converting types – `int()`, `float()`

- Converts **string** representing numbers to the represented numbers
  - `int('5')`□`5`
    - Cannot convert strings not representing an int :
      - ~~`int('5.5')`, `int('Hello')`~~
- Converts **float** to int by rounding the number down.
  - `int(5.9)`□`5`
- Converts **string** representing numbers to the represented numbers
  - `float('5.5')`□`5.5`
    - Cannot convert strings not representing a float:
      - ~~`float('Hello')`~~
- Converts **int** to float by treating it as a round number.
  - `float(5)`□`5.0`

User input

- To make a program interactive we can ask **the user** for some inputs and act upon them.
- The function `input(s)` :
  - Prints to the screen `s`
  - Halts the program execution and waits for the user to insert some input and press enter
  - Return a string representing the user's input

User input - Example

```
square_side = input('Insert side length: ')
# Wait for user …
3
```

- The value of `square_side` is 3

```
area = square_side * square_side
```

Will this work?

User input - Example

```
area = square_side * square_side
'3'*'3'
TypeError: can't multiply sequence by
non-int of type 'str'
```

Input returns a string, and we can’t multiply string by string. So what do we do? **Convert types**

User input - Example

```
square_side = float(input('Insert side length: '))
# Wait for user …
3
area = square_side * square_side
```

The value of area is 9.0

Functions with input

```
def function_name(param1, param2,…,paramN):
    #indented code is here
    #as usual
```

The name of the parameters that shall be used in the functions, are listed within the parentheses.

Functions with input

- When we call a function with input parameters, we can use the parameters’ value inside the function using their name.

Functions with input

```
def print_hello_user(user_name):
    print('hello ' + user_name)


print_hello_user('John')
```

Function definition

Function body (implementation)

Function call

Functions with input

```
def print_hello_user(user_name):
    print('hello ' + user_name)
```

3) We can use `user_name` inside the function and it will have the value with which the function was called

2) The function parameter (user_name) is assigned the value with which the function was called ('John')

1) When we call the function

```
print_hello_user('John')
```

Functions with input

```
def print_hello_user(user_name):
    print('hello ' + user_name)
```

```
print_hello_user('John')
 >>> hello John
print_hello_user('Doe')
 >>> hello Doe
```

A word about scopes

```
def print_hello_user(user_name):
    print('hello ' + user_name)
```

```
print_hello_user('John')
print('Good bye ' + user_name)
```

What do you think will happen?

A word about scopes

```
def print_hello_user(user_name):
    print('hello ' + user_name)
```

```
print_hello_user('John')
print('Good bye ' + user_name)
```

NameError: name ‘user_name' is not defined

The parameter `user_name` is defined at the function `print_hello_user` and hence it is not known outside the **scope** of the function.

A word about scopes

```
ROOT = 2
def square_root(number):
    print(number**(1/ROOT))

square_root(4)
  >>> 2.0
```

A new scope still knows the variables of the scope in which it is contained.
Here, `ROOT` is defined in the general scope hence the function which opens a new scope, still knows the value of `ROOT`.

A word about scopes

```
x = 2
def example():
    x = 5
    print(x)
example()
print(x)
>>> 5  # When in the function, a new scope is defined, and the new variable x shadows the definition of the x from upper scope.
>>> 2  # When exiting the function's scope, the scope of the function is not regarded any more hence the x of the outer scope kicks in.
```

However, this is confusing and considered bad style. To avoid confusions of the sort, pick unique variable names across scopes.

A function with more than 1 input

```
def get_details(name, password):
  print('Name is :' + name + ', Password
  is:' + password )

get_details('John', '1234')
```

Q: How does the function knows which value goes where (that *name* is *John* and *password* is *1234* and not the other way around).
A: According to **variables order**.

Functions’ parameters default value

- Sometimes ….
  - A function has an obvious use case that will be utilized most of the time
  - You have prepared a good option for the user but don’t want to force her to use it
- In such cases, you can define a **default value** to the function’s parameters. A value that will be used if no other value is specified.

Functions' parameters default value

```
def shoresh(number, root=2):
    print(number ** (1/root))
```

- The first parameter, `number`, has no default value. Hence every call to the function must indicate its value.
- The second parameter, `root`, has a default value. Hence if we don't indicate its value it will get the default declared value, 2.

Functions’ parameters default value

```
def shoresh(number, root=2):
    print(number ** (1/root))

shoresh(64) # Here we didn’t indicate the second variable, hence the default value was used
>> 8
shoresh(64, 3) # Here we indicated the second variable, hence its value was used and not the default
>> 4
```

Function’s return value

- Many times we want functions to not only perform some functionality, but also **to return a result**.
- Using the `return` keyword, a function is able to return a value.

Function's return value

```
def always_return_5():
    return 5
    print('hi')
```

`return` means we terminate the function's run and return the value 5

This line is never executed

Function’s return value

```
def always_return_5():
    return 5
    print('hi')
```

The function returns the value 5, which is considered as a regular int by the + operator.

```
print(3 + always_return_5())
>>> 8
```

Function calling a function

- We can use the return value of one function as another function's input.

```
def per_week(per_day=1):
    return per_day * 7

def per_year(how_many_per_week):
    return how_many_per_week * 52

print('Apples per year : ' + str(per_year(per_week())))
```

What happens here?

Function calling a function

```
def per_week(per_day=1):
    return per_day * 7 # return 7

def per_year(how_many_per_week):
    return how_many_per_week * 52

print('Apples per year : ' + str(per_year(per_week())))
```

`per_week()` is called with no value and so gets the default value, 1, hence its return value is 7.

Function calling a function

```
def per_week(per_day=1):
   return per_day * 7 # return 7

def per_year(how_many_per_week):
    return how_many_per_week * 52 # return 364


print('Apples per year : ' + str(per_year(7)))
```

`per_year()` is called with the value 7 and so returns the value 364

Function calling a function

- We can use the return value of one function as another function's input.

```
def per_week(per_day =1):
   return per_day * 7 # return 7

def per_year(how_many_per_week):
    return how_many_per_week * 52 # return 364


print('Apples per year : ' + str(per_year(7)))

>>> Apples per year : 364
```

Multiple outputs functions

- To return more than one value, separate return values by comma

```
def diff_and_ratio(num1, num2):
    return num1-num2, num1/num2

diff, ratio = diff_and_ratio(1, 5)
print(diff)
print(ratio)
```

None

- `None` is a special value which is used to represent absence of value.
- Every function which does not return value explicitly, return `None` implicitly.

None - example

```
def print_hi():
    print('hi')
x = print_hi() # x is assigned
  the value None
print(x)
>>>hi
>>>None
```

The Boolean type

- Like `int`, `str` and `float` , **Boolean** is another Python type.
- Boolean can get only one of two values :
  - `True`
  - `False`

```
type(True)
>>> <class 'bool'>
```

Boolean expressions

- Boolean expressions are expressions which use **Boolean operators** to evaluate a value of True or False.
- For example **>** is a Boolean operator. Its Boolean evaluation is “Is the object on the right larger than the object on the left?”
- **5 > 7** is a Boolean expression because it uses a Boolean operator. Its value is False.

Boolean operators

Symbol in Python	Name	Example
>	Greater than	7 > 6 (True)
>=	Greater than or equal to	7 >= 7 (True)
<	Smaller than	7 < 6 (False)
<=	Smaller than or equal to	7 <= 6 (False)
==	Equals to	7 == 6 (False)
!=	Not equal to (different than)	7 != 6 (True)

```
type(5 > 7)
>>> <class 'bool'>
```

Boolean expressions

- `7 == 4` □ ?
- `(7 != 2) == (5 > 4)` □ ?
- `type(5 > 7) == type(8 < 3)` □ ?

Boolean expressions

- `7 == 4` □ False
- `(7 != 2) == (5 > 4)` □ True
- `type(5 > 7) == type(8 < 3)` □ True

Complex Boolean operators

- Take few Boolean operators and evaluate a new Boolean value from them.
  - `and` and `or` evaluate 2 Boolean expressions
  - `not` evaluates 1 Boolean expression
- The return value of complex Boolean operators could be represented in a **Truth table** – a table that lists al the combination of truth value of input variables and their evaluated output

Complex Boolean operators
Truth table

exp2	exp1	exp1 **and** exp2
T	T	T
F	T	F
T	F	F
F	F	F

exp2	exp1	exp1 **or** exp2
T	T	T
F	T	T
T	F	T
F	F	F

exp1	not(exp1)
T	F
F	T

Conditional operation

- We do not always want to execute all the lines in our code. Sometimes we want to execute some lines **only if** a certain condition is maintained.
- For example : Divide 9 by user's input.
  - We get the number from the user.
  - Only **if** the number is different than 0, we can divide 9 by it.

Conditional operation - if

- How do we implement this notion in Python?

```
if boolean_expression:
  #Code to perform if the
  #boolean_expression is True
  #(Note the indentation under the if
  #block).
```

Conditional operation - if

- For example :

```
num = float(input('Insert a number'))
if num != 0 :
   print(9/num)
```

- But what if the number does equal 0? We still want to let the user know.

Conditional operation - if

```
num = float(input('Insert a number'))
if num != 0 :
    print(9/num)
if num == 0 :
    print('Cannot divide by 0')
```

This is ***not a natural way to present our intention***. What we would usually say is : **if** the number is different than 0 divide, **else** print some message to the user.

Python lets us use such structure using the **else** keyword.

Conditional operation - else

```
num = float(input('Insert a number'))
if num != 0 :
    print(9/num)
else:
    print('Cannot divide by 0')
```

else should appear directly under an if block with the same indention.

Conditional operation - elif

And what if we had some **more options to choose from**?

If *condition1* then *result1,*

if not, than if *condition2* then *result2*

…

if not, than if *conditionN* then *resultN*

If none of the above then *result_Final*

Use **elif**! (=else if)

Conditional operation - elif

```
if now == 'Morning':
    print('Good morning!')
elif now == 'Noon':
    print('Good noon')
else:
    print('It must be evening')
```

1. The first elif should appear directly under an if block with the same indention.
2. As many elif's as you wish can follow.
3. elif can be terminated by a single else, or not at all.

Nested if

- What operations could be included inside an if block? Any operations we like :
  - print
  - input
  - … and – another if!
- An if inside another if is called **nested** if – it opens a new block with its own indentation.

Nested if - example

```
if now == 'morning':
    if 'y' == input('Are you hungry?'):
     print('Bon appetit!')
    else:
     print('Some other time than')
elif now == 'Noon':
    print('Good noon')
else:
    print('Good night')
```

Nested if - example

```
if now == 'morning':
    if 'y' == input('Are you hungry?'):
     print('Bon appetit!')
    else:
     print('Some other time than')
elif now == 'Noon':
    print('Good noon')
else:
    print('Good night')
```

This if is nested inside this one

`split()`

The method **split()** returns a list of all the words in the string, using a given string as the separator (default is whitespace)

```
# a = 'hello';  b = 'world'
>>> a,b = 'hello world'.split()

# a = 'hell'; b = 'w'; c = 'rld'
>>> a,b,c = 'hello world'.split('o')
```

Example

- Calculate the circumference (היקף) of a circle or square according to user request.
- Let's break the problem into parts :
  1. Get user input
  2. Validate if is it either a circle or a rectangle
     - If it is not print an error message and do not continue

  3(a). If it is a circle
     - Ask for the radius, calculate circumference

  3 (b). If it is a square
     - Ask for the side's length, calculate circumference

  4. Report to user the calculated result

Example – break it up into functions

- calculate_circle_ circumference()
- calculate_rectangle_ circumference()
- is_valid_shape_choice(choice)
- get_user_input()
- calculater_user_choice_circumference()
- error_safe_circumference()

Then call the function to run the program:

```
error_safe_circumference()
```

```
PI = 3.14

CHOICE_CIRCLE = 'C'
CHOICE_RECTANGLE = 'R'

MESSAGE_INPUT_REQUEST = 'Choose shape(C,R): '
MESSAGE_OUTPUT_REPORT = 'The circumference of the shape is: '
MESSGAE_INSTRUCTIONS = 'This program calculate the
circumference of either a circle or a square'
MESSGAE_RADIUS_REQUEST = 'Insert circle radius: '
MESSGAE_SIDE_REQUEST = 'Insert length of side: '

ERROR_NO_SUCH_SHAPE = 'No such shape'

def calculate_circle_circumference(): # runs 6’th (opt. 1)
    return 2*PI*float(input(MESSGAE_RADIUS_REQUEST))

def calculate_rectangle_circumference(): # runs 6’th (opt. 2)
    return 4*float(input(MESSGAE_SIDE_REQUEST))

def get_user_input(): # runs 3’rd
    print(MESSGAE_INSTRUCTIONS)
    return input(MESSAGE_INPUT_REQUEST)
```

```
def calculate_circumference(shape): # runs 5’th
    if shape == CHOICE_CIRCLE :
        return calculate_circle_circumference()
    elif shape == CHOICE_RECTANGLE:
        return calculate_rectangle_circumference()

def is_valid_shape_choice(choice): # runs 4’th
        return (choice == CHOICE_CIRCLE) or (choice ==
CHOICE_RECTANGLE)

def calculater_user_choice_circumference(): # runs 2’nd
    user_choice = get_user_input()
    if not is_valid_shape_choice(user_choice):
        return None
    else:
        circumference = calculate_circumference(user_choice)
        return circumference

def error_safe_circumference(): # this function runs 1’st
    circumference = calculater_user_choice_circumference()
    if circumference == None:
        print(ERROR_NO_SUCH_SHAPE)
    else:
        print(MESSAGE_OUTPUT_REPORT + str(circumference))
```

Summary

- Today we have learned :
  - How to use **variable**
  - What are **types** and how to **convert** between them
  - How to receive an **input** from a user
  - How to use **functions** which get input and **return** output
  - Conditional operation : **if, elif, else**