HO CHI MINH CITY UNIVERSITY OF TRANSPORT
FACULTY OF INFORMATION TECHNOLOGY
SOFTWARE ENGINEERING DEPARTMENT

CHAPTER 6
BUILT-IN FUNCTIONS AND MODULES

CONTENTS

Writing a Simple Program

- **Problem**: Calculating the area of a circle.
- **Algorithm:**
  1. Get the circle's radius from the user.
  2. Compute the area by applying the following formula:
     area = radius * radius * PI
  3. Display the result.
- **Important issues**:
  - How to read the radius from console?
  - How to use pi in library?
  - How to display result to console?

1. Functions

- A function is a group of statements that performs a specific task.
- **Functions** take **arguments** and return a **result**.
- Any programming language provides a library of functions that perform common operations.
- One of the best things about Python is the vast number of high-quality custom libraries that have been written for it.
- Some of these libraries are in the "standard library“. Others libraries can be easily added.
- Some **built-in functions** are always available in the Python interpreter. → Don’t have to import any modules to use these functions

Built-in Functions

Built-in Functions				
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()	

2. Getting Help

- The **help()** function is possibly the most important Python function
- When you're looking up a function, remember to pass in the name of the function itself

```
In [5]: help(round)
```

```
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negative.
```

3. Input/Output

Function	Description
print( )	Prints to a text stream or the console
input( )	Reads input from the console
format( )	Converts a value to a formatted representation
open()	Opens a file and returns a file object

Print to a text stream or the console

- The **print()** function prints the specified message to:
  - Console (Screen)
  - A text stream (file)
- Syntax:

print(objects,..., sep=' ', end='\n', file=sys.stdout, flush=False)

- Prints the **objects** to a stream, or to sys.stdout by default.
- Optional arguments:
  - **sep**: string inserted between values, default a space
  - **file**: a stream file, defaults to the current sys.stdout (screen).
  - **end**: string appended after the last value, default a newline (\n).
  - **flush**: specifying if the output is flushed (True) or buffered (False). Default is False

Print to a text stream or the console

In [3]:
```
a = 1
b = 2
# Print 4 objects
print("a = ", a, ", b = ", b)
```

```
a =  1 , b =  2
```

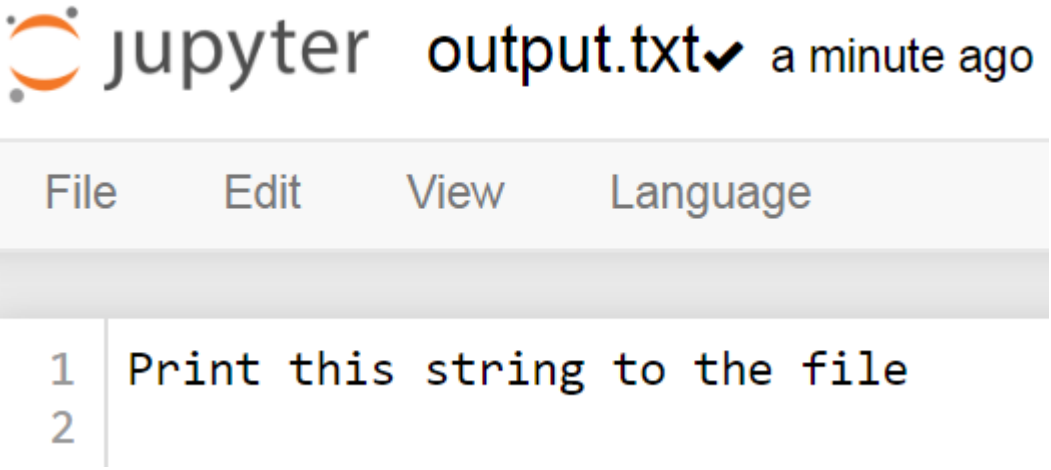
In [6]:
```
# Print with separator and end parameters
a = 1
b = 2
print(a,b,sep=',')
print("Hi!","How are you?",sep='\n',end='\n\n')
print("How old are you?",end='')
print("Bye!")
```

```
1,2
Hi!
How are you?

How old are you?Bye!
```

Print to a text stream or the console

In [1]:
```
# Print objects to the file
# Open file python.txt in writing mode
#If this file doesn't exist, new file is created
sourcefile = open('output.txt','w')
print('Print this string to the file', file=sourcefile)
sourcefile.close()
```



Print to a text stream or the console

```
In [2]: f = open('print_flush.txt', 'w')
        print('a', 'b', 'c', file=f)
```

At this time, there's nothing in the file. Print the contents of the file are stored in memory until the file is closed.

```
In [4]: f = open('print_flush.txt', 'w')
        print('a', 'b', 'c', file=f,flush=True)
```

At this time, contents have been written to the file.

Formatting Strings

- Python supports multiple ways to format text strings
  1. “Old Style” String Formatting (Modulo Operator)
  2. “New Style” String Formatting (str.format())
  3. f-Strings (Python 3.6+)

Formatting Strings

1. Using string modulo operator(%)

- Syntax

  <format_string> % (values)

- *<format_string>*: a string containing one or more **conversion specifiers**
- *values*: inserted into format_string in place of the conversion specifiers

```
In [3]: quantity = 5
        price = 30.75

        print('%d %s cost $%.2f' % (quantity, 'books', price))

        5 books cost $30.75
```

Formatting Strings

- **Note**:
  - String modulo operation isn't only for printing
  - We can also format values and assign them to another string variable

```
In [1]: name = input('Enter your name:')
        word = 'Hi! My name is %s' % name
        print(word)

Enter your name:Tien
Hi! My name is Tien
```

Formatting Strings

Conversion Specifiers

- Conversion specifiers appear in the <format_string> and determine how values are formatted when they're inserted.
- Syntax:

  %[<flags>][<width>][.<precision>]<type>

- *%* and <*type*> are required
- <flags>, <width>, <precision>: optional

Formatting Strings

Conversion Type

type	Conversion Type	Example
d, i, u	Decimal integer	
x, X	Hexadecimal integer	
o	Octal integer	
f, F	Floating point	
e, E	Exponential	
c	Single character	
s, r, a	String	

```
a = 250
print('Decimal:%d\tOctal:%o\tHex:%x'%(a,a,a))
```

```
Decimal:250     Octal:372       Hex:fa
```

```
print('%f, %F' % (3.14159, 3.14))
print('%e, %E' % (1000.0, 100.0))
```

```
3.141590, 3.140000
1.000000e+03, 1.000000E+02
```

```
print('%c \t %c' % (97,'a'))
print('%s \t %r \t %a' % ("Hi","Hi","Hi"))
```

```
a        a
Hi       'Hi'    'Hi'
```

Formatting Strings

Width and Precision Specifiers

- *<width>:* specifies the minimum width of the output field.

```
print( '%5d\n%05d\n%5d' % (123,123,123456))
```

```
  123
00123
123456
```

- *<precision>:* affects the floating point, exponential, and string conversion types

```
val = 123.456789
s = 'Python'
print('%.2f %.2e %.2s' % (val,val,s))
```

```
123.46 1.23e+02 Py
```

Formatting Strings

Conversion Flags

- Allow finer control over the display of certain conversion types

Character	Controls
#	Display of base or decimal point for integer and floating point values
0	Padding of values that are shorter than the specified field width
-	Value to be left-justified
+	Display of leading sign for numeric values

Formatting Strings

Conversion Flags

```
In [42]: print('%x %#x' % (16,16))
         print('%.0f %#.0f' % (123,123))
         print('%05d' % 123)
         print('%5d \n%-5d' % (123,123))
         print('%+d' % 3)
```

```
10 0x10
123 123.
00123
  123
123
+3
```

Formatting Strings

2. Using str.format() method

- Converts a value to a formatted representation
- Syntax

<format_string>.format(arguments)

- *<format_string>*: a string contain “replacement fields” surrounded by curly braces “{}”
- **Note**: If you need to print a brace character, it can be escaped by doubling: "{{" and "}}".
- *arguments*: inserted into format_string in place of the replacement fields

Formatting Strings

Replacement fields.

- Syntax:

  { [field_name] [! conversion] [: format_spec] }

- *field_name*
  - Specifies the object whose value is to be formatted.
  - May be number or keyword
  - Number: position of argument in format() function (0,1,2,…)
  - Keyword: name of argument in format() function
- *Conversion*
  - Force a type to be formatted as a string
  - Three conversion flags: !s, !r, !a

Formatting Strings

In [1]:
```
print('{0}, {1}, {2}'.format('a', 'b', 'c'))
# field names are omitted
# 0, 1, 2, … will be automatically inserted in that order
print('{}, {}, {}'.format('a', 'b', 'c'))
#re-arranging the order of display without changing the arguments.
print('{2}, {1}, {0}'.format('a', 'b', 'c'))
```

```
a, b, c
a, b, c
c, b, a
```

In [59]:
```
# field names are keywords
print('{h}:{m}:{s}'.format(h=7,m=30,s=20))
```

```
7:30:20
```

In [63]:
```
# Access attribute of argument
import math
print('{0.pi}'.format(math))
s = "abc"
print('{0[2]} and {text[2]}'.format(s,text=s))
```

```
3.141592653589793
c and c
```

Formatting Strings

Format Specification

- Syntax:

[[fill]align][sign][#][0][width][grouping_option][.precision][type]

Fill	<any character>
Align	<, >, ^: Left aligns, Right aligns, Center aligns = : Places the sign to the left most position
Sign	+, - : Use for positive or negative numbers Space: Use a leading space for positive numbers
#	Alternate form for different types
Width	A decimal integer defining the minimum field width
Grouping_option	, _: Use for a thousands separator
Precision	for a floating point value and string
Type	d, b, o, x/X: integer in base 10, 2, 8, 16 f/F, e/E, c, s: float, exponent, character, string

Formatting Strings

In [35]:
```
print('Left\tRight\tCenter')
print('{0:<05d} , {1:>05d}, {2:*^5d}, {3:=05d}'.format(6,6,6,-6))
```

```
Left    Right   Center
60000 , 00006, **6**, -0006
```

In [24]:
```
# Using the comma as a thousands separator
print('{:,}'.format(1234567890))
# Display values to different bases
print('Dec:{0:d}; hex:{0:#x}; oct:{0:#o}; bin:{0:#b}'.format(42))
# Specifying a sign
a = 3.51625
print('{:+.2f}; {:+.3f}'.format(a, -a))
```

```
1,234,567,890
Dec:42; hex:0x2a; oct:0o52; bin:0b101010
+3.52; -3.516
```

Formatting Strings

3. Using f-Strings

- Provide a way to embed expressions inside string literals.
- f-strings are string literals that are prefixed by the letter 'f' or 'F'
- Syntax

f ‘<format_string>’

{ Expression [! conversion] [: format_spec] }

- <*format_string*>: a string contain expressions inside braces “{}”
- *Expressions*: evaluated at run time, not a constant value.
- Following each expression, an optional type conversion or format specifier may be specified.
- F-strings use the same format specifier as str.format()

Formatting Strings

In [26]:
```
name = 'Lena'
age = 20
print('%s is %d years old'%(name,age))
print('{} is {} years old'.format(name,age))
print(f'{name} is {age} years old')
```

```
Lena is 20 years old
Lena is 20 years old
Lena is 20 years old
```

In [27]:
```
quantity = 5
cost = 30.75
print(f'Total cost of apples = {quantity*cost}')
```

```
Total cost of apples = 153.75
```

Formatting Strings

In [40]:
```
pi = 3.14159
print(f'pi = {pi:.2f}')
x,y,z = 2,5,10
print('x \t x^2 \t  x^3')
print(f'{x:02} \t {x*x:3} \t {x*x*x:4}')
print(f'{y:02} \t {y*y:3} \t {y*y*y:4}')
print(f'{z:02} \t {z*z:3} \t {z*z*z:4}')
```

```
pi = 3.14
x         x^2       x^3
02          4         8
05         25       125
10        100      1000
```

In [41]:
```
import datetime
name = 'Lena'
birthday = datetime.date(1999,3,12)
print(f'My name is {name} \nMy birthday is {birthday:%A,%d/%m/%Y}')
```

```
My name is Lena
My birthday is Friday,12/03/1999
```

Formatting DateTime

Directive	Meaning	Example
%a	Weekday as locale's abbreviated name.	Sun, Mon, …, Sat
%A	Weekday as locale's full name.	Sunday, Monday, …, Saturday
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, …, 6
%d	Day of the month as a zero-padded decimal number.	01, 02, …, 31
%b	Month as locale's abbreviated name.	Jan, Feb, …, Dec
%B	Month as locale's full name.	January, February, …, December
%m	Month as a zero-padded decimal number.	01, 02, …, 12
%y	Year without century as a zero-padded decimal number.	00, 01, …, 99
%Y	Year with century as a decimal number.	1970, 1988, 2001

Formatting DateTime

Directive	Meaning	Example
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, …, 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, …, 12
%p	Locale's equivalent of either AM or PM.	AM, PM
%M	Minute as a zero-padded decimal number.	00, 01, …, 59
%S	Second as a zero-padded decimal number.	00, 01, …, 59
%c	Locale's appropriate date and time representation.	Tue Aug 16 21:30:00 1988
%x	Locale's appropriate date representation.	08/16/88 (None); 08/16/1988 (en_US);
%j	Day of the year as a zero-padded decimal number.	001, 002, …, 366

Reading Input from the Console

- Reading input enables the program to accept input from the user.
  - From **Console (Keyboard)**
  - From File
- Use **input( )** function to read input from the Console.

  input([*<prompt>*])

  - *<prompt>:* optional if it is present, it displays message to standard output without a trailing newline before the input.
  - Read input as a **string** → need to convert the string to the appropriate type with the functions: int(), float(), complex(),… or **eval()**

Reading Input from the Console

```
In [1]: name = input('What is your name?')
        print(f'Hello {name}')
```

```
What is your name?Lena
Hello Lena
```

```
In [2]: num = input('Enter a number:')
        print(num + 1)
```

```
Enter a number:5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (mo
<ipython-input-2-556abc7d8358> in <module>
      1 num = input('Enter a number:')
----> 2 print(num + 1)

TypeError: can only concatenate str (not "int") to str
```

Reading Input from the Console

```
In [3]: num = int(input('Enter a number:'))
        print(num + 1)

        Enter a number:5
        6
```

- **Example**: Ask the user to input a value for the radius

```
s = input("Enter a value for radius: ") # Read input as a string
radius = eval(s) # Convert the string to a number
# Compute area
area = radius * radius * 3.14159
 # Display results
print("The area for the circle of radius", radius, "is", area)
```

```
Enter a value for radius: 2.5
The area for the circle of radius 2.5 is 19.6349375
```

Reading Input from the Console

Reading many values :

```
In [9]: a,b = eval(input('Enter two numbers:'))
        print(f'{a} + {b} = {a+b}')

        Enter two numbers:2,3
        2 + 3 = 5
```

Reading list of numbers:

```
In [16]: ds = eval(input('Enter a list of numbers separated by comma:'))
         print(f'List: {ds}')
         type(ds)

         Enter a list of numbers separated by comma:[3,-1,0]
         List: [3, -1, 0]

Out[16]: list
```

4. Math Functions

Function	Description
abs()	Returns absolute value of a number
divmod()	Returns quotient and remainder of integer division
max()	Returns the largest of the given arguments or items in an iterable
min()	Returns the smallest of the given arguments or items in an iterable
pow()	Raises a number to a power
round()	Rounds a floating-point value
sum()	Sums the items of an iterable

Math Functions

abs() Funtion

- Return the absolute value of a number
- Syntax: **abs(*x*)**
- *x*: required, may be an int or a float number. If the argument is a complex number, its magnitude is returned.

```
In [2]: x = abs(-3)
        print(x)
        y = abs(3+5j)
        print(y)
```

```
3
5.830951894845301
```

divmod() Funtion

- Return a pair of numbers consisting of their quotient and remainder
- Syntax:

  divmod(divident, divisor)
- *Divident and divisor*: required, may be an integer or a floating point number.

```
In [10]: x = divmod(5,2)
         print(x)
```

```
(2, 1)
```

Math Functions

min() Funtion

- Returns the smallest item in an iterable
- Syntax:

min(n1, n2, n3, ...)

min(iterable)

```
x = min("Mike", "Vicky","John")
print(x)
ds = [4,1,6,3]
print(min(ds))
```

```
John
1
```

max() Funtion

- Returns the largest item in an iterable
- Syntax:

max(n1, n2, n3, ...)

max(iterable)

```
x = max("Mike", "Vicky","John")
print(x)
ds = [4,1,6,3]
print(max(ds))
```

```
Vicky
6
```

Math Functions

pow() Funtion

- Return *x* to the power *y*, modulo z;
- Syntax:

pow(x, y[, z])

```
# same as (2*2*2)%5
x = pow(2,3,5)
print(x)
```

```
3
```

- *If z is present, x and y must be of integer types, and y must be non-negative.*

```
y = pow(2,0.5,2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-ebded0006ba7> in <module>
----> 1 y = pow(2,0.5,2)

TypeError: pow() 3rd argument not allowed unless all arguments are integers
```

Math Functions

round() Funtion

- Return number rounded to ndigits precision after the decimal point.
- Syntax:

  round(number[, ndigits])
- If *ndigits* is omitted or is None, it returns the nearest integer to its input

```
In [10]: n = 5.76543
         x = round(n, 2)
         print(x)
         y = round(n)
         print(y)

         5.77
         6
```

sum() Funtion

- Sums *start* and the items of an *iterable* from left to right and returns the total.
- *start* defaults to 0.
- Syntax:

  sum(iterable[, start])

```
In [14]: x = [3,1,2,4]
         print(sum(x))
         print(sum(x,1))

         10
         11
```

Type Conversion

Function	Description
ascii()	Returns a string containing a printable representation of an object
bin()	Converts an integer to a binary string
bool()	Converts an argument to a Boolean value
chr()	Returns string representation of character given by integer argument
complex()	Returns a complex number constructed from arguments
float()	Returns a floating-point object constructed from a number or string
hex()	Converts an integer to a hexadecimal string
int()	Returns an integer object constructed from a number or string
oct()	Converts an integer to an octal string
ord()	Returns integer representation of a character
repr()	Returns a string containing a printable representation of an object
str()	Returns a string version of an object
type()	Returns the type of an object or creates a new type object

5. Some commonly used functions

Function	Description
eval()	Evaluates a Python expression
len()	Returns the length of an object
range()	Generates a range of integer values
dir()	Returns a list of names in current local scope or a list of object attributes
id()	Returns the identity of an object

Some commonly used functions

eval() Funtion

- Syntax:

  eval(expression[, globals[, locals]])

- *Expression*: a string, is parsed and evaluated as a Python expression
- *Globals(Optional*): is a dictionary to specify the available global methods and variables.
- *Locals (Optional*): is a dictionary to specify the available local methods and variables
- Using globals and locals to make our **eval** function safe from any possible hacks.

Some commonly used functions

eval() Funtion

```
In [4]: x = 1
        y = eval('x + 1')
        print(y)
        print(type(y))

        2
        <class 'int'>
```

```
In [22]: from math import *
         a = eval('sqrt(5) + 1')
         print(a)

         3.23606797749979
```

If you pass an empty dictionary as globals, only the built-in functions are available to expression

```
In [13]: b = eval('pow(2,3)',{})
         print(b)

         8
```

Some commonly used functions

eval() Funtion

```
from math import *
a = eval('sqrt(5) + 1',{})
print(a)
```

```
---------------------------------------
NameError
<ipython-input-14-6fc3852a5649> in <mo
      1 from math import *
----> 2 a = eval('sqrt(5) + 1',{})
      3 print(a)

<string> in <module>

NameError: name 'sqrt' is not defined
```

Expression can use **sqrt()** methods and **pi** along with built-in functions

```
from math import *
c = eval('sqrt(5) + pi',{'sqrt':sqrt,'pi':pi})
print(c)
```

```
5.377660631089583
```

Some commonly used functions

len() Funtion

- Returns the number of items in an object.
- Syntax:

len(object)

- *Object*: may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

```
In [5]: s = "Python Programming"
        # returns the number of characters in the string
        print(len(s))
        ds = [3,1,4,6] # ds is list
        # returns the number of items in list
        print(len(ds))

        18
        4
```

Some commonly used functions

range() Funtion

- Returns a sequence of integers from start to stop by step.
- Syntax:

  range([start,] stop[, step])

- *start*: Optional. An integer number specifying at which position to start. Default is 0.
- *stop*: Required. An integer number specifying at which position to end.
- *step*: Optional. An integer number specifying the incrementation. Default is 1.
- Example:

  range(n) produces 0,1,2,…,n-1

  range(i, j) produces i, i+1, i+2, ..., j-1

Some commonly used functions

range() Funtion

```
ds = list(range(10))
print(ds)
ds = list(range(1,10))
print(ds)
ds = list(range(1,10,2))
print(ds)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 3, 5, 7, 9]
```

```
r = range(2,10,2)
print(f'Number of items in range is {len(r)}')
# get item in range using index
print(r[1])
# get index of item
print(r.index(4))
# check whether value 5 is in range ?
print(5 in r)
```

```
Number of items in range is 4
4
1
False
```

```
r = range(10)
for i in r:
    if i < 9:
        print(i,end=',')
    else:
        print(i)
```

```
0,1,2,3,4,5,6,7,8,9
```

```
r = range(10,-5,-2)
for i in r:
    if i > -5:
        print(i,end=',')
    else:
        print(i)
```

```
10,8,6,4,2,0,-2,-4,
```

Some commonly used functions

dir() Funtion

- Returns all properties and methods of the specified object.
- Syntax:

  dir([object])

- *If object is omitted, return the list of names in the current local scope.*

```
import math
print(dir(math))
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acos
h', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cos
h', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floo
r', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfini
te', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'mod
f', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'tau', 'trunc']
```

Some commonly used functions

dir() Funtion

```
class Person:
  name = "John"
  age = 36
  country = "Norway"

print(dir(Person))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__for
mat__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init
_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce_
_', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__sub
classhook__', '__weakref__', 'age', 'country', 'name']
```

```
print(dir(range))
```

```
['__bool__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__re
versed__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count',
'index', 'start', 'step', 'stop']
```

Some commonly used functions

id() Funtion

- Return the “identity” of an object. All objects in Python has its own unique id.
- This is the address of the object in memory and will be different for each time you run the program
- Syntax: **id(object)**
- *Object:* Any object such as String, Number, List, Class,....

```
In [62]: x,y = 257,257
         z = 'Python'
         print(id(x))
         print(id(y))
         print(id(z))
```

```
98012560
98012336
30990944
```

6. Working with External Libraries

- A module is a file containing variables and functions intended for use in other Python programs.
- **Module** contents are made available to the caller with the **import** statement
- There are many Python modules that come with Python as part of the standard library:
  - math
  - datetime, time
  - random
  - string
  - …

Working with External Libraries

- The **import** statement:

  import <module_name>

- To access variables and functions in the module, need to use <*module_name*> and *dot notation*.

```
In [1]: import math
        x = math.sqrt(5)
        print(x)

        2.23606797749979
```

- Import many modules in single statement:

  import <module_name>[, <module_name> ...]

Working with External Libraries

- If we know we'll be using functions in module frequently we can import it under a shorter alias:

import <module_name> as <alias>

```
In [4]: import math as mt
        mt.pi

Out[4]: 3.141592653589793
```

- Directly access to objects and functions without any dotted prefix:

from <module_name> import *

→ import everything from a module

```
In [6]: from math import *
        print(pi, "\n",log(32, 2))

3.141592653589793
 5.0
```

Working with External Libraries

- **Import *** isn't necessarily recommended in large-scale production code. It's a bit dangerous.

```
In [7]: from math import *
        from numpy import *
        print(pi, log(32, 2))
```

```
---------------------------------------------------------------
TypeError                                 Traceback
<ipython-input-7-5045b296ad83> in <module>
      1 from math import *
      2 from numpy import *
----> 3 print(pi, log(32, 2))

TypeError: return arrays must be of ArrayType
```

- The problem in this case is that the math and numpy modules both have functions called log, but they have different semantics.

→ log function in numpy overwrites the log function in math

Working with External Libraries

- Import only the specific things we'll need from each module:

from <module_name> import <name(s)>

```
In [9]: from math import log, pi
        from numpy import asarray
        print(pi,"\n",log(32, 2))
```

```
3.141592653589793
 5.0
```