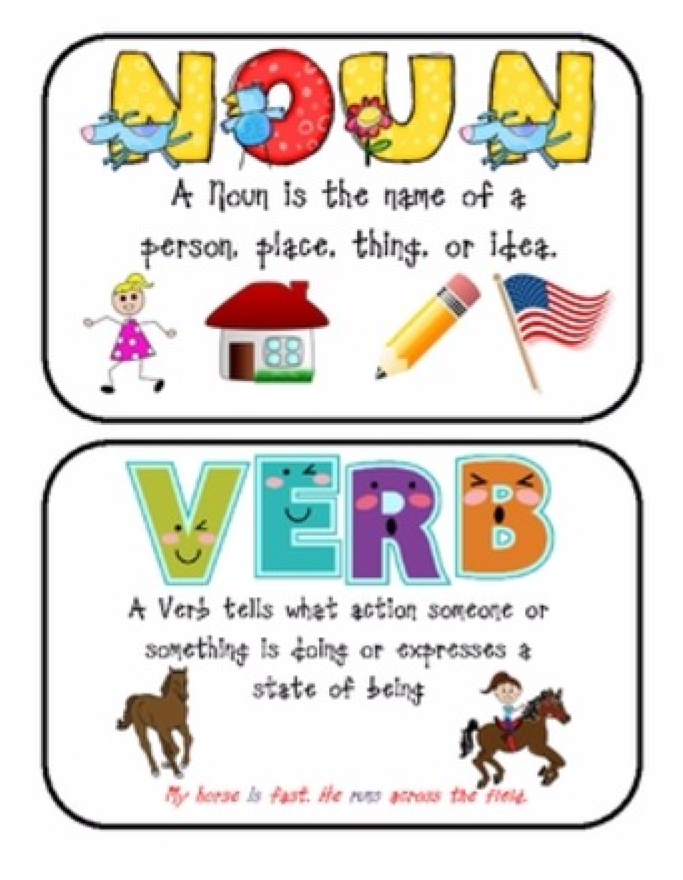
py4kids (https://github.com/wgong/py4kids)

Built-in functions (https://docs.python.org/3/library/functions.html)

In this lesson, we learn many useful built-in functions

Why function?

Function and Data Type in python are like Verb and Noun in English.



A Noun is the name of a person, place, thing, or idea.

A Verb tells what action someone or something is doing or expresses a state of being

My horse is fast. He runs across the field.

Data Types

that describe Object and Subject
(e.g. people, thing, place, time)

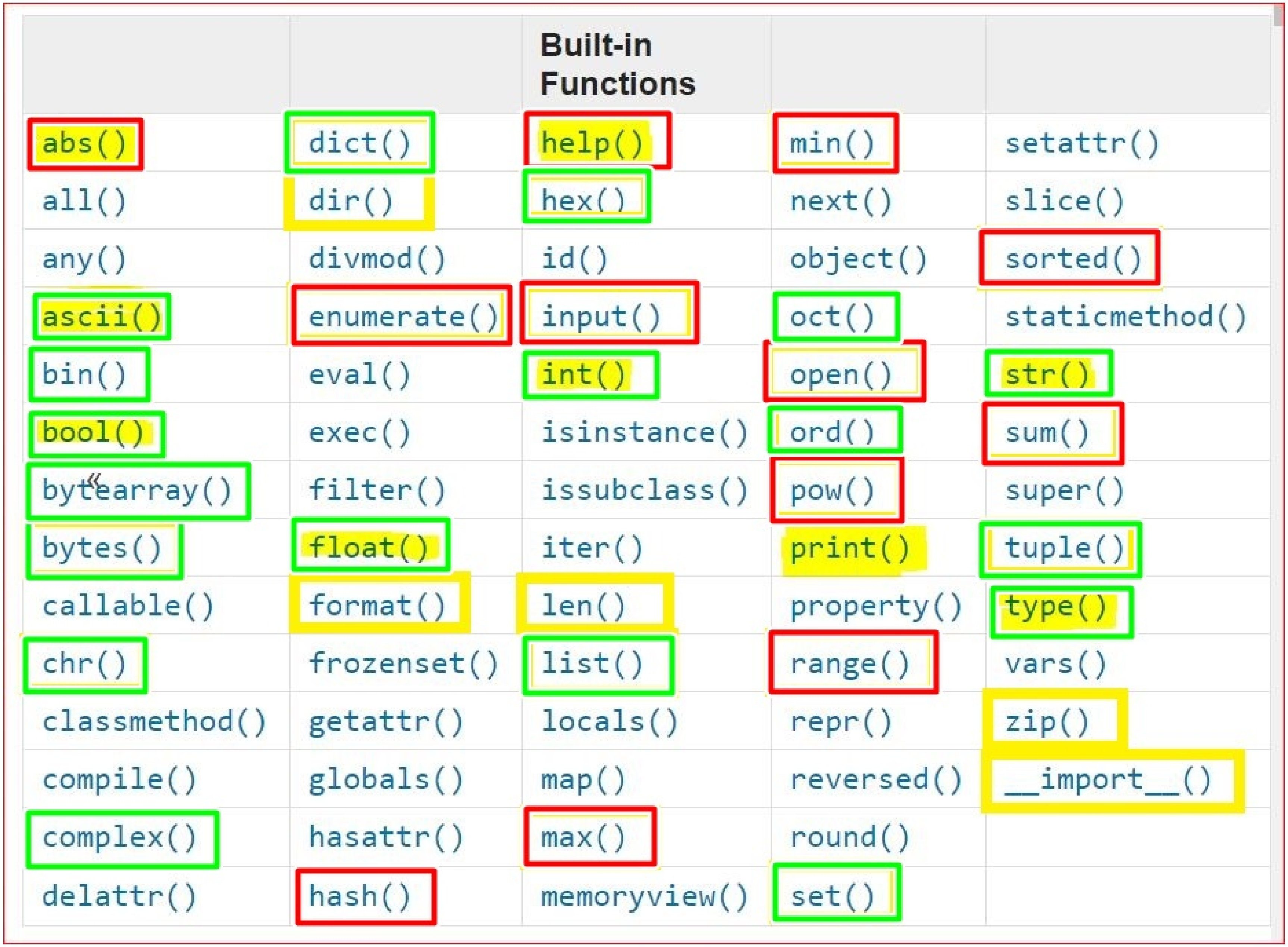
int, str, list, dict, ...

Functions

that describe actions, relations
(e.g. walk, own, grow, copy)

print, read, write, ...

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



		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	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()	
delattr()	hash()	memoryview()	set()	

- Help yourself
    - help()
- I/O - working with files
    - input()

    - open()
- Math - crunching numbers
    - range()
    - abs()
    - min()
    - max()
    - sum()
    - pow()
    - round()
- Useful Others
    - enumerate()
    - sorted()
    - reversed()
    - hash()
- Data Structure and conversion
    - ascii()
    - chr()
    - ord()
    - oct()
    - bin()
    - bool()
    - int()
    - float()
    - complex()
    - bytes()
    - bytearray()
    - str()
    - list()
    - tuple()
    - set()
    - dict()
    - type()

In [1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

Out[1]:

- Help yourself
- I/O - Input/Output
  - input() - talk to computer
  - open, read/write, close - work with files
- Math - crunching numbers
- Useful Others
  - enumerate
  - sorted
  - hash
- Data Structure and conversion

Help yourself

- read online documentation (https://docs.python.org/3/index.html)
- notebook inline help
- ask question at stackoverflow (https://stackoverflow.com/questions/415511/how-to-get-current-time-in-python)

In [2]:
```
# online help
help(print)
```

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

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
    
    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

In [3]:
```
print?
```

print shift tab

```
In [4]: help(input)
```

```
Help on method raw_input in module ipykernel.kernelbase:

raw_input(prompt='') method of ipykernel.ipkernel.IPythonKernel instance
    Forward raw_input to frontends

    Raises
    ------
    StdinNotImplentedError if active frontend doesn't support stdin.
```

```
In [5]: input?
```

ask stackoverflow:

[How to get current time in Python? (https://stackoverflow.com/questions/415511/how-to-get-current-time-in-python)](https://stackoverflow.com/questions/415511/how-to-get-current-time-in-python)

I/O - Input/Output

input() - talk to computer

```
In [9]: # get inputs from user
        your_name = input('What is your name?')
```

```
What is your name?allen
```

```
In [7]: your_age = input('What is your age?')
```

```
What is your age?13
```

```
In [8]: your_city = input('Which city are you from?')
```

```
Which city are you from?chapel hill
```

In [10]:

```
print(" name: %s\n age: %s\n city: %s"%(your_name, your_age, your_city))
```

```
 name: allen
 age: 13
 city: chapel hill
```

open, read/write, close - work with files

- read data from file
- write data to file

In [11]:

```
file_zen_python = '../data/zen-of-python.txt'
with open(file_zen_python, 'r') as f:
    text = f.read()
    print(text)
```

```
The Zen of Python
by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

In [12]:
```
f = open(file_zen_python, 'r')
for l in f:
    print(l)
f.close()
```

```
The Zen of Python

by Tim Peters



Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than right now.
```

```
If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!
```

In [13]:

```
for i in f:
    print(i)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-e5d63ea06a3b> in <module>()
----> 1 for i in f:
      2     print(i)

ValueError: I/O operation on closed file.
```

In [14]:

```
print(text)
```

```
The Zen of Python
by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

In [15]:

```
# add line number before each line
len(text)  # number chars
```

Out[15]:

```
855
```

In [16]:

```
words = text.split()   # number words
len(words)
```

Out[16]:

```
144
```

In [17]:

```
lines = text.split('\n')  # number lines
len(lines)
```

Out[17]:

```
24
```

In [18]:

```
n = 0
for i in lines:
    n = n + 1
    print("[%02d] %s" % (n, i))
```

```
[01] The Zen of Python
[02] by Tim Peters
[03]
[04] Beautiful is better than ugly.
[05] Explicit is better than implicit.
[06] Simple is better than complex.
[07] Complex is better than complicated.
[08] Flat is better than nested.
[09] Sparse is better than dense.
[10] Readability counts.
[11] Special cases aren't special enough to break the rules.
[12] Although practicality beats purity.
[13] Errors should never pass silently.
[14] Unless explicitly silenced.
[15] In the face of ambiguity, refuse the temptation to guess.
[16] There should be one-- and preferably only one --obvious way to do it.
[17] Although that way may not be obvious at first unless you're Dutch.
[18] Now is better than never.
[19] Although never is often better than right now.
[20] If the implementation is hard to explain, it's a bad idea.
[21] If the implementation is easy to explain, it may be a good idea.
[22] Namespaces are one honking great idea -- let's do more of those!
[23]
[24]
```

In [19]:

```
# write out to a file
filename = 'my-first-file.txt'
file_out = open(filename, 'w')
n = 0
for i in lines:
    n = n + 1
    file_out.write("[%02d] %s\n" % (n, i))
file_out.close()
```

Math - crunching numbers

```
In [20]: list_1 = list(range(10))
         list_1

Out[20]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [21]: # math operation on a list of numbers
         min(list_1), max(list_1), sum(list_1)

Out[21]: (0, 9, 45)
```

```
In [22]: float_1 = 2.12345
         print(round(float_1))

         2
```

```
In [23]: # what is 2 to the power of 10?
         pow(2,10)

Out[23]: 1024
```

Useful Others

enumerate

```
In [24]: list_2 = [100, -100, 21, 33, 10, 1000]
```

```
In [25]: # get the index number of a list
         enumerate(list_2)

Out[25]: <enumerate at 0x4c72318>
```

```
In [26]: for n,item in enumerate(list_2):
             print("n=%s, item=%s" % (n,item))
```

```
n=0, item=100
n=1, item=-100
n=2, item=21
n=3, item=33
n=4, item=10
n=5, item=1000
```

```
In [27]: set_1 = {1, 10, 100, 1000}
```

```
In [28]: for n,item in enumerate(set_1):
             print("n=%s, item=%s" % (n,item))
```

```
n=0, item=1000
n=1, item=1
n=2, item=10
n=3, item=100
```

sorted

```
In [29]: # sort a list
         ordered_list = sorted(list_2)
         ordered_list
```

```
Out[29]: [-100, 10, 21, 33, 100, 1000]
```

```
In [30]: rev_order_list = sorted(list_2,reverse=True)
         rev_order_list
```

```
Out[30]: [1000, 100, 33, 21, 10, -100]
```

```
In [31]: # did not change the original list
         list_2
```

```
Out[31]: [100, -100, 21, 33, 10, 1000]
```

In [32]:

```
# sort a list
ordered_list2 = reversed(list_2)
ordered_list2
```

Out[32]:

```
<list_reverseiterator at 0x4c9a470>
```

In [33]:

```
for i in ordered_list2:
    print(i)
```

```
1000
10
33
21
-100
100
```

hash

Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

use hash to compare two things quickly

In [34]:

```
number_1 = 123
number_2 = 1.23E2
cond_0 = number_1 == number_2
print(cond_0)
```

```
True
```

In [35]:

```
print(hash(number_1) == hash(number_2))
```

```
True
```

In [36]:

```
sentence_1 = "I like to watch movie"
sentence_2 = "i like to watch movie"
```

In [37]:
```
cond_1 = sentence_1 == sentence_2
print(cond_1)
```

```
False
```

In [38]:
```
cond_2 = hash(sentence_1) == hash(sentence_2)
print(hash(sentence_1), hash(sentence_2), cond_2)
```

```
186059012331224520 -4586553458247168616 False
```

In [39]:
```
print(text)
```

```
The Zen of Python
by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

In [40]:
```
print(hash(text))
```

```
-797979431730635856
```

Data Structure and conversion

In [41]:
```
# get binary representation of a decimal number
bin(2)
```

Out[41]:
```
'0b10'
```

In [42]:
```
bin(1024)
```

Out[42]:
```
'0b10000000000'
```

In [43]:
```
pow(2,10)
```

Out[43]:
```
1024
```

In [44]:
```
chr(65)   # decimal to char
```

Out[44]:
```
'A'
```

In [45]:
```
ord('A')  # char to decimal
```

Out[45]:
```
65
```

In [46]:
```
float('3.14159')
```

Out[46]:
```
3.14159
```

In [47]:
```
float('nan')
```

Out[47]:
```
nan
```

In [48]:
```
infinity_number = float('Infinity')
print(infinity_number)
```

```
inf
```

In [49]:
```
biggy_1 = float("9e99999")    # produce a big number
print(biggy_1)
```

```
inf
```

In [50]:
```
biggy_2 = float("9e999")    # produce a big number
print(biggy_2)
```
```
inf
```

In [51]:
```
biggy_1 - biggy_2
```
Out[51]:
```
nan
```

In [52]:
```
str(10000000)
```
Out[52]:
```
'10000000'
```

In [53]:
```
# bytes
s1 = "Hello World"

s1.encode()

bytes(s1, encoding='utf-8')
```
Out[53]:
```
b'Hello World'
```

In [54]:
```
s2 = '中国'
s2b = s2.encode(encoding='utf-8')
s2b
```
Out[54]:
```
b'\xe4\xb8\xad\xe5\x9b\xbd'
```

In [55]:
```
type(s2b)
```
Out[55]:
```
bytes
```

In [56]:
```
s2.encode(encoding='utf-16')
```
Out[56]:
```
b'\xff\xfe-N\xfdV'
```

In [57]:
```
tuple([1,2,3])
```
Out[57]:
```
(1, 2, 3)
```

```
In [58]: list("hello")

Out[58]: ['h', 'e', 'l', 'l', 'o']

In [59]: list(('a','e','i','o','u'))

Out[59]: ['a', 'e', 'i', 'o', 'u']

In [60]: dict(a=10,b=30,c='red')

Out[60]: {'a': 10, 'b': 30, 'c': 'red'}

In [ ]:

In [ ]:
```