My One Cheat Sheet

by Balan275 via cheatography.com/163857/cs/36262/

Variables

Variable is a container which stores values. But it cannot have reserved Keywords.

Rules for Variable Names:

* It must start with ‘letter’ or ’ _’

* It cannot start with number

* It can contain a-z,A-Z,0-9 and _

* Case sensitive. age and AGE different

Keywords

Python keywords are special reserved words that have specific meanings and purposes and can't be used for anything but those specific purposes. There are 35 Keywords in Python 3.

We can get List of Keywords by the following command in REPL.

```
>>> help("keywords")
```

To get details of each Keyword

```
>>> help("pass")
```

To get current version available keywords use below commands after import keyword in REPL with snippet >>> import keyword

```
>>> keyword.kwlist
```

['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

```
>>> len(keyword.kwlist)
```

35

```
>>> keyword.iskeyword('try')
```

Keywords (cont)

True

Lets See All Python Keywords with Usage

Value Keywords: True, False, None

```
>>> x = True
>>> x is True
```
```
True
```

```
>>> x = 'True'
>>> x is True
```
```
False
```

```
>>> y = False
>>> y is False
```
```
True
```

```
>>> x = 'False'
>>> x is False
```
```
False
```

```
>>> x = ''
>>> x is True
```
```
False
```

```
>>> x = "this is a truthy value"
>>> x is True
```
```
False
```

```
>>> x = "this is a truthy value"
>>> bool(x) is True
```
```
True
```

```
>>> y = "" # This is falsy
>>> y is False
```
```
False
```

```
>>> y = "" # This is falsy
>>> bool(y) is False
```
```
True
```

```
>>> x = "this is a truthy value"
>>> if x is True: print("x is True") # Don't do this
```

```
>>> if x: print("x is truthy") # Do this
```
```
x is truthy
```

```
>>> def func(): print("hello")
>>> x = func()
```
```
hello
```

Lets See All Python Keywords with Usage (cont)

```
>>> def func(): print("hello")
>>> x = func()
>>> print(x)
```
```
None
```

'None' is also the default value returned by a function if it doesn’t have a return statement:

Operator Keywords: and, or, not, in, is

Math Operator	Other Languages	Python Keyword
AND, ∧	&&	and
OR, ∨	||	or
NOT, ¬	!	not
CONTAINS, ∈		in
IDENTITY	===	is

Examples of and, or, not, in, is..

```
>>> x = True
>>> y = False
```

```
>>> x and y
```
```
False
```

```
>>> x and not y
```
```
True
```

```
>>> x or y
```
```
True
```

```
>>> not x or y
```
```
False
```

```
>>> x is y
```
```
False
```

```
>>> x is not y
```
```
True
```

```
>>> not x is y
```
```
True
```

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

Examples of and, or, not, in, is.. (cont)

```
>>> name = "Chad"
>>> "c" in name
```

```
False
```

```
>>> name = "Chad"
>>> "C" in name
```

```
True
```

Control Flow Keywords: if, elif, else

```
x = ""
y = ""
z = ""
if x: print('x')
elif y: print('y')
else: print('z')
```

```
z
```

```
x = ""
y = "Hi"
z = ""
if x: print('x')
elif y: print('y')
else: print('z')
```

```
y
```

```
x = "Hi"
y = ""
z = ""
if x: print('x')
elif y: print('y')
else: print('z')
```

```
x
```

Iteration Keywords:

Iteration Keywords: (cont)

Break Statement

```
nums = [1,2,3,4,5,6,7,8,9,10]
total_sum = 0
for num in nums:
    total_sum += num
    print(num)
    if total_sum > 10:
        break
print("Total Sum:",total_sum)
```

```
1
2
3
4
5
Total Sum:15
```

Continue Statement

```
nums = [1,2,3,4,5]
total_sum = 0
for num in nums:
    total_sum += num
    print(num)
    if total_sum > 3:
        continue
    print("Check")
print("Total Sum:",total_sum)
```

```
1
Check
2
Check
3
4
5
Total Sum: 15
```

The else Keyword Used With Loops

When else keyword used with a loop, the else keyword specifies code that should be run if the loop exits normally, meaning break was not called to exit the loop early.

```
for i in range(1,5,1):
    print(i)
else:
    print("Finished")
```

```
1
2
3
4
Finished
```

Iteration Keywords: (cont)

```
for n in range(2, 10):
    prime = True
    for x in range(2, n):
        if n % x == 0:
            prime = False
            print(f"{n} is not prime")
            break
    if prime:
        print(f"{n} is prime!")
```

```
2 is prime!
3 is prime!
4 is not prime
5 is prime!
```

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"{n} is not prime")
            break
    else:
        print(f"{n} is prime!")
```

```
2 is prime!
3 is prime!
4 is not prime
5 is prime!
```

Structure Keywords:

def, class, with, as, pass, lambda

The def Keyword

Python’s keyword **def** is used to define a function or method of a class.

The basic syntax for defining a function with **def** looks like this:

```
def <function>(<params>):
    <body>
```

Example

```
def func():
    print('Hi')
x = func()
```

```
Hi
```

The class Keyword

Classes are powerful tools in object-oriented programming, To define a class in Python, you use the **class** keyword.

for, while, break, continue, else

for Loop

```
>>> for num in range(1, 4):
        print(num)
```

```
1
2
3
```

```
>>> people = ["Kevin", "Creed", "Jim"]
>>> for person in people:
        print(f"{person} is in Office.")
```

```
Kevin is in Office.
Creed is in Office.
Jim is in Office.
```

while Loop

```
>>>n = 3
>>>while n>0:
        n=-1
        print(n)
```

```
2
1
0
```

```
for i in range(1,5,1):
    print(i)
    if i>3:
        break
else:
    print("Finished")
```

```
1
2
3
4
```

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

Structure Keywords: (cont)

The general syntax for defining a class with **class** is as follows:

```
class MyClass(<extends>):
    <body>
```

Example

The with Keyword

Using **with** gives you a way to define code to be executed within the context manager's scope. The most basic example of this is when you're working with file I/O in Python.

The general syntax for using **with** is as follows:

```
with <context manager> as <var>:
    <statements>
```

Context managers are a really helpful structure in Python. Each **context manager** executes specific code before and after the statements you specify.

If you wanted to open a file, do something with that file, and then make sure that the file was closed correctly, then you would use a **context manager**. Consider this example in which names.txt contains a list of names, one per line:

Example

```
with open("names.txt") as input_file:
    for name in input_file:
        print(name.strip())
```

```
Jim
Pam
Cece
```

The file **I/O context manager** provided by **open()** and initiated with the **with** keyword opens the file for reading, assigns the open file pointer to input_file, then executes whatever code you specify in the with block. Then, after the block is executed, the file pointer closes. Even if your code in the with block raises an exception, the file pointer would still close.

Structure Keywords: (cont)

The as Keyword Used With with

If you want access to the results of the **expression** or **context manager** passed to **with**, you'll need to alias it using **as**. You may have also seen **as** used to alias **imports and exceptions**, and this is no different. The alias is available in the with block:

The Basic syntax for using **as** is given below:

```
with <expr> as <alias>:
    <statements>
```

Most of the time, you'll see these two Python keywords, **with** and **as**, used together

The pass Keyword

Since Python doesn't have block indicators to specify the end of a block, the **pass** keyword is used to specify that the block is intentionally left blank. It's the equivalent of a **no-op**, or **no operation**.

Here are a few examples of using pass to specify that the block is blank:

```
def my_function():
    pass


class MyClass:
    pass


if True:
    pass
```

The lambda Keyword

The **lambda** keyword is used to define a function that doesn't have a name and has only one statement, the results of which are returned. Functions defined with lambda are referred to as **lambda functions**:

The Basic syntax of using **lambda** keyword given below:

```
lambda <args>: <statement>
```

Structure Keywords: (cont)

A basic example of a **lambda function** that computes the argument raised to the power of 10 would look like this:

```
p10 = lambda x: x**3
```

This is equivalent to defining a function with def:

```
def p10(x):
    return x**3
```

The above examples return the value **8**.

One common use for a **lambda** function is specifying a different behavior for another function. For example, imagine you wanted to sort a list of strings by their integer values. The default behavior of **sorted()** would sort the strings alphabetically. But with **sorted()**, you can specify which key the list should be sorted on.

A lambda function provides a nice way to do so::

```
>>> ids = ["id1", "id2", "id30", "id3", "id20", "id10"]
>>> sorted(ids)
['id1', 'id10', 'id2', 'id20', 'id3', 'id30']

>>> sorted(ids, key=lambda x: int(x[2:]))
['id1', 'id2', 'id3', 'id10', 'id20', 'id30']
```

This example sorts the list based not on alphabetical order but on the numerical order of the last characters of the strings after converting them to integers. Without lambda, you would have had to define a function, give it a name, and then pass it to sorted(). lambda made this code cleaner.

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

Returning Keywords: return, yield

There are two Python keywords used to specify what gets returned from functions or methods: return and yield. Understanding when and where to use return is vital to becoming a better Python programmer. The yield keyword is a more advanced feature of Python, but it can also be a useful tool to understand.

The return Keyword

Python’s **return** keyword is valid only as part of a function defined with def. When Python encounters this keyword, it will exit the function at that point and return the results of whatever comes after the **return** keyword:

Basic Syntax for using **return** given below:

```
def <function>():
    return <expr>
```

When given no expression, **return** will return None by default:

```
>>> def return_none():
            return

>>> return_none()
>>> r = return_none()
>>> print(r)
None
```

Most of the time, however, you want to return the results of an expression or a specific value:

```
>>> def plus_1(num):
            return num + 1

>>> plus_1(9)
10
>>> r = plus_1(9)
>>> print(r)
10
```

Returning Keywords: return, yield (cont)

You can even use the return keyword multiple times in a function. This allows you to have multiple exit points in your function. A classic example of when you would want to have multiple return statements is the following recursive solution to calculating factorial:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

In the factorial function above, there are two cases in which you would want to return from the function. The first is the base case, when the number is 1, and the second is the regular case, when you want to multiply the current number by the next number’s factorial value.

The yield Keyword

Python’s **yield** keyword is kind of like the return keyword in that it specifies what gets returned from a function. However, when a function has a yield statement, what gets returned is a **generator**. The generator can then be passed to Python’s built-in **next()** to get the next value returned from the function.

When you call a function with yield statements, Python executes the function until it reaches the first yield keyword and then returns a generator. These are known as generator functions:

Basic Syntax of using **yield** statement given below:

```
def <function>():
    yield <expr>
```

Returning Keywords: return, yield (cont)

The most straightforward example of this would be a generator function that returns the same set of values:

```
>>> def family():
            yield "Pam"
            yield "Jim"
            yield "Cece"
            yield "Philip"

>>> names = family()
>>> names
<generator object family at
0x7f47a43577d8>
>>> next(names)
'Pam'
>>> next(names)
'Jim'
>>> next(names)
'Cece'
>>> next(names)
'Philip'
>>> next(names)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Once the **StopIteration** exception is raised, the generator is done returning values. In order to go through the names again, you would need to call **family()** again and get a new generator. Most of the time, a generator function will be called as part of a **for** loop, which does the **next()** calls for you.

Test

Hi
 I am