



Python 講座

【第 2 回】 2015.05.12

きょうの話

3

リストと辞書

▼ ここから ▼

4

条件分岐（+真偽値）

5

繰り返し

6

関数



4

条件分岐（+ 真偽値）

if, elif, else and, or (+ True, False)

条件分岐？

- 「こういう場合にだけ実行する」という書き方
 - 「if 文」と呼んだりする
- **if** ★:
`space` ★が真の場合に実行する内容
- **elif** ☆:
`space` ★が偽、☆が真の場合に実行
- **elif** ◎:
`space` ★も☆も偽だが、◎が真の場合に(ry
...
- **else**:
`space` どの場合でもないときに実行

if 文の書き方 #1

条件「score の値 \geq 60」
後でたくさん紹介します。

実行してほしい処理の
行頭にスペースを入れる。
[TAB] キーが便利。
インデントと呼びます。

```
score = 74↵  
if score >= 60:↵  
    # 60点以上!  
    print('合格!')↵
```

合格!

elif と else は
片方もしくは両方とも
欠けてもよい。

if 文の書き方 #2

```
date = 28↵  
if date >= 21↵  
    print('下旬')↵  
elif date >= 11:↵  
    print('中旬')↵  
else: ↵  
    print('上旬')↵  
print('{}日'.format(date))↵
```

下旬
28日

上から順に、**最初に真と分かった処理**だけ実行される。
なので、11以上であるが「中旬」とは表示されない。

この行はインデントが無いので、条件分岐が終了済みとみなされる。

if 文の書き方 #3

```
score = 74  
if score >= 60:  
    print('合格!')  
    if score >= 80:  
        print('優')  
    elif score >= 70:  
        print('良')  
    else:  
        print('可')  
else:  
    print('不合格!')
```

合格!
良

if 文の処理内で、さらに if 文を使うこともできる。

if, elif, else を入れ子にするときは、常に親子関係を意識すると吉

親子関係を表現する役割を担うのが、インデント

処理のかたまりのことを、ブロックという

条件いろいろ

```
if x < y:
```

「 $x < y$ の場合」

```
if x <= y:
```

「 $x \leq y$ の場合」

```
if x > y:
```

「 $x > y$ の場合」

```
if x >= y:
```

「 $x \geq y$ の場合」

```
if x == y:
```

「 x と y が等しい場合」

```
if x != y:
```

「 $x \neq y$ の場合」

= の位置は
右側になる

比較する時は
2 個

! の部分を
思い出し辛い

if 文の書き方 #4

```
power_level = input('戦闘力: ')  
  
if power_level == 5:  
    print('「戦闘力...たったの5か...」')  
else:  
    print('「...」')
```

```
戦闘力: 5  
「...」
```

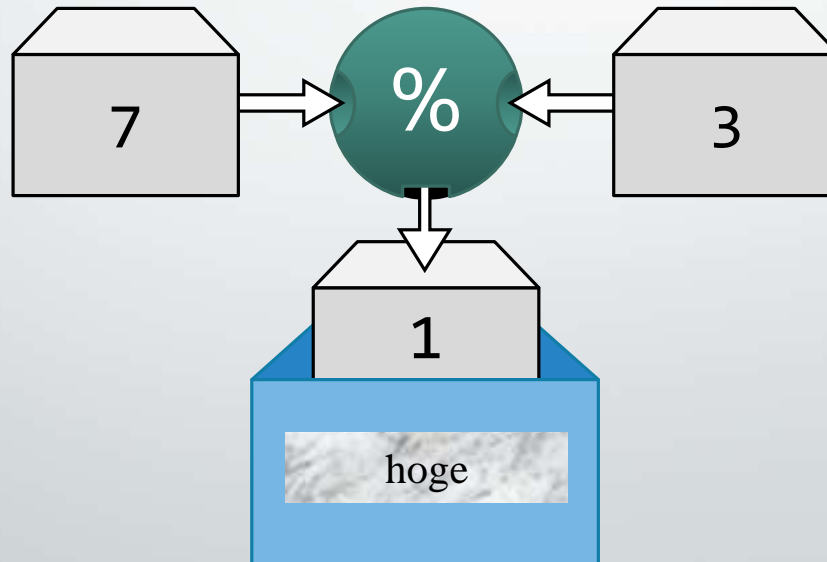
文字列と数字は別物扱い。
必要であれば、int 関数や
str 関数を使う。

そして真偽値へ

- 前置きとして、演算子の話。

```
hoge = 7 % 3 ↵  
print(hoge) ↵  
1
```

演算子は、近くの値に応じて値を計算して、「返す」。
%演算子なら、左を右で割った余りを返す。

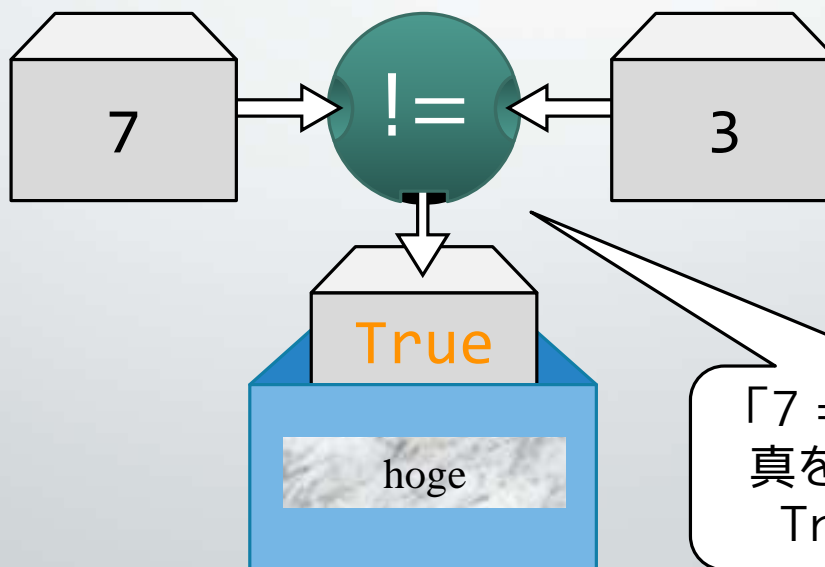


比較演算子

- `a == b` も近くの値に応じて計算している……

```
hoge = 7 != 3 ↵  
print(hoge) ↵  
True
```

さっきの `==` や `<` たちは実は演算子。
違うのは、返すのが数値ではなく、
真か偽を表す値であるところ。



「`7 ≠ 3`」は正しい。
真を表す値である
`True` を返す。

真偽値とは……

- 文字通り「真」と「偽」を表す値。
 - 「真」担当は True 。
 - 相方は「偽」担当の False 。
- 比較演算子 `<=` とか `!=` が返す値。
 - if 文に使っていたのは比較演算子の式
 - つまり if 文が本当に受け取っていたのは、True か False のどちらかの値
 - 受け取ったのが True なら実行する

(°Д°) ?

- 分からなくても死にはしません。
 - if 文の中で真偽値を使うことを意識する必要は、基本的には無いです。



条件と条件の組み合わせ

```
if ★ and ☆:
```

「★かつ☆の場合」

```
if ★ or ☆:
```

「★または☆の場合」

- 優先順位は `and > or`
 - コントロールはやっぱりカッコ () で
 - `if ★ or ☆ and ◆:`
= 「★」または「☆かつ◆」の場合
 - `if (★ or ☆) and ◆:`
= 「★または☆」かつ「◆」の場合

and の例

```
user_name = input('ユーザー名: ')↵  
password = input('パスワード: ')↵  
  
if user_name == 'root' and password == '1234':↵  
    print('ログイン成功!')↵  
else:↵  
    print('間違っています')↵
```

```
ユーザー名: root↵  
パスワード: 1234↵  
ログイン成功!
```

両方条件に一致すると実行される。

```
ユーザー名: admin↵  
パスワード: 1234↵  
間違っています
```

片方でも一致しないと実行されない。

or の例

```
month = int(input('何月生まれ? (1-12): '))  
  
if month < 1 or month > 12:  
    print('1から12の範囲で入力してください')  
...
```

```
何番目? (1-12): 0  
1 から 12 の範囲で入力してください
```

```
何番目? (1-12): 13  
1 から 12 の範囲で入力してください
```

```
何番目? (1-12): 2  
誕生日は「アメジスト」です
```

どちらかでも
一致すると実行

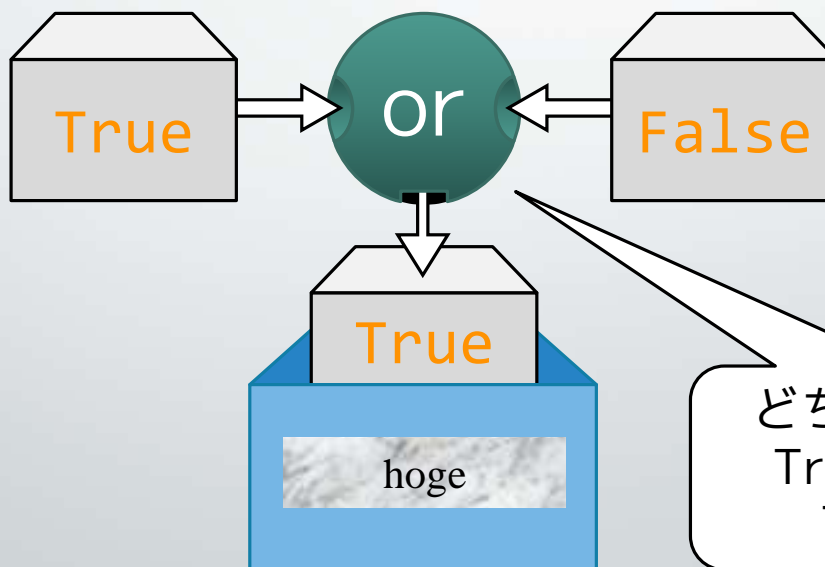
いずれでもない場合

論理演算子

- and や or も近くの値に応じて計算している……

```
hoge = True or False ↵  
print(hoge) ↵  
True
```

and や or も「論理演算子」というもの。隣にくるものと返すものは、基本的に真偽値。



どちらか一方でも Trueであれば、 Trueを返す

ここまで OK ?

- 条件分岐とは
 - 「こういう条件に一致したらこういう挙動をする」というプログラムの流れ
- 条件分岐の作り方
 - if ☆: elif ☆: else:
 - 「ブロック」を「インデント」で表現
- 条件いろいろ
 - 未満、以下、より大きい、以上、等しい、異なる
 - and / or



5

繰り返し

while と for の話、break と continue の話

繰り返し？

- コードの一部を繰り返し実行したい時の書き方。大きく分けて 2 種類
- A) 要素の集まり（リスト等）に対して**先頭の要素から最後まで**繰り返す
 - Python における for 文
- B) **条件が満たされる間ずっと**繰り返す
 - while 文

先頭から最後まで・for

- `for` <変数名> `in` <要素の集まり>:
 `space` (繰り返す処理)

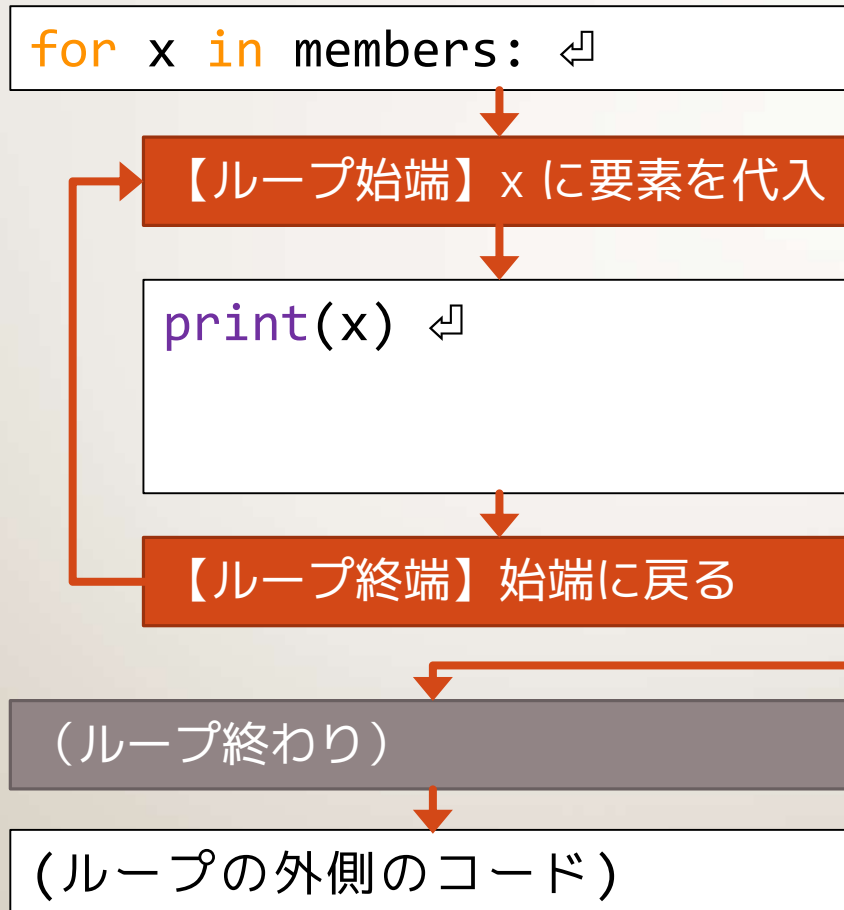
```
members = ['ムサシ', 'コジロウ', 'ニャース'] ↵
```

```
for x in members: ↵  
    print(x) ↵
```

```
ムサシ  
コジロウ  
ニャース
```

変数 `x` に 1 個放り込み、`print` して、
次の要素を放り込み、`print` して、
また次の要素を放り込み…… の繰り返し。

for



赤色の部分を
Python が勝手に
実行してくれる。

要素の代入が
できないとき、
ループ終わりに
移動する。

指定回数だけ・for

- 指定回数だけ繰り返すには、その回数分の要素が詰まった集まりを作って、それをforに指定すればよい
- 例えば、0, 1, 2, ..., nのような連番が詰まったものが作ればよい
 - range 関数に任せる！

連番を詰め込んで返す・range

- `range(★, ☆)`
 - range 関数。★ から ☆-1 までの整数を順に詰め込んだリストっぽいのを返す。
 - ★ …… 開始値
 - ☆ …… 終了値 +1

```
for x in range(0, 3): ↵  
    print(x) ↵
```



```
0  
1  
2
```

0、1、2 の詰まったリストのようなものを for に指定してあげた例。

x に 0 を放り込んで print、次は 1 を、……という具合。

☆ = (終了値 + 1) = 3 なので、格納されるのは 2 まで。

range 関数の話

- 増分を指定できます。すなわち、3 ずつ増やすなど。
 - 第 3 引数に指定。省略すると 1。

```
for x in range(0, 10, 3):  
    print(x)
```



0
3
6
9

- リストっぽいものとは何ぞ？
 - 実は「範囲」を表すデータを持っています。
 - リストが「0 番目の要素は 0」「2 番目の要素は 1」という風に独立したデータを持つのに対して、
 - 範囲 (range) オブジェクトは「開始値」「終了値+1」「増分」の 3 データだけを持ちます。そして、「n 番目の要素は 開始値 + n × 増分」という風に逐一計算して求めています。省メモリです。素敵。

条件を満たす限り・while

- `while` <条件> :
(繰り返す処理)

```
name = ''  
while name == '' :      # 条件「name が空」 ↵  
    name = input('きみの なまえは? ') ↵  
    print('ふむ.....') ↵  
print(name + ' というんだな!') ↵
```

```
きみの なまえは? ↵  
ふむ.....  
きみの なまえは? わしにしね↵  
ふむ.....  
わしにしね というんだな!
```

name 変数が空であれば、
ループに突入 & 継続する。

while

```
while name == '': ↵
```

【ループ始端】条件を満たすか確認

```
name = input('なまえは?') ↵  
print('ふむ.....') ↵
```

【ループ終端】始端に戻る

(ループ終わり)

```
print(name + 'というんだな!') ↵
```

赤色の部分を
Python が勝手に実行
してくれる。

条件を満たされない時、
ループ終わりに移動。

条件チェックは始端で
のみ行われる。

ここまで OK ?

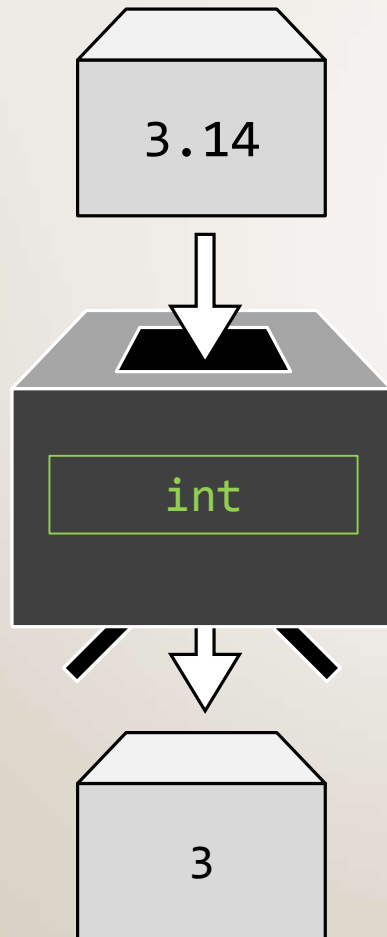
- ループというもの
 - 「先頭から最後まで繰り返す」という処理・for
 - for x in member:
 - 指定回数だけ繰り返すには range 関数を使うと便利
 - 「こういう条件の間繰り返す」という処理・while
 - while name == "":



関数

何かを入れると、処理をして、必要なら何かを返すもの

関数の例 (1/2)

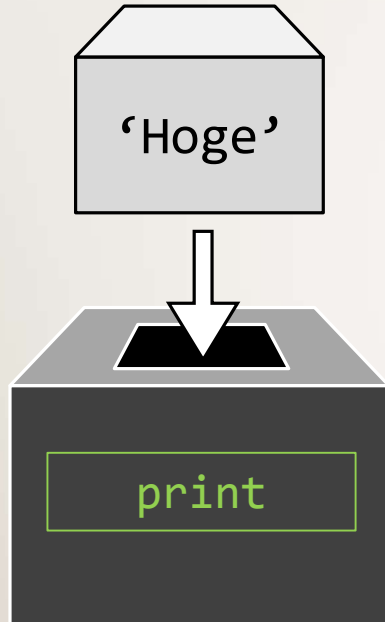


- 例えば `int` 関数。
 - 実数を入れると、少数を切り捨て、それを整数として返すもの。
- 数学の“関数”に似てるのでこう呼ばれる。
 - 関数 $f(x)$ の x に値を入れると結果が y に入る……みたいなもの。

```
int(3.14) ↵  
3
```

こんな感じに
呼び出すのが
関数。

関数の例 (2/2)



```
print('Hoge')  
Hoge
```

- 例えば `print` 関数。
 - 何かを入れるとそれをディスプレイに表示するもの。
- 何も返してないけど……
 - Python では何も返さないものもひっくるめて「関数」と呼ぶ。

```
a = print('fuga')  
Hoge  
print(a)  
None
```

None = 「何もない」。
変数 (箱) だけの状態。

関数を使うことのメリット

- 関数があることで……
 - 使うときに小難しいことを考えないで良くなる
 - ただ「整数にして返してくれる」「画面に表示してくれる」ことだけ知っていればよい
 - 意図が伝わりやすい
 - 長ったらしいコードだとどんな処理をさせたいのか理解しづらい
 - タイピング量を減らせる場合が多い
 - 使う頻度が高ければ高いほど！

関数が無いことのデメリット

- もしも `int` 関数がなかったら

もしも組み込み関数の `int` 関数が使えない場合、整数の文字列を変換する時こんなようなコードを毎回書く必要がある。まず、対象となる文字列に対して `for` をかけて1文字ずつ切りだす。その1文字ごとに0で初期化された結果格納用の変数を10倍してアスキーコードを基にどの数字かを計算して加算、というのを行う。この書き方、バイトで書ける `int` 関数 | うるさい黙れ | 3行のコードが必要なので、数担が少ない。 `ord` 関数や `print` ..

長い!

```
TARGET = "123"
RESULT = 0
ZERO = ord('0')
for c in TARGET:
    RESULT *= 10
    RESULT += ord(c) - ZERO
print(RESULT)
```

or

```
TARGET = '123' ↵
RESULT = int(TARGET) ↵
print(RESULT)
```

得られる結果は全く同じ

関数を作ろう

- 関数は自分で作れます！
 - 今回一緒に作る関数は……
- `input_int(★)`
 - `input_int` 関数。ユーザーからの入力を「数値として」受け取る。
 - ★ …… 入力を促す文字列

Before

```
year = int(input('何年?')) ↵
```

After

```
year = input_int('何年?') ↵
```

関数の作り方

- `def <関数名>(引数名1, 引数名2, ...):`
 `space` (関数の処理)

▼ここまで書こう

```
def input_int(): ↵  
    pass ↵
```

pass とは、
「何もしない」という処理。
※詳細は次項

```
# 呼び出しテスト  
input_int('年齢は?')↵
```

- 今回は `input_int` という関数名
 - 分かりやすければ何でもOK。
`int` は整数 (integer) の略語。

pass?

- 「何もしない」という処理。
 - **今は**何も書きたくないけど、何か処理を書かないとエラーになる……等のときに使う。

```
if x >= 0:  
    # 後で書く  
    pass  
else:  
    print('負数です')
```

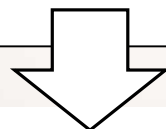
```
if x >= 0:  
    # 何も無いとエラー  
else:  
    print('負数です')
```

- pass は一時的に必要なときだけ！
 - 例えば、上のコードを「負数の時にだけ何かしたい」という意図の下に書くのは間違い。if の条件を $x < 0$ にすれば、pass は必要ありません。

試しに呼び出してみる

▼ここまで書いて実行

```
def input_int(): ↵  
    pass ↵  
  
input_int('年齢を入れてね') ↵
```



```
Traceback (most recent call last):  
  File "D:/Storage/!Uncategorized/hoge.py", line 4, in <module>  
    input_int('年齢を入れてね')  
TypeError: input_int() takes 0 positional arguments but 1 was given  
>>> |
```

input_int 関数を 1 個の引数とともに呼び出したけど、引数を受け取るなんて聞いていませんよ、というエラー。

今から受け取れるようにコードを追加します。

引数を受け取れるようにする

- `def <関数名>(引数名1, 引数名2, ...):`
 space (関数の処理)

▼ここまで書こう

```
def input_int(msg): ↵  
    pass  
  
# 呼び出し & 表示用  
age = input_int('年齢は? ') ↵  
print(age)
```

- 関数名の後ろのカッコ ()
の中に、受け取りたい引
数を羅列していく
 - 今回は msg としておく
 - 分かりやすい名前をつけて
あげると吉

input_int 関数を作る

▼ここまで書こう

```
def input_int(msg): ↵
    # 入力受けとり
    typed_str = input(msg)↵
    # 整数に変換
    number = int(typed_str)↵

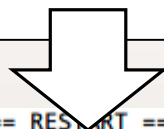
# 呼び出し & 表示用
age = input_int('年齢は? ') ↵
print(age)
```

- 関数の中でも、書くコードはいつも通り
 - 変数を作って中をいじれる
 - 別の関数も呼び出せる
 - input とか、int とか、あるいは自分で作ったほかの関数も
- 引数は、関数内ではただの変数
 - input_int の実行時、勝手に msg = '年齢は?' という風に代入されるようなイメージ

試しに呼び出してみる

▼ここまで書いて実行

```
def input_int(msg):  
    # 入力受けとり  
    typed_str = input(msg)  
    # 整数に変換  
    number = int(typed_str)  
  
    # 呼び出し & 表示用  
age = input_int('年齢は? ')  
print(age)
```



```
>>> ===== RESULT =====  
>>>  
年齢は? 19  
None  
>
```

変数の中には
何も入って
いない (None)

中で呼び出す input 関数が入力を促してくれています。
中では入力を受け取って変換まではしていますが、
まだそれを返す処理を書いていません。

input_int 関数に返させる

▼ここまで書こう

```
def input_int(msg): ↵
    # 入力受けとり
    typed_str = input(msg)↵
    # 整数に変換
    number = int(typed_str)↵

    # 変換した整数を返す
    return number ↵

# 呼び出し & 表示用
age = input_int('年齢は? ') ↵
print(age)
```

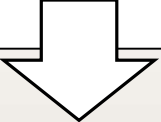
● return ★

- ★ … 関数が返すもの。
- return に到達した時点で、関数の処理は終了する。
- ★ を省略して、単に「関数の処理を終了」という風な書き方もできる。何も返さないのと同じ。

できあがり

▼ここまで書いて実行

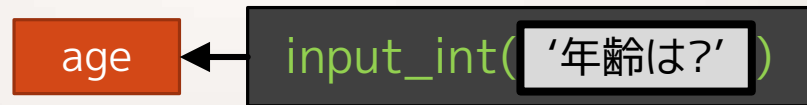
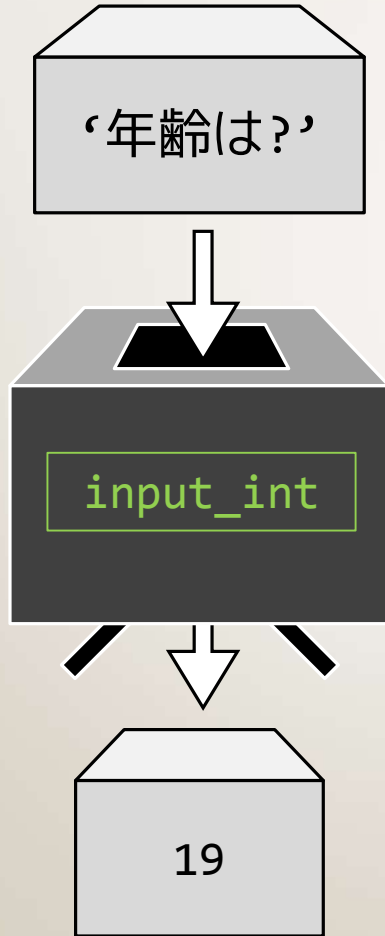
```
def input_int(msg):  
    # 入力受けとり  
    typed_str = input(msg)  
    # 整数に変換  
    number = int(typed_str)  
    # 変換した整数を返す  
    return number  
  
# 呼び出し & 表示用  
age = input_int('年齢は? ')  
print(age)
```



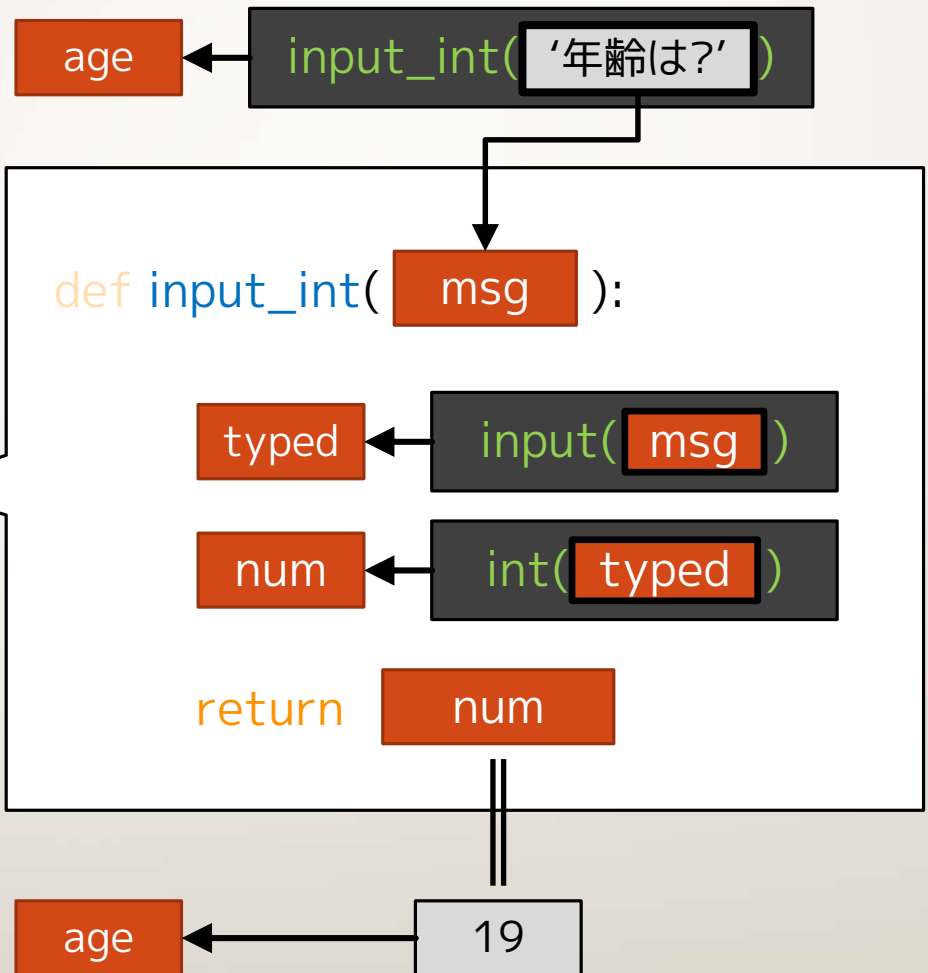
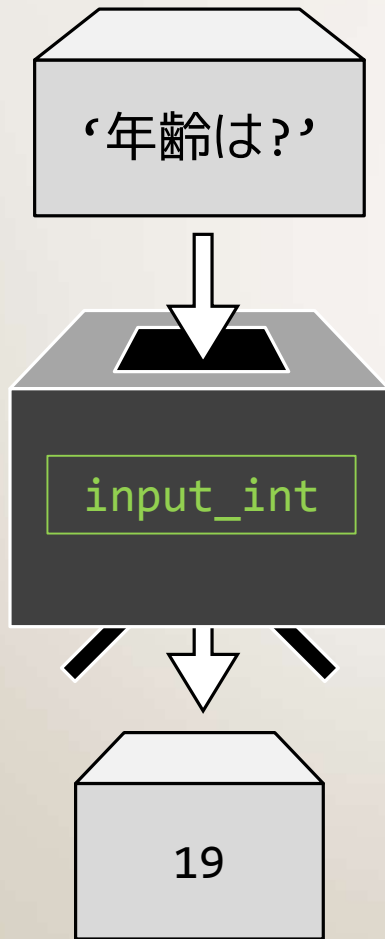
```
>>>  
年齢は? 19  
19  
>>>
```

age の中には無事数値としての 19 が入りました！これでひとまず input_int 関数はできあがり。

input_int



input_int



ここまで OK ?

- 関数とは
 - 何か（引数）を入れると、処理をして、必要なら何か（戻り値）を返すもの
- 関数の作り方
 1. `def <関数名>(引数名1, 引数名2, ……):`
 2. 処理をいつも通りに書く
 3. 必要なら `return` ★
(★は省略してもいい)



演習 #4~6

Enjoy.

演習 #4

- ユーザーが時刻を入力すると、それが 0:00 ~ 23:59 の範囲内であることを確認するプログラムを作成してください。
- ただし、ユーザーは必ず整数を入力します。

時： 23↵
分： 59↵
正しい時刻

時： 0↵
分： 0↵
正しい時刻

時： -31↵
分： 77↵
おかしい時刻

演習 #5

- 1 から 100 までの数字を表示するプログラムを作成してください。
ただし、3 の倍数の時は Fizz を、5 の倍数の時には Buzz を、3 と 5 の両方の倍数の時には FizzBuzz を、数字の代わりに表示してください。
 - 「FizzBuzz 問題」という、プログラミングに関する有名な問題です。（2 分以内に解けたらえらい）

```
>>>  
1  
2  
Fizz  
4  
Buzz  
Fizz  
7
```

```
13  
14  
FizzBuzz  
16  
17  
Fizz  
19  
Buzz
```



時間を測るなら……
→ [Online Stopwatch](#)

2 年生以上や自信のある
1 年生の皆さんはれっつ
たいむあたっく

演習 #6

- 定価と税率（%）を引数に受け取ると、税込価格を返す関数 `get_price_with_tax` を作ってください。
 - 返す税込み価格は整数（小数以下切捨）にしてください。

```
# ▼コピー▼  
  
#  
# 【ここに関数 get_price_with_tax を作ってください】  
#  
  
# 動作確認用  
price_2014 = get_price_with_tax(100, 5)      # 100円5%  
price_2015 = get_price_with_tax(100, 8)      # 100円8%  
print(price_2014)  
print(price_2015)  
  
# △ここまで△
```

ヒント

- 演習 #4

- きました、整数の入力です！今作った関数の出番です！
- 時は0以上24未満、分は0以上60未満の範囲です。この範囲外になると、おかしい時刻です。

- 演習 #5

- 繰り返しを使います。繰り返す回数が決まっていることを念頭に置くとシンプルなコードが書けます。
- 倍数かどうかを調べるのに有効な演算子があります。思い出せなければ、資料を見返してみてください。

- 演習 #6

- 価格と税率（%）を入れると、税込み価格を計算して、それを返す関数です。
- やり方はいろいろあります。短くすれば1行で済みます。
- 計算の順序だけ注意。

次スライドは解答です！
できるだけ自力で解いてみよう



演習 #4 ・ 解答例

```
def input_int(msg):  
    typed_str = input(msg)  
    number = int(typed_str)  
    return number  
  
hour = input_int('時: ')  
minute = input_int('分: ')  
  
if 0 <= hour and hour < 24 and 0 <= minute and minute < 60:  
    print('正しい時刻')  
else:  
    print('おかしい時刻')
```

1. input_int をつくる
2. 時と分を入力してもらう
3. 「時が 0 以上」かつ「時が 24 未満」かつ
「分が 0 以上」かつ「分が 60 未満」なら
正しい時刻と表示する

演習 #5 ・ 解答例

```
for i in range(1, 101):
    if i % 15 == 0:
        print('FizzBuzz')
    elif i % 3 == 0:
        print('Fizz')
    elif i % 5 == 0:
        print('Buzz')
    else:
        print(i)
```

- 妙な数字が出ていないか？
- 最初に「1」が、最後に 100 番目として「Buzz」が出ているか？
- 「Fizz」「Buzz」「FizzBuzz」の3種がちゃんと出ているか？

1. for を使って 100 回のループを実現するべく、1 から 100 が詰まったリストのようなものを range 関数で用意する。
2. それを変数 i に放り込んでいき、適当な文字列を表示する。

演習 #6 ・ 解答例

```
# ▼コピー▼  
  
def get_price_with_tax(price, tax):  
    return int(price * (1 + tax / 100))  
  
# 動作確認用  
umai_stick_2014 = get_price_with_tax(100, 5) # 100円+5%  
umai_stick_2015 = get_price_with_tax(100, 8) # 100円+8%  
print(umai_stick_2014)  
print(umai_stick_2015)  
# △ここまで△
```

「tax を 100 で割った値に 1 を足した値」と
定価の積を求めて、それを整数にして返す。
※ 税率は 8 や 5 など、% 単位の数値です