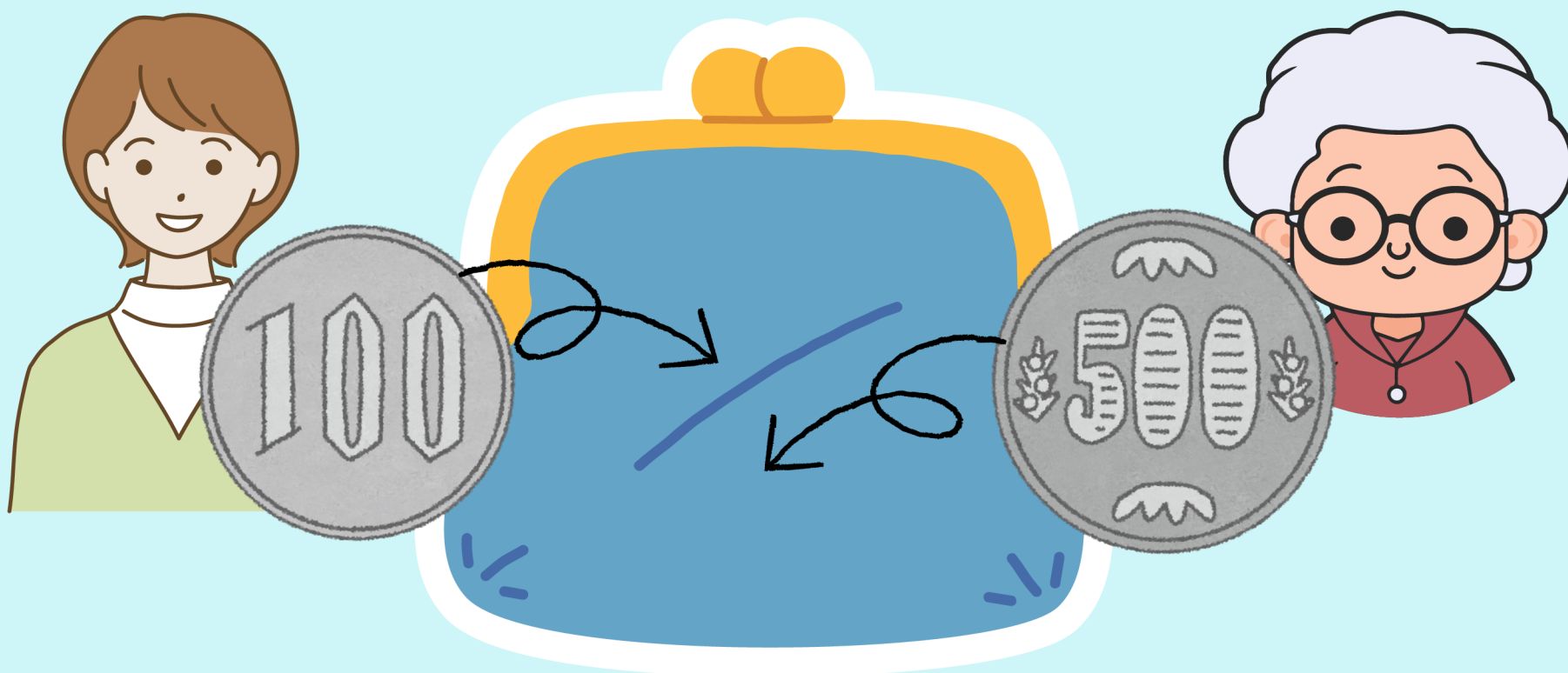


その 1. 変数 → お財布

プログラムが使えるお財布みたいなものですよ



変数名は、「osaifu」とします

難しく
考えないでね～

※下記のコードはイメージです

- ① お財布の中身は空です
コード例： $\text{osaifu} = 0$
- ② お母さんからもらった100円を入れました
コード例： $\text{osaifu} = \text{osaifu} + 100$
お財布の中身は、100円になります
- ③ おばあちゃんからもらった500円を入れました
コード例： $\text{osaifu} = \text{osaifu} + 500$
お財布の中身は、600円になります

ここがPoint♥

お財布は1個だけど、
中身を増やす事や減らすことが自由にできます
この考え方が「変数」の基本だよ♥



プログラミングのツボ

その 1-1. 変数（リスト）

→ 同じお財布がいくつもある場合



変数名は、「osaifu」とします

※下記のコードはイメージです

難しく
考えないでね～



①全部のお財布の中身を空にします

```
osaifu = []
```

②それぞれのお財布にお金を入れます

```
osaifu.append(100)
```

```
osaifu.append(500)
```

```
osaifu.append(1000)
```

2つめのお財布の中身は、

```
okane = osaifu[1] ←0番から始まるよ
```

③合計の計算はカンタンだよ ←sum()を使うよ

```
gokei = sum(osaifu)
```

ここがPoint♥

同じお財布がいくつもある場合には、

お財布の番号（0～）で中身が出入れできるよ

この考え方が「変数（リスト）」の基本だよ♥



その 1-2. 変数 (ディクショナリ : 辞書)

→ それぞれのお財布名前 (索引) がついている場合



変数名は、「osaifu」とします

※下記のコードはイメージです

難しく
考えないでね～

① 全部のお財布の中身を空にします

`osaifu = {}` ← {} だよ

② それぞれのお財布にお金を入れます

`osaifu["黄"] = 100`

`osaifu["青"] = 500`

`osaifu["緑"] = 1000`

「青」のお財布の中身は、

`okane = osaifu["青"]` ← 名前を指定

③ 合計の計算はカンタンだよ ← `sum()` を使うよ

`gokei = sum(osaifu.values())`

ここがPoint♥

同じお財布がいくつもある場合には、

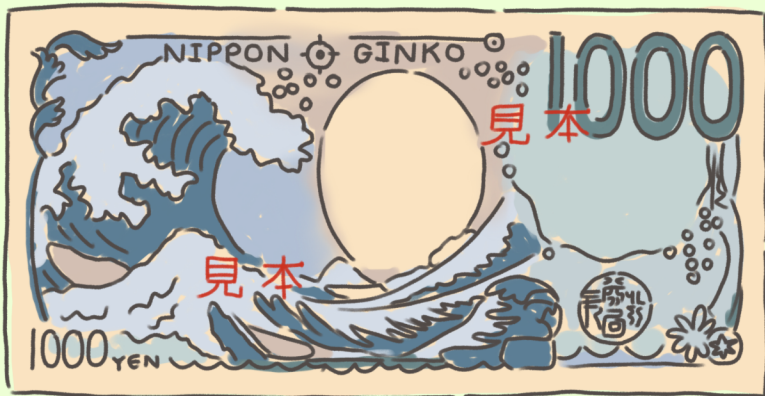
お財布の名前 (索引) で中身が出入れできるよ

この考え方が「変数 (ディクショナリ)」の基本だよ♥

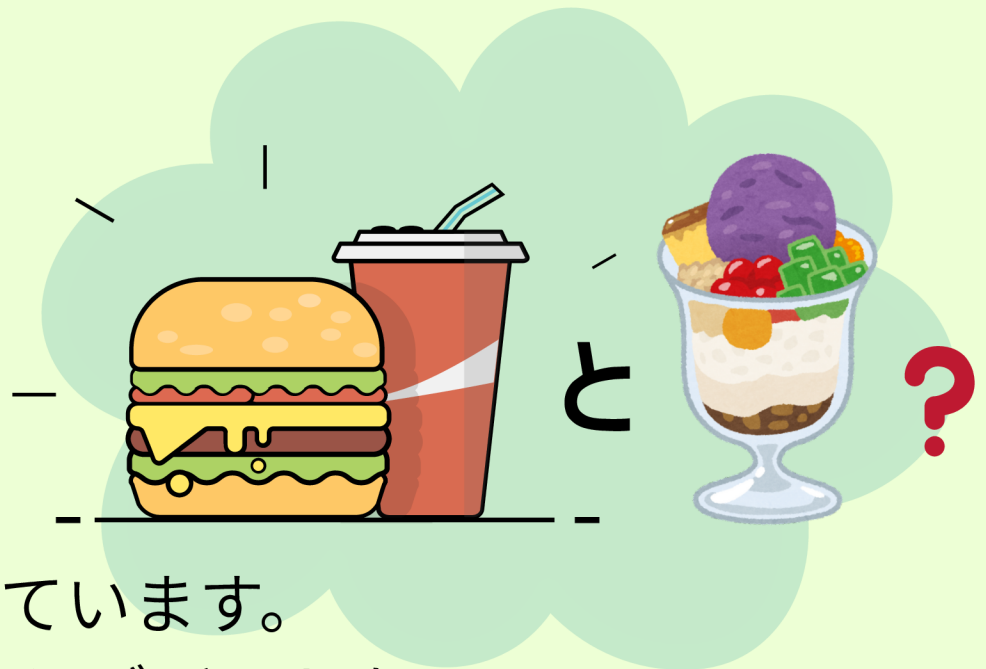


その 2.条件分岐 → お金、足りるかな

予算オーバーの時にどうするかを考えること



で



1,000円持っています。
ハンガーセットとデザートを買いたいのですが...

難しく
考えないでね～



① お金が足りなかったら？
デザートをガマンしましょう

コード例：

もし、(お金) < (欲しいものの合計) だったら
デザートを諦める ※下記のコードはイメージです

```
if ( Okane ) < ( hamburger_set + dessert ) then  
    hoshiimono = hamburger_set  
else  
    hoshiimono = hamburger_set + dessert
```

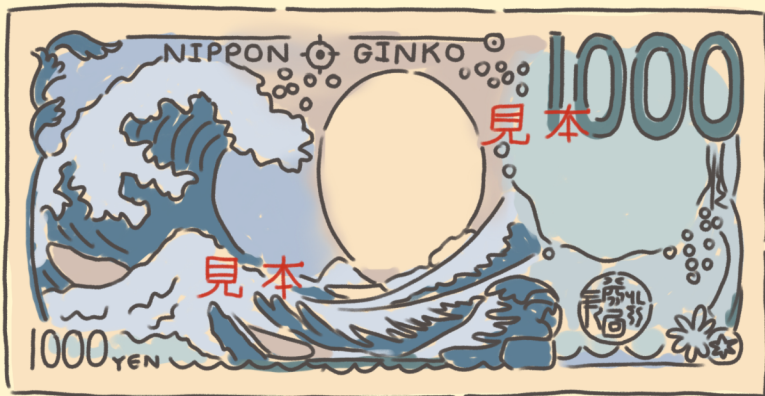
ここがPoint♥

何かの条件（お金が足りるか？）で
行動の内容をかえること

この考え方が「条件分岐」の基本だよ♥



その 3. 繰り返し → 同じことを何度も 繰り返すけど、いつやめるか？



で



を、何本も

1,000円持っています。
1本120円（税込）のジュースを
お腹いっぱい飲みたいのですが...

難しく
考えないでね～



😊 ジュースを飲む、を何度も繰り返します
いつやめるかを考えてみると...
3つの考え方ができそうですね

- ① 最初から飲む本数を決めておく
「繰り返し回数を指定する」と考えます
- ② お金が無くなったらやめる
「終了条件を指定する」と考えます
- ③ お腹がいっぱいになったらやめる
「永久ループから脱出する」と考えます

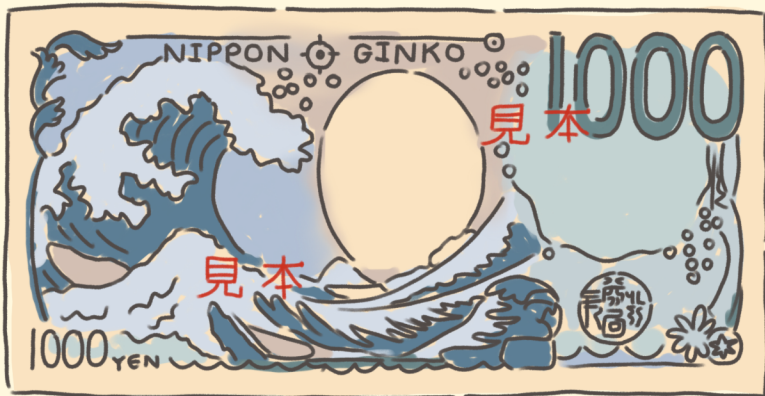
ここがPoint♥

同じ行動を何度も実行することを
「繰り返し」と言います。大切なのは、やめ時。
この考え方が「繰り返し」の基本だよ♥



その 3-1.繰り返し→繰り返し回数指定

繰り返すことができる回数を計算する



で



1本120円（税込）

は、8本まで

1,000円持っています。

1本120円（税込）のジュースは
8本まで買って飲むことができます

難しく
考えないでね～



ジュースを買える本数を計算して繰り返す

※下記のコードはイメージです

```
gokei = 0  
tanka = 120  
okane = 1000
```

①1,000円で何本買えるかを計算します

```
honsu = okane // tanka ← okane / tanka は、8.333  
okane // tanka は、8
```

②買った本数分、飲みましょう！

```
for kaisu in range(honsu):  
    gokei = gokei + tanka  
    print ("{}本め：{}円".format((kaisu+1), gokei))
```

ここがPoint♥

何回繰り返せるかを

計算してから繰り返します

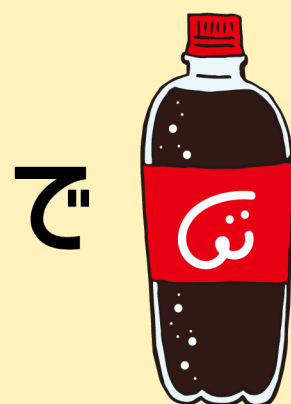
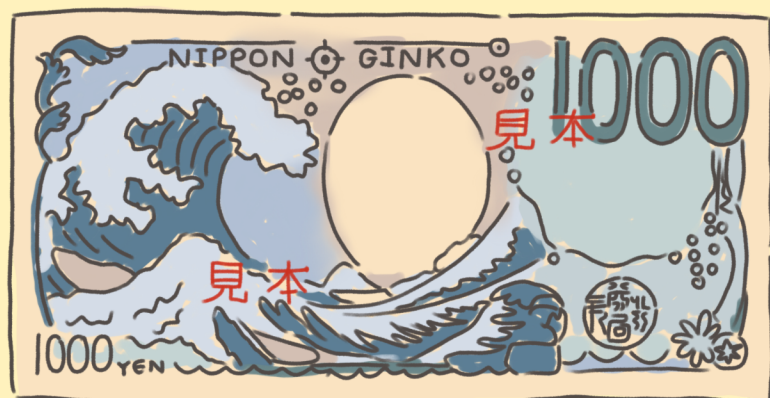
この考え方が「繰り返し」の基本①だよ♥



プログラミングのツボ

その 3-2.繰り返し → 終了条件の設定

繰り返しをやめる条件を設定する



1本120円（税込）

で を、1,000円まで

1,000円持っています。

1本120円（税込）のジュースを

1,000円を超えるまで買って飲むことができます

ジュースを買った金額を計算して繰り返す

※下記のコードはイメージです

難しく
考えないでね～



tanka = 120 # 単価

okane = 1000 # お金

gokei = 0 # 合計

--- 最初に1本買います ---

honsu = 1 # 買って飲んだ本数

gokei = gokei + tanka

1本は買います♥

①買った金額が1,000円まで飲みましょう！

while (gokei < okane): ← お金、足りてるかな？

print ("{}本め：{}円".format(honsu, gokei))

honsu = honsu + 1

gokei = gokei + tanka

ここがPoint♥

print ("お腹一杯")

繰り返しながら計算して

設定された条件を満足する間、繰り返します

この考え方が「繰り返し」の基本②だよ♥



プログラミングのツボ

その 3-3.繰り返し→ やめる判定をする

繰り返しをやめる判定をする



満腹
になったら



1本550ml

を飲むのをやめる

僕は2,000mlで満腹になります。

1本550mlのジュースを

満腹になるまで飲み続けます。

ジュースを飲んだmlを計算して繰り返す

※下記のコードはイメージです

難しく
考えないでね～

```
tippon    = 550    # 1本550ml
nonda     = 0      # 飲んだ料
manpuku   = 2000   # 満腹になるml
honsu     = 0      # 飲んだ本数
```

①満腹になったらやめましょう！

飲み続けます♥

```
while True:
```

```
    honsu = honsu + 1
```

```
    nonda = nonda + tippon
```

```
    if ( nonda >= manpuku ):
```

```
        break
```

```
    print ( "{}本め：{}ml".format(honsu, nonda))
```

満腹になった？

← やめる、は break だよ

ここがPoint♥

```
print ( "お腹一杯" )
```

繰り返しながら計算して

設定された条件に該当したら抜け出します

この考え方が「繰り返し」の基本③だよ♥



その 4. データ型

→ 1000と"1000"は違うものなんですよ

👉 数値

1000 と

文字 👉



どちらも1000ですが、違う扱いなんですよ

コンピュータの中では、
数値型データと文字型データがあります。
(実は、他にもいろいろあります。)

難しく
考えないでね～



数値型データ

→計算に使えます 例えば、1000

文字型データ

→計算に使えません 例えば、"1000"

「」(ダブルコーテーションマーク)が付くと
文字型データなんですよ。

ここがPoint♥

同じに見えてもプログラミングの中では
別々の扱いになる。計算できたり、できなかったり。
この考え方が「データ型」の基本だよ♥



その 4-1. データ型 (型変換:数値へ)

→ 文字は計算できません



単価に、100

数量に、20

と入力して計算するとエラーに！

**TypeError: can't multiply sequence by
non-int of type 'str'**

↑ 掛け算に文字型の値は使えません、という意味です

コンピュータの中では、
数値だけが計算できます

※キーボードで入力した値は文字となります

数値型データへの変換命令「int」を使います

※下記のコードはイメージです (Python)

```
tanka = input("単価？")
```

```
kosu = input("個数？")
```

```
gokei = int(tanka) * int(kosu)
```



```
print("単価：{}、個数{}、合計{}".format(tanka,  
kosu, gokei))
```

難しく
考えないでね～



ここがPoint♥

数字は文字、数字を数値に変換すると

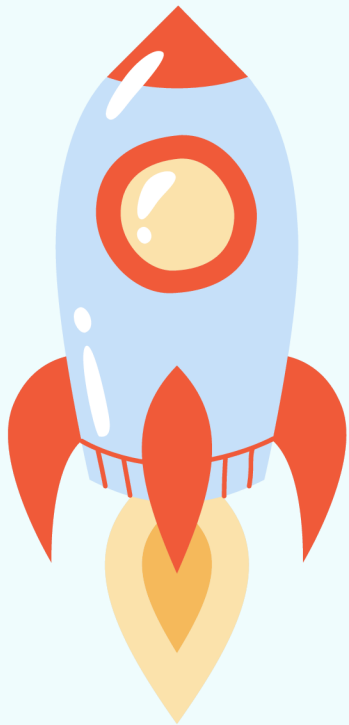
プログラミングの中で計算できるようになる。

この考え方が「データ型変換 (int)」の基本だよ♥



その 4-2. データ型 (型変換:文字へ)

→ 数値は文字列結合できません



ロケット名前に
`rocket="アポロ" + 13 + "号"`
と入力するとエラーに！

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

↑文字列同士の結合しかできません、という意味です
指定したデータに文字が含まれていると、
全部が文字か？ってチェックされます。

難しく
考えないでね～



コンピュータの中では、
文字だけが結合できます

※「"」(ダブルコーテーションマーク) が
付くと文字となります

文字型データへの変換命令「str」を使います
※下記のコードはイメージです (Python)

```
rocket="アポロ" + str(13) + "号"  
print(rocket)
```



ここがPoint♥

数値は計算用、数値を数字 (文字) に変換すると
プログラミングの中で結合できるようになる。
この考え方が「データ型変換 (str)」の基本だよ♥



プログラミングのツボ

その 5. 関数

→ 作成は1回、利用はずっと



電卓では...

「税込」ボタンを押すだけで、
税額を計算して
税込金額を表示してくれます



プログラミングでは...

この税込金額を計算する処理を
いつでも利用できるように

「関数」にすることができます

→ 「税込」計算関数

難しく
考えないでね～



関数のルール

- ①関数に名前を付ける
- ②引数（関数に渡す材料）を定義する
引数の初期値を指定することもできます
- ③戻り値（関数から返ってくる結果）を設定する
- ④関数の処理をプログラミングする
- ⑤誰かから呼び出してもらう
電卓の税込ボタンを押すイメージです
(関数は勝手に動きません)

ここがPoint♥

関数は同じ内容の処理を何度も
プログラミングしなくて良い便利な仕組みだよ
この考え方が「関数」の基本だよ♥



プログラミングのツボ

その 5-1. 関数（引数）

→ どんな計算をして欲しいかの材料



電卓では...

「100」の次に「税込」ボタン→110と表示
「200」の次に「税込」ボタン→220と表示
※消費税率は事前に設定



プログラミングでは...

引数(1)に、計算する金額
引数(2)に、消費税率
を指定して、税込価格を通知

→ 「税込金額」計算関数

※例えば、`keisan_zei_komi(500, 0.08)`
の計算結果は、540だよ

難しく
考えないでね～



引数の例

```
def keisan_zei_komi(price, tax_rate=0.1):
```

① `keisan_zei_komi`

→ 関数の名前

② `price`

→ 第一引数（計算対象金額）

③ `tax_rate`

→ 消費税率

指定されなかったら10%（0.1）←初期値といいます

ここがPoint♥

関数で同じ計算をする時に、その条件（金額、消費税率）を引数で指定すると計算結果が変わるよ

この考え方が「関数の引数」の基本だよ♥



プログラミングのツボ

その 5-2. 関数（戻り値）

→ 指定された計算の結果を通知します



電卓では...

「100」の次に「税込」ボタンを押すと...
→ 計算結果として110と表示



プログラミングでは...

引数の値で計算して、
戻り値で税込価格を通知

→ **「税込金額」 計算関数**

※ 例えば、`keisan_zei_komi(500, 0.08)`
の計算結果は、540だよ

難しく
考えないでね～



引数の例

```
def keisan_zei_komi(price, tax_rate=0.1):  
    zeikomi_kingaku = price * (1 + tax_rate)
```

```
    zeikomi_kingaku1 = round( zeikomi_kingaku )  
    # →roundは四捨五入
```

```
    return zeikomi_kingaku1
```

→returnは、計算結果を呼び出し元に通知する

※ちなみに...

```
    zeikomi_kingaku2 = int( zeikomi_kingaku )  
    # →intは切り捨てになるよ
```

ここがPoint♥

関数で同じ計算をする時に、指定された条件（金額、消費税率）で計算した結果を教えてあげることができるよ

この考え方が「関数の戻り値」の基本だよ♥



プログラミングのツボ

その 5-3. 関数（処理）

→ 指定された計算を実行します
処理部分は自分で自由に作れるよ（足し算・掛け算・平均など）

電卓では...



「100」の次に「税込」ボタンを押すと...
→ 計算結果として110と表示



プログラミングの関数では...

- ① 引数の値を受け取って、
- ② 計算をして、
- ③ 計算結果を通知します

→ 「税込金額」計算関数

関数の処理の例

```
def keisan_zei_komi(price, tax_rate=0.1):
    # defで関数が始まることを宣言します
    # ()で引数を受け取ります

    # 必要な処理を実行します
    zeikomi_kingaku = price * (1 + tax_rate)
    zeikomi_kingaku1 = round(zeikomi_kingaku)

    return zeikomi_kingaku1
# → returnは、計算結果を呼び出し元に通知します
```

実行例：① `print(keisan_zei_komi(1000))` → 1100
 実行例：② `print(keisan_zei_komi(500, 0.08))` → 540

難しく
考えないでね～



ここがPoint♥

関数は、引数を受け取り、計算をして計算結果を
呼び出し元に通知します（処理は関数の心臓部分！）

この考え方が「関数の処理」の基本だよ♥



プログラミングのツボ

その 5-4. 関数（呼び出し）

→ 作成された関数を実行します
関数は、どこから利用されて実行します



電卓では...

「100」の次に「税込」ボタンを押した時
→電卓に組込まれている計算が実行されます

ボタンを
押さないと
計算しないよ～

プログラミングの関数では...

- ①引数を指定して、
- ②関数を実行して
- ③計算結果を受け取ります

→ **「税込金額」が分かります**

難しく
考えないでね～



関数の処理の例

```
def keisan_zei_komi(price, tax_rate=0.1):
    zeikomi_kingaku = price * (1 + tax_rate)
    zeikomi_kingaku1 = round( zeikomi_kingaku )
    return zeikomi_kingaku1
```

```
# 📌関数を実行します
print ( keisan_zei_komi(1000) )
# 戻り値（計算結果）を表示します
```

実行例：①print(keisan_zei_komi(1000)) → 1100

実行例：②print(keisan_zei_komi(500, 0.08)) → 540

※関数の実行を直接print命令で利用できます

ここがPoint♥

関数は、どこから呼び出されたときに、
処理を実行することができます（勝手に動きません！）

この考え方が「関数呼び出し」の基本だよ♥

これで関数の基本4ステップが完成！

